

Programación competitiva

**Steven Halim
Felix Halim**

Programación competitiva

Manual para concursantes del ICPC y la IOI

OJBooks
Valladolid

Programación competitiva
Manual para concursantes del ICPC y la IOI

Primera edición en castellano

ISBN: 978-1-711-02481-3

Depósito legal: VA 1039-2019

Traducido de la edición en inglés de:
Competitive Programming 3
Steven Halim y Felix Halim
Lulu Enterprises, Inc., 2013, 13872395
ISBN: 580-0-095-81064-6

Edición original ©2013 by Steven Halim and Felix Halim

Presente edición, traducción al castellano y diseño de portada ©2019 by Miguel Revilla Rodríguez

Compuesto con \LaTeX

10 9 8 7 6 5 4 3 2 1

Fecha de revisión: 23 de diciembre de 2019

Este libro no puede ser reproducido en todo o en parte, ni transmitido por ningún medio mecánico, electrónico o análogo, incluyendo fotocopiado, escaneado o almacenado, sin autorización expresa y por escrito de los titulares de los derechos del texto original y de la traducción.

Índice general

Prólogo	XIII
Prefacio	XV
Perfil de los autores	XXVII
Abreviaturas	XXIX
Índice de tablas	XXXII
Índice de figuras	XXXVII
1. Introducción	1
1.1. Programación competitiva	1
1.2. Consejos para ser competitivo	3
1.2.1. Consejo 1: escribe más rápido	4
1.2.2. Consejo 2: identifica rápidamente el tipo de problema	5
1.2.3. Consejo 3: analiza el algoritmo	7
1.2.4. Consejo 4: domina los lenguajes de programación	13
1.2.5. Consejo 5: domina el arte de probar el código	15
1.2.6. Consejo 6: práctica y más práctica	18
1.2.7. Consejo 7: trabaja en equipo (para ICPC)	19
1.3. Primeros pasos: los problemas fáciles	20
1.3.1. Anatomía de un problema	20
1.3.2. Rutinas típicas de entrada/salida	20
1.3.3. Empieza el viaje	22
1.4. Los problemas <i>ad hoc</i>	24
1.5. Soluciones a los ejercicios no resaltados	30
1.6. Notas del capítulo	35

2. Estructuras de datos y bibliotecas	37
2.1. Introducción y motivación	37
2.2. Estructuras de datos lineales con bibliotecas integradas	39
2.3. Estructuras de datos no lineales con bibliotecas integradas	49
2.4. Estructuras de datos con nuestras propias bibliotecas	57
2.4.1. Grafo	57
2.4.2. Conjuntos disjuntos para unión-buscar	61
2.4.3. Árbol de segmentos	65
2.4.4. Árbol de Fenwick (binario indexado)	70
2.5. Soluciones a los ejercicios no resaltados	76
2.6. Notas del capítulo	79
3. Paradigmas de resolución de problemas	81
3.1. Introducción y motivación	81
3.2. Búsqueda completa	82
3.2.1. Búsqueda completa iterativa	83
3.2.2. Búsqueda completa recursiva	87
3.2.3. Consejos	90
3.3. Divide y vencerás	98
3.3.1. Usos destacados de la búsqueda binaria	98
3.4. Voraz	104
3.4.1. Ejemplos	104
3.5. Programación dinámica	111
3.5.1. Ilustración de programación dinámica	112
3.5.2. Ejemplos clásicos	122
3.5.3. Ejemplos no clásicos	132
3.6. Soluciones a los ejercicios no resaltados	139
3.7. Notas del capítulo	142
4. Grafos	143
4.1. Introducción y motivación	143
4.2. Recorrido de grafos	144
4.2.1. Búsqueda en profundidad	144
4.2.2. Búsqueda en anchura	146
4.2.3. Búsqueda de componentes conexos (grafo no dirigido)	147
4.2.4. Relleno por difusión - etiquetado/coloreado de componentes conexos	148
4.2.5. Orden topológico (grafo acíclico dirigido)	149
4.2.6. Comprobación de grafo bipartito	151
4.2.7. Comprobación de las propiedades de las aristas de un grafo	152

4.2.8. Búsqueda de puntos de articulación y puentes (grafo no dirigido)	154
4.2.9. Búsqueda de componentes fuertemente conexos (grafo dirigido)	158
4.3. Árbol de expansión mínimo (MST)	163
4.3.1. Introducción y motivación	163
4.3.2. Algoritmo de Kruskal	164
4.3.3. Algoritmo de Prim	165
4.3.4. Otras aplicaciones	167
4.4. Caminos más cortos de origen único (SSSP)	173
4.4.1. Introducción y motivación	173
4.4.2. SSSP en un grafo no ponderado	173
4.4.3. SSSP en un grafo ponderado	176
4.4.4. SSSP en un grafo con ciclo de peso negativo	180
4.5. Caminos más cortos entre todos los pares	184
4.5.1. Introducción y motivación	184
4.5.2. Explicación de la solución de DP de Floyd Warshall	185
4.5.3. Otras aplicaciones	188
4.6. Flujo de red	193
4.6.1. Introducción y motivación	193
4.6.2. Método de Ford Fulkerson	193
4.6.3. Algoritmo de Edmonds Karp	195
4.6.4. Modelado de grafos de flujo - parte 1	197
4.6.5. Otras aplicaciones	199
4.6.6. Modelado de grafos de flujo - parte 2	200
4.7. Grafos especiales	203
4.7.1. Grafo acíclico dirigido	203
4.7.2. Árbol	212
4.7.3. Grafo euleriano	214
4.7.4. Grafo bipartito	215
4.8. Soluciones a los ejercicios no resaltados	223
4.9. Notas del capítulo	227
5. Matemáticas	229
5.1. Introducción y motivación	229
5.2. Problemas matemáticos <i>ad hoc</i>	229
5.3. Clase BigInteger de Java	237
5.3.1. Características básicas	237
5.3.2. Características adicionales	239
5.4. Combinatoria	244
5.4.1. Sucesión de Fibonacci	245

5.4.2. Coeficientes binomiales	246
5.4.3. Números de Catalan	246
5.4.4. Notas sobre combinatoria en concursos de programación	248
5.5. Teoría de números	252
5.5.1. Números primos	252
5.5.2. Máximo común divisor y mínimo común múltiplo	254
5.5.3. Factorial	255
5.5.4. Búsqueda de factores primos con división por tentativa optimizada	255
5.5.5. Trabajo con factores primos	256
5.5.6. Funciones que implican factores primos	257
5.5.7. criba modificada	259
5.5.8. Aritmética modular	260
5.5.9. Euclídeo extendido: solución de la ecuación diofántica lineal	260
5.5.10. Comentarios sobre teoría de números en concursos de programación	261
5.6. Teoría de probabilidad	265
5.7. Búsqueda de ciclos	267
5.7.1. Soluciones utilizando estructuras de datos eficientes	267
5.7.2. Algoritmo de búsqueda de ciclos de Floyd	267
5.8. Teoría de juegos	270
5.8.1. Árbol de decisión	270
5.8.2. Mecanismos matemáticos para simplificar la solución	272
5.8.3. Juego del <i>nim</i>	273
5.9. Soluciones a los ejercicios no resaltados	273
5.10. Notas del capítulo	276
 6. Procesamiento de cadenas	277
6.1. Introducción y motivación	277
6.2. Habilidades básicas de procesamiento de cadenas	278
6.3. Problemas de procesamiento de cadenas <i>ad hoc</i>	280
6.4. Coincidencia de cadenas	285
6.4.1. Soluciones con bibliotecas	286
6.4.2. Algoritmo de Knuth-Morris-Pratt (KMP)	286
6.4.3. Coincidencia de cadenas en una rejilla bidimensional	289
6.5. Procesamiento de cadenas con programación dinámica	290
6.5.1. Alineación de cadenas (distancia de edición)	290
6.5.2. Subsecuencia común más larga	293
6.5.3. Procesamiento de cadenas no clásico con DP	294
6.6. Trie/Árbol/Array de sufijos	296
6.6.1. Trie de sufijos y aplicaciones	296

6.6.2.	Árbol de sufijos	297
6.6.3.	Aplicaciones del árbol de sufijos	298
6.6.4.	<i>Array</i> de sufijos	302
6.6.5.	Aplicaciones del <i>array</i> de sufijos	308
6.7.	Soluciones a los ejercicios no resaltados	314
6.8.	Notas del capítulo	318
7.	Geometría (computacional)	319
7.1.	Introducción y motivación	319
7.2.	Objetos de geometría básicos con bibliotecas	321
7.2.1.	Objetos sin dimensión: puntos	321
7.2.2.	Objetos unidimensionales: líneas	323
7.2.3.	Objetos bidimensionales: círculos	329
7.2.4.	Objetos bidimensionales: triángulos	331
7.2.5.	Objetos bidimensionales: cuadriláteros	334
7.3.	Algoritmos en polígonos con bibliotecas	338
7.3.1.	Representación de polígonos	338
7.3.2.	Perímetro de un polígono	338
7.3.3.	Área de un polígono	339
7.3.4.	Comprobación de si un polígono es convexo	339
7.3.5.	Comprobación de si un punto está dentro de un polígono	340
7.3.6.	Corte de un polígono con una línea recta	342
7.3.7.	Búsqueda de la envolvente convexa de un conjunto de puntos	343
7.4.	Soluciones a los ejercicios no resaltados	349
7.5.	Notas del capítulo	352
8.	Materias más avanzadas	353
8.1.	Introducción y motivación	353
8.2.	Técnicas de búsqueda más avanzadas	353
8.2.1.	<i>Backtracking</i> con máscara de bits	354
8.2.2.	<i>Backtracking</i> con poda intensa	359
8.2.3.	Búsqueda estado-espacio con BFS o Dijkstra	360
8.2.4.	Encuentro en el medio (búsqueda bidireccional)	362
8.2.5.	Búsqueda informada: A* e IDA*	364
8.3.	Técnicas de DP más avanzadas	368
8.3.1.	DP con máscara de bits	369
8.3.2.	Recopilación de parámetros comunes (de DP)	370
8.3.3.	Gestión de valor de parámetros negativos con técnica de desplazamiento	371
8.3.4.	¿MLE? Prueba a utilizar un BST equilibrado como tabla recordatoria	372

8.3.5. ¿MLE/TLE? Usa una mejor representación de estados	373
8.3.6. ¿MLE/TLE? Abandona un parámetro, recuperándolo de los otros	374
8.4. Descomposición de problemas	378
8.4.1. Dos componentes: búsqueda binaria de la respuesta y otro	378
8.4.2. Dos componentes: con RSQ/RMQ estática unidimensional	381
8.4.3. Dos componentes: procesamiento previo de un grafo y DP	381
8.4.4. Dos componentes: con grafos	382
8.4.5. Dos componentes: con matemáticas	383
8.4.6. Dos componentes: búsqueda completa y geometría	383
8.4.7. Dos componentes: con estructura de datos eficiente	383
8.4.8. Tres componentes	384
8.5. Soluciones a los ejercicios no resaltados	391
8.6. Notas del capítulo	392
9. Temas poco habituales	393
9.1. Problema 2-SAT	394
9.2. Problema de la galería de arte	396
9.3. Problema del viajante bitónico	397
9.4. Emparejamiento de paréntesis	398
9.5. Problema del cartero chino	399
9.6. Problema del par más cercano	401
9.7. Algoritmo de Dinic	402
9.8. Fórmulas o teoremas	403
9.9. Algoritmo de eliminación gausiana	404
9.10. Emparejamiento de grafos	407
9.11. Distancia de círculo máximo	411
9.12. Algoritmo de Hopcroft Karp	412
9.13. Caminos independientes y arista-disjuntos	413
9.14. Índice de inversión	414
9.15. Problema de Josefo	415
9.16. Movimientos del caballo	416
9.17. Algoritmo de Kosaraju	417
9.18. Ancestro común mínimo	419
9.19. Construcción de un cuadrado mágico (de tamaño impar)	421
9.20. Multiplicación de cadenas de matrices	422
9.21. Potencia de matrices	424
9.22. Conjunto independiente ponderado máximo	429
9.23. Flujo (máximo) de coste mínimo	430
9.24. Cobertura de caminos mínima en un DAG	431

9.25. Ordenación de tortitas	432
9.26. Algoritmo de factorización de enteros rho de Pollard	436
9.27. Calculadora posfija y conversión	437
9.28. Números romanos	439
9.29. Problema de selección	441
9.30. Algoritmo más rápido para el camino más corto	444
9.31. Ventana corredera	445
9.32. Ordenación en tiempo lineal	448
9.33. Estructura de datos de tabla dispersa	450
9.34. Torres de Hanoi	452
9.35. Notas del capítulo	454
A. uHunt	457
B. Créditos	461
Bibliografía	463
Índice alfabético	467

Prólogo

El martes 11 de noviembre de 2003, a las 03:55:57 UTC, recibí el siguiente mensaje:

“Debo decirle, en pocas palabras, que con el UVa Online Judge ha nacido una nueva CIVILIZACIÓN y que, con los libros que escribe (se refería a “Desafíos de programación” [60], junto a Steven S. Skiena), inspira a los soldados para que sigan marchando. Le deseo una larga vida de servicio a la Humanidad, para que pueda crear programadores sobrehumanos.”

Aunque la exageración resultaba evidente, me dió qué pensar. Yo tenía un sueño: crear una comunidad alrededor del proyecto que había iniciado entorno a mi labor docente en la Universidad de Valladolid, en la que gente de todo el mundo trabajase unida por un mismo ideal. Con un poco de búsqueda, rápidamente encontré todo un ecosistema de páginas web.

Para mí, ‘Methods to Solve’, de Steven Halim, un joven estudiante de Indonesia, era una de las páginas más impresionantes. Me inspiraba pensar que el sueño podría llegar a convertirse en una realidad, porque en esa página encontré el resultado del duro trabajo de un genio de los algoritmos y la informática. Además, sus objetivos declarados coincidían plenamente con el corazón de mi sueño: servir a la Humanidad. Para hacerlo todo incluso mejor, Steven tenía un hermano con intereses y capacidades similares, Felix Halim.

Es una pena que lograr que una auténtica colaboración fructifique consuma tanto tiempo, pero así es la vida. Por suerte, seguimos trabajando juntos, paralelamente, avanzando hacia la consecución del sueño. Este libro es una prueba de ello.

No se me ocurre un complemento mejor para el Online Judge. En estas páginas se utilizan innumerables ejemplos tomados del juez UVa, seleccionados con mimo y categorizados por tipos y por las técnicas que los solucionan, proporcionando una ayuda inestimable a los usuarios.

Es evidente que el libro “Programación competitiva” es perfecto para aquellos programadores que quieren mejorar su desempeño en competiciones del ICPC y la IOI. Ambos autores han competido en esos concursos, primero como estudiantes y, después, como entrenadores. Pero también este libro resulta perfecto para los recién llegados pues, como Steven y Felix dicen en la introducción, ‘este es un libro para leer varias veces’. Cuando lo hayas hecho, descubrirás que no solo eres un mejor programador, sino una persona más feliz.

Miguel A. Revilla (1951-2018), Universidad de Valladolid
Creador del UVa Online Judge
<http://onlinejudge.org>

Prefacio

Este debe ser el libro de cabecera de todo programador competitivo. Dominar el contenido de la presente obra es una condición necesaria, aunque quizás no suficiente, para dar el salto desde ser un programador normal a convertirse en uno de los mejores del mundo.

Entre los lectores a quienes va dirigido este libro se encuentran:

1. Estudiantes universitarios que compiten en las fases regionales del International Collegiate Programming Contest [66] y en la propia final mundial.
2. Estudiantes de secundaria que compiten en la International Olympiad in Informatics (IOI) [34], incluyendo sus fases locales.
3. Preparadores que buscan material de formación para sus estudiantes [24].
4. Cualquiera interesado en la resolución de problemas de programación. Existen muchos concursos a los que pueden acceder quienes ya quedan fuera del ámbito del ICPC, entre ellos TopCoder Open, Google CodeJam, Internet Problem Solving Contest (IPSC), etc.

Requisitos previos

Este libro *no* está escrito para programadores novatos. Su objetivo son lectores que tienen, al menos, conocimientos básicos en metodología de programación, están familiarizados con, como mínimo, los lenguajes C/C++ o Java (preferiblemente ambos), han cursado formación básica en estructuras de datos y algoritmos (estas materias se encuentran, normalmente, entre las que se imparten en el primer año de ingeniería informática en las universidades), y comprenden análisis algorítmicos sencillos (al menos, la notación *Big O*). En esta edición¹, se ha añadido contenido que permite utilizar el libro como *lectura complementaria* de un curso básico de *estructuras de datos y algoritmos*, en especial las implementaciones eficientes, sus visualizaciones de algoritmos integradas y varios ejercicios escritos y de programación.

¹A lo largo del texto, se hacen numerosas referencias a diferentes ediciones del libro, así como a la fecha de la última revisión de la presente. Recordamos al lector que esta primera edición en castellano, que tiene en sus manos, está basada en la tercera en inglés, publicada el 24 de mayo de 2013. Si bien, en contadas ocasiones, se ha alterado ligeramente el texto para corregir aspectos que resultarían anacrónicos en un libro publicado en 2019, hemos procurado mantener la integridad del original, conscientes de que la diferencia temporal entre la versión en inglés y la traducida podría causar una ligera confusión al lector. No nos cabe duda, sin embargo, de la absoluta vigencia de los temas tratados y de que, estas pequeñas ‘disfunciones temporales’, no pasan de ser meramente anecdóticas (N. del T.).

A los concursantes del ICPC

Sabemos que, probablemente, será difícil ganar una fase regional del ICPC utilizando solo el contenido de este libro. Aunque hemos incluido muchos materiales nuevos (con respecto a las ediciones anteriores), somos conscientes de que se requiere mucho más de lo que se ofrece aquí para lograr tal objetivo. Las notas de los capítulos contienen referencias adicionales, para aquellos lectores hambrientos de más conocimiento. Creemos, sin embargo, que tu equipo obtendrá resultados mucho mejores en futuros concursos, después de dominar este contenido. Esperamos que este libro sirva tanto de inspiración como de motivación para el viaje de 3 o 4 años, como concursante en el ICPC, durante tu etapa universitaria.

A los concursantes de la IOI

Muchos de los consejos para los concursantes del ICPC también se pueden aplicar aquí. El ICPC y el temario de la IOI son muy similares, con la excepción de que la IOI no es un concurso de velocidad y, *de momento*, excluye los temas mostrados en la tabla 1. Puedes ignorar estos temas hasta que llegues a la universidad (cuando te unas a los equipos universitarios del ICPC). Sin embargo, conocer estas técnicas por adelantado podría resultar beneficioso, ya que algunas de las tareas de la IOI serán más sencillas con conocimientos adicionales.

Sabemos que no se puede ganar una medalla de la IOI solo por conocer el contenido de la *versión actual* de este libro. Aunque confiamos en haber incluido muchas de las materias de la IOI (con la esperanza de permitirte lograr una puntuación respetable), somos muy conscientes de que las tareas más recientes requieren capacidades de resolución de problemas excepcionales, así como una creatividad tremenda, virtudes que son imposibles de transmitir a través de un libro de texto. Esta obra puede proporcionar conocimientos, pero el trabajo duro te corresponde realizarlo a tí. Con la práctica llega la experiencia, y con la experiencia se adquiere la capacidad. Así que, sigue practicando.

Tema	En este libro
Estructuras de datos: conjuntos disjuntos para unión-buscar	Sección 2.4.2
Grafos: búsqueda de SCC, flujo de red, grafos bipartitos	Secciones 4.2.1, 4.6.3, 4.7.4
Matemáticas: BigInteger, teoría de probabilidad, juego del nim	Secciones 5.3, 5.6, 5.8
Procesamiento de cadenas: árboles y arrays de sufijos	Sección 6.6
Materias más avanzadas: A*/IDA*	Sección 8.2
Muchos temas poco habituales	Capítulo 9

Tabla 1: Materias *todavía* no incluídas en el temario de la IOI [20]

A los profesores y preparadores

Este libro se utiliza en el curso CS3233 - ‘Programación competitiva’, impartido por Steven en la Escuela de Informática de la Universidad Nacional de Singapur. El CS3233 se desarrolla a lo largo de 13 semanas lectivas, utilizando el plan de estudios de la tabla 2. Las diapositivas en PDF del curso (sólo la versión pública) están disponibles en la página web que complementa a

este libro. Los profesores, o preparadores, que utilicen el curso como referencia, tienen la libertad de modificar el plan de estudios, para ajustarlo a las necesidades de sus alumnos. Al final de cada capítulo, se pueden encontrar pistas o soluciones breves de los ejercicios **no marcados con un asterisco** que aparecen en el libro. Algunos de los ejercicios escritos, **marcados con un asterisco**, son muy difíciles y no tienen ni pistas ni soluciones. Se podrían utilizar como preguntas de examen o problemas para un concurso (obviamente, el profesor deberá resolverlos).

Este libro también se utiliza como lectura complementaria del curso CS2010 - ‘Estructuras de datos y algoritmos’, también impartido por Steven, mayormente por la implementación de varios algoritmos, su visualización y los ejercicios escritos y de programación.

Semana	Tema	En este libro
1	Introducción	Cap. 1, sec. 2.2, 5.2, 6.2, 6.3, 7.2
2	Estructuras de datos y bibliotecas	Capítulo 2
3	Búsqueda completa (divide y vencerás, voraz)	Secciones 3.2-3.4; 8.2
4	Programación dinámica 1 (ideas básicas)	Secciones 3.5; 4.7.1
5	Programación dinámica 2 (más técnicas)	Secciones 5.4; 5.6; 6.5; 8.3
6	Concurso por equipos a mitad del semestre	Capítulos 1-4; parte del cap. 9
-	Vacaciones a mitad del semestre	(trabajo en casa)
7	Grafos 1 (flujo de red)	Sección 4.6; parte del cap. 9
8	Grafos 2 (emparejamiento)	Sección 4.7.4; parte del cap. 9
9	Matemáticas (introducción)	Capítulo 5
10	Procesamiento de cadenas (nocións básicas, array de sufijos)	Capítulo 6
11	Geometría (computacional) (bibliotecas)	Capítulo 7
12	Materias más avanzadas	Sección 8.4; parte del cap. 9
13	Concurso por equipos final	Capítulos 1-9 y más
-	No hay examen final	-

Tabla 2: Plan de estudios del curso CS3233 de Steven

Para cursos de *estructuras de datos* y *algoritmos*

El contenido de este libro ha sido extendido en esta edición, de forma que los *cuatro primeros* capítulos resulten más accesibles para estudiantes de informática de *primer año*. Los temas y ejercicios que hemos considerado de mayor dificultad y, por tanto, innecesariamente desalentadores para los principiantes, se han trasladado al capítulo 8, ahora más completo, o al nuevo capítulo 9. De esta forma, los estudiantes que comienzan sus estudios de informática, no se sentirán intimidados por esos primeros cuatro capítulos.

El capítulo 2 ha sido actualizado profundamente. Anteriormente, la sección 2.2 constaba únicamente de una lista informal de estructuras de datos clásicas y sus bibliotecas. Ahora hemos extendido la redacción y añadido muchos ejercicios escritos, para que este libro se pueda utilizar también como base de un curso de *estructuras de datos*, sobre todo en los detalles relativos a la *implementación*.

Los cuatro paradigmas de resolución de problemas tratados en el capítulo 3 aparecerán frecuentemente en cursos de *algorítmica*. El texto de ese capítulo ha sido ampliado y corregido, para ayudar a los estudiantes recién llegados a la informática.

Hay secciones del capítulo 4 que también pueden utilizarse como lectura adicional, o guía de *implementación*, para mejorar la *matemática discreta* [56, 15], o para un curso básico de *algorítmica*. También hemos incluido nuevos puntos de vista, para tratar las técnicas de programación dinámica, como algoritmos en DAGs. Temas que, por desgracia, siguen siendo materias poco comunes en muchos textos sobre ciencias de la computación.

A todos los lectores

Dada su diversidad en cobertura y profundidad de tratamiento, este libro *no debe* ser leído una única vez, sino que pretende ser consultado con frecuencia. Hay una buena cantidad de ejercicios escritos (≈ 238) y de programación (≈ 1675), distribuidos entre casi todas las secciones. Puedes ignorar dichos ejercicios en una primera aproximación, si te resultan muy difíciles, o si requieren más técnicas y aprendizaje, y volver sobre ellos después de completar el resto de la lectura. Resolver esos ejercicios fortalecerá tu comprensión de los conceptos tratados, ya que, normalmente, implican aplicaciones interesantes y variantes de los temas descritos. Debes intentar resolverlos en algún momento, ya que hacerlo no será tiempo perdido.

Creemos que este libro es, y será, relevante para muchos estudiantes de secundaria y universitarios. Los concursos de programación, como el ICPC o la IOI, permanecerán en el tiempo, al menos durante muchos años. Los nuevos estudiantes deben intentar entender e internalizar los conocimientos básicos presentados, antes de buscar nuevos retos. Sin embargo, el término ‘básicos’ no debe llevar a confusión, como podrás comprobar después de leer la tabla de contenidos.

El propósito de este libro es claro: queremos mejorar las capacidades de programación y resolución de problemas de todos los lectores y, con ello, *subir el listón* de los concursos de programación, como el ICPC o la IOI. Con más concursantes dominando el contenido de esta obra, esperamos que el año 2010 (cuando se publicó la primera edición) marque un punto de inflexión, que resulte en una mejora acelerada de los estándares en los concursos de programación. Esperamos ayudar a más equipos a resolver más (≥ 2) problemas en los futuros ICPC y contribuir a que más concursantes logren mejores (≥ 200) puntuaciones en las futuras IOI. También deseamos ver a muchos preparadores del ICPC y de la IOI de todo el mundo (especialmente en el sudeste asiático y la subregión de la Península del Pacífico) adoptar este texto, por la ayuda que proporciona en el dominio de temas que los estudiantes no pueden ignorar en el contexto de concursos de programación. Si logramos nuestra pretendida proliferación del conocimiento ‘mínimo’ requerido en la programación competitiva, habremos alcanzado nuestro objetivo de elevar el nivel del conocimiento humano y nosotros, como autores de este libro, recibiremos la mayor satisfacción por ello.

Convenciones

En este libro se incluye mucho código fuente en C/C++ y, también, algo de Java (sobre todo en la sección 5.3). Cuando aparezca, estará impreso en esta **tipografía monoespaciada**.

En el código fuente de C/C++ hemos adoptado el uso frecuente de `typedef` y macros, características que se utilizan habitualmente por programadores competitivos por su comodidad, brevedad y velocidad en la programación. No es posible utilizar estas técnicas en Java, ya que no tiene características análogas. Mostramos, a continuación, algunos ejemplos de nuestros atajos:

```

1 // Eliminar algunas advertencias durante la compilación (solo para VC++)
2 #define _CRT_SECURE_NO_DEPRECATE
3
4 // Atajos para tipos de datos "comunes" en concursos
5 typedef long long ll;           // los comentarios integrados en el código
6 typedef pair<int, int> ii;       // aparecen justificados a la derecha
7 typedef vector<ii> vii;
8 typedef vector<int> vi;
9 #define INF 1000000000          // 1000M, más seguro que 2000M para Floyd Marshall
10
11 // Opciones de memset habituales
12 memset(memo, -1, sizeof memo); // inicializar tabla memoización de DP con -1
13 memset(arr, 0, sizeof arr);    // vaciado de un array de enteros

```

Los siguientes atajos aparecen frecuentemente en nuestros códigos de C/C++ y Java:

```

1 i++;                                // para simplificar: i = i+1;
2 ans = a ? b : c;                    // para simplificar: if (a) ans = b; else ans = c;
3 ans += val;                         // para simplificar: ans = ans+val; y sus variantes
4 index = (index+1) % n;              // index++; if (index >= n) index = 0;
5 index = (index+n-1) % n;            // index--; if (index < 0) index = n-1;
6 int ans = (int)((double)d + 0.5);   // redondeo al entero más cercano
7 ans = min(ans, new_computation);    // atajo para mínimo/máximo
8 // forma alternativa, no usada en el libro: ans <?= new_computation;
9 // algún código utiliza los operadores de bits && (AND) y || (OR)

```

Categorización de problemas

En la fecha de entrega de esta edición, entre Steven y Felix han resuelto 1903 problemas del Online Judge (el $\approx 46,45\%$ del total de problemas contenidos en la plataforma). En este libro tratamos, y hemos categorizado, unos ≈ 1675 de ellos. Desde finales de 2011, algunos problemas del Live Archive se encuentran integrados en el Online Judge. En este libro, utilizamos las numeraciones *de ambos*, pero la clave de referencia principal, que encontramos en el índice alfabético, será el número de problema UVa.

Los problemas se han asignado a las diferentes categorías, según una estrategia de ‘balanceo de carga’: si un problema se puede clasificar en más de una categoría, se presentará en aquella con un menor número de entradas. Por ello, el lector puede encontrar que, algunos problemas, aparecen reflejados en una categoría ‘incorrecta’, pues podrían no corresponderse, en apariencia, con la técnica utilizada para resolverlos. Lo que sí podemos garantizar es que, si un problema X aparece en una categoría Y, *hemos* podido resolverlo en esos términos.

También hemos limitado cada categoría a un máximo de 25 problemas, repartiéndolos en varias de ellas cuando ha sido necesario.

Todos los problemas que hemos resuelto se encuentran enumerados en el índice alfabético, al final del libro. Si el lector busca alguno en concreto, debería hacerlo ahí en vez de recorrer

capítulo a capítulo, lo que supondrá un ahorro de tiempo considerable. El índice contiene una lista de problemas de UVa/LA, ordenados por su número (puedes encontrarlos mediante búsqueda binaria), y las páginas en las que son comentados (así como las estructuras de datos y/o algoritmos necesarios).

Puedes utilizar esta categorización como medio de entrenamiento. Resolver, al menos, alguno de los problemas de cada categoría (especialmente aquellos resaltados como **obligatorios** *), es un medio excelente para diversificar las habilidades de resolución de problemas. Para no resultar abrumadores al lector, hemos limitado el número de problemas resaltados a un máximo de tres por categoría.

Cambios en la segunda edición en inglés

Ha habido cambios *sustanciales* entre las primera y segunda ediciones en inglés de este libro. Como autores, hemos adquirido nuevos conocimientos y resuelto cientos de problemas de programación, durante el año que separa a ambas. Además, hemos recibido comentarios de los lectores, especialmente de los alumnos del segundo semestre del curso 2010/2011 de la clase de CS3233 de Steven, y hemos incorporado no pocas de esas sugerencias.

Enumeramos, a continuación, algunos de los cambios más significativos:

- El cambio más evidente lo encontramos en el diseño del libro. Cada página contiene ahora mucha más información. La segunda edición se presenta con interlineado simple, en vez de con el interlineado de 1,5 de la primera. También se ha mejorado la ubicación de las figuras más pequeñas, para obtener un resultado visual más compacto. Esto ha evitado un aumento innecesario del número de páginas al extender el contenido.
- Hemos corregido pequeños errores en nuestros ejemplos de código fuente (tanto en los que aparecen en el libro, como los que se pueden encontrar en la página web). Además, los comentarios de esos códigos son ahora mucho más aclaratorios.
- Se han corregido un buen número de erratas y errores gramaticales o de estilo.
- Además de mejorar el tratamiento de muchas estructuras de datos, algoritmos y problemas de programación, hemos añadido las siguientes materias *nuevas* a cada capítulo:
 1. Hay muchos más problemas *ad hoc* al inicio del libro (sección 1.4).
 2. Un conjunto breve de técnicas con booleanos (manipulación de bits, sección 2.2), grafos implícitos (2.4.1) y estructuras de datos con el árbol de Fenwick (2.4.4).
 3. Más programación dinámica: una explicación más clara de la DP de abajo a arriba, la solución para el problema LIS en $O(n \log k)$, la mochila 0-1/suma de subconjuntos y el problema del viajante con DP, usando máscaras de bits (sección 3.5.2).
 4. Una reorganización de la materia relativa a grafos en: recorrido de grafos (tanto DFS como BFS), árbol de expansión mínimo, caminos más cortos (de origen único y entre todos los pares), flujo máximo y grafos especiales. Los nuevos temas incluyen el algoritmo MST de Prim, el uso de la DP como mecanismo de recorrido en DAG implícitos (sección 4.7.1), grafos eulerianos (sección 4.7.3) y el algoritmo de aumento de camino (sección 4.7.4).

5. Una reorganización de técnicas matemáticas (capítulo 5) en: *ad hoc*, BigInteger de Java, combinatoria, teoría de números, teoría de probabilidad, búsqueda de ciclos, teoría de juegos (nuevo) y potencias de una matriz cuadrada (nuevo). Cada tema se ha vuelto a escribir desde cero, para darle mayor claridad.
 6. Elementos de procesamiento de cadenas básico (sección 6.2), más problemas relacionados con cadenas (sección 6.3), incluyendo coincidencia de cadenas (sección 6.4) y una explicación mejorada del árbol y del *array* de sufijos (sección 6.6).
 7. Más bibliotecas de geometría (capítulo 7), especialmente en lo relativo a puntos, líneas y polígonos.
 8. Un nuevo capítulo 8, con temas como la descomposición de problemas, técnicas de búsqueda avanzadas (A^* , búsqueda limitada en profundidad, profundidad iterativa, IDA *), técnicas de DP avanzadas (más técnicas de máscara de bits, el problema del cartero chino, una recopilación de estados de DP habituales, un tratamiento sobre mejores estados de DP y algunos problemas de DP más difíciles).
- Muchas de las figuras del libro se han redibujado y mejorado. Además, se han añadido otras nuevas, para ayudar a explicar los conceptos con mayor claridad.
 - La primera edición se escribió, principalmente, desde el punto de vista del concursante del ICPC y programador de C++. En la segunda, hemos procurado equilibrarnos más hacia la perspectiva de la IOI. También ha mejorado sustancialmente la presencia de Java. Sin embargo, por el momento no hemos incluido otros lenguajes de programación.
 - La página web de Steven ‘Methods to Solve’ se ha incorporado completamente al libro, en forma de ‘consejos de una línea’ para cada problema, además del índice alfabético del final del libro. De esta forma, alcanzar los 1000 problemas resueltos en el Online Judge ya no es un sueño inalcanzable (de hecho, creemos que es perfectamente posible para cualquier estudiante de cuarto año de ingeniería informática que lo intente *en serio*).
 - Algunos ejemplos de la primera edición utilizaban problemas de programación antiguos. En la segunda, estos han sido sustituidos por otros más actuales.
 - Steven y Felix han resuelto ≈ 600 problemas más entre el Online Judge y el Live Archive, que han sido incorporados al libro. También hemos añadido muchos más ejercicios escritos, acompañados de consejos y soluciones breves.
 - Hemos adaptado de Wikipedia [71], u otras fuentes, pequeños perfiles sobre los inventores de los algoritmos y estructuras de datos tratados. Nunca viene mal ampliar nuestros conocimientos de historia.

Cambios en la tercera edición en inglés

Nos hemos concedido dos años para preparar un número de mejoras *sustanciales* y materias adicionales para la tercera edición de este libro. Éstas son las más importantes:

- La tercera edición utiliza un cuerpo de texto ligeramente más grande (12 puntos), en comparación al de la segunda (11 puntos), lo que supone un incremento del 9 %. Esperamos que esto ayude a hacer el texto más fácil de leer. También hemos aumentado el tamaño

de las figuras. Estas decisiones, sin embargo, han hecho aumentar el número de páginas. También hemos ajustado los márgenes, para mejorar el aspecto general².

- Hemos procurado equilibrar visualmente el comienzo de cada sección, para aportar una experiencia de lectura más cómoda.
- Hemos añadido *muchos más* ejercicios escritos a lo largo del libro, y los hemos clasificado como **no resaltados** (para la autoevaluación del lector, con consejos o soluciones al final de cada capítulo) y **resaltados *** (para los que no proporcionamos solución). Estos ejercicios escritos se encuentran ahora incluidos en sus secciones correspondientes.
- Steven y Felix han resuelto ≈ 477 problemas más entre el Online Judge y el Live Archive que han sido, consecuentemente, añadidos al libro. Con ello, nos mantenemos en un respetable $\approx 50\%$ (para ser precisos, $\approx 46,45\%$) de ejercicios resueltos, de entre todos los que ofrecen las plataformas, teniendo en cuenta que, durante este tiempo, también ha aumentado el número de problemas que ofrecen. Estos problemas nuevos aparecen listados en *cursiva*. Algunos de ellos han sustituido a otros que, anteriormente, considerábamos **obligatorios ***. Todos los ejercicios de programación aparecen ahora al final de cada sección.
- Ahora tenemos pruebas de que los estudiantes *aplicados* puede lograr resolver ≥ 500 problemas del Online Judge en un solo semestre universitario (4 meses), gracias a este libro.
- Las materias *nuevas* (o revisadas), capítulo a capítulo, son:
 1. El capítulo 1 contiene una introducción más amable para el lector recién llegado a la programación competitiva. Tratamos formatos de entrada/salida estrictos en problemas de programación típicos, y las técnicas más habituales para abordarlos.
 2. Hemos añadido la estructura de datos lineal ‘deque’ a la sección 2.2. El capítulo 2 contiene ahora un tratamiento más detallado de casi todas las estructuras de datos mencionadas, especialmente en las secciones 2.3 y 2.4.
 3. Encontraremos una explicación más detallada de varias técnicas de búsqueda completa en el capítulo 3: bucles anidados, generación iterativa de subconjuntos/permutoaciones y *backtracking* recursivo. Además, un truco interesante para escribir y mostrar soluciones de DP de arriba abajo y el algoritmo de Kadane para la suma de rango unidimensional máxima.
 4. En el capítulo 4, hemos revisado las etiquetas blancas/grises/negras (según [7]) para adecuarlas a su nomenclatura estándar, renombrando ‘flujo máximo’ a ‘flujo de red’ en el proceso. También hacemos referencia al artículo original del autor del algoritmo, para una mejor comprensión de las ideas empleadas. Y tenemos nuevos diagramas del DAG implícito en problemas de DP clásicos, en la sección 3.5.
 5. En el capítulo 5, hemos ampliado el tratamiento de problemas matemáticos *ad hoc*, incluyendo una operación interesante con BigInteger de Java, `isProbablePrime`,

²Estas consideraciones no se aplican a esta edición en castellano. El libro original está publicado en versiones de tamaño DIN A4 y DIN A5, utilizando la misma paginación en ambos. Para la versión traducida, hemos optado por un único formato, el precioso *Crown Quarto* (de 189×246 milímetros), que tiene no pocas connotaciones históricas en lo que a publicación de textos se refiere. Para satisfacer al lector más curioso, nos permitimos aportar que el texto está compuesto utilizando la tipografía *Computer Modern* en cuerpo de 10 puntos, para los segmentos monoespaciados hemos elegido *Inconsolata LGC* y para la rotulación de capítulos y secciones, además de otros elementos, utilizamos *Myriad Pro Condensed* en distintos tamaños (N. del T.).

añadido/extendido varias fórmulas de combinatoria habituales y algoritmos de criba modificados, extendido/revisado las secciones sobre teoría de probabilidad (5.6), búsqueda de ciclos (5.7) y teoría de juegos (5.8).

6. En el capítulo 6, hemos reescrito la sección 6.6, para dar una mejor explicación sobre los *array*/*árbol*/*trie* de sufijos, al volver a introducir el concepto del carácter de terminación.
7. Hemos dividido el capítulo 7 en dos secciones nucleares y mejorado la calidad del código de biblioteca.
8. El capítulo 8 incluye ahora los temas más complejos que aparecían en los capítulos 1 a 7 en la segunda edición. Algunos de ellos, incluso, han sido desplazados al nuevo capítulo 9. Además, hemos añadido el tratamiento de rutinas de *backtracking* más complejas, búsqueda estado-espacio, encuentro en el medio, el truco de utilizar un BST equilibrado como tabla recordatoria y una sección mucho más completa sobre descomposición de problemas.
9. El nuevo capítulo 9 incluye varios temas poco habituales, que pueden aparecer esporádicamente en concursos de programación. Algunos son sencillos, pero la mayoría son complejos y podrían ser determinantes en la clasificación de un concurso.

Páginas web de apoyo

Este libro se acompaña de una página web oficial, en cpbook.net, en la que se puede obtener una copia descargable del código fuente de ejemplo y de las diapositivas en PDF (en su versión *pública*) utilizadas por Steven en su curso CS3233.

La herramienta uhunt.felix-halim.net, incluye todos los ejercicios de programación que aparecen en este libro y se encuentran en el Online Judge en onlinejudge.org.

Además, desde la tercera edición, contamos con visualizaciones interactivas de muchos de los algoritmos, en la página web visualgo.net.

Agradecimientos de la primera edición en inglés

De Steven:

Quiero dar las gracias a

- Dios, Jesucristo y el Espíritu Santo, por concederme el talento y la pasión por la programación competitiva.
- Mi encantadora esposa, Grace Suryani, por permitirme utilizar nuestro precioso tiempo en este proyecto.
- Mi hermano menor y coautor, Felix Halim, por compartir muchas estructuras de datos, algoritmos y trucos de programación, que han mejorado el contenido de este libro.
- Mi padre, Lin Tjie Fong, y mi madre, Tan Hoey Lan, por educarnos y animarnos a desempeñar con éxito nuestros estudios y nuestro trabajo.
- La Escuela de Informática de la UNS, por contratarme y permitirme impartir el módulo CS3233 - ‘Programación competitiva’, del que nace este libro.
- Los profesores, antiguos y actuales, de la Universidad Nacional de Singapur, que han modelado mis capacidades de programación y como preparador: profesor Andrew Lim Leong Chye, profesor asociado Tan Sun Teck, Aaron Tan Tuck Choy, profesor asociado Sung Wing Kin, Ken y el doctor Alan Cheng Holun.
- Mi amigo Ilham Winata Kurnia, por revisar el manuscrito de la primera edición.
- Los compañeros del curso CS3233 y los preparadores del ICPC en la Universidad Nacional de Singapur: Su Zhan, Ngo Minh Duc, Melvin Zhang Zhiyong y Bramadia Ramadhana.
- Mis alumnos del CS3233 del segundo semestre del curso 2008/2009, que me inspiraron para recopilar mis notas de clase, y mis alumnos del segundo semestre del curso 2009/2010, que revisaron el contenido de la primera edición de este libro y proporcionaron la contribución inicial del Live Archive

Agradecimientos de la segunda edición en inglés

De Steven:

Además, quiero dar las gracias a

- Los primeros ≈ 550 compradores de la primera edición del libro, a fecha del 1 de agosto de 2011 (la cifra ya no está actualizada). Vuestro apoyo ha sido la fuerza que nos impulsa.
- Un compañero profesor auxiliar del curso CS3233 en la UNS: Victor Loh Bo Huai.
- Mis alumnos del curso CS3233 del segundo semestre del curso 2010/2011, que han aportado tanto a los aspectos técnicos como de presentación de la segunda edición. En orden alfabético: Aldrian Obaja Muis, Bach Ngoc Thanh Cong, Chen Juncheng, Devendra Goyal,

Fikril Bahri, Hassan Ali Askari, Harta Wijaya, Hong Dai Thanh, Koh Zi Chun, Lee Ying Cong, Peter Phandi, Raymond Hendy Susanto, Sim Wenlong Russell, Tan Hiang Tat, Tran Cong Hoang, Yuan Yuan y a otro estudiante que prefiere el anonimato.

- Los correctores de pruebas, siete de los alumnos del curso CS3233 mencionados (aparecen subrayados) y Tay Wenbin.
- Por último, pero no por ello menos importante, renuevo mi agradecimiento a mi esposa, Grace Suryani, por permitirme emplear mi tiempo en otro periodo de edición, mientras estaba embarazada de nuestra primera hija: Jane Angelina Halim.

Agradecimientos de la tercera edición en inglés

De Steven:

Nuevamente, quiero dar las gracias a

- Los ≈ 2000 compradores de la segunda edición del libro, a fecha del 24 de mayo de 2013 (la cifra ya no se actualiza). Gracias.
- Mis compañeros profesores del curso CS3233 en la UNS durante los dos últimos años: Harta Wijaya, Trinh Tuan Phuong y Huang Da.
- Mis alumnos del curso CS3233 del segundo semestre del curso 2011/2012, cuyas aportaciones en los aspectos técnicos y formales a la tercera edición del libro han sido fundamentales. En orden alfabético: Cao Sheng, Chua Wei Kuan, Han Yu, Huang Da, Huynh Ngoc Tai, Ivan Reinaldo, John Goh Choo Ern, Le Viet Tien, Lim Zhi Qin, Nalin Ilango, Nguyen Hoang Duy, Nguyen Phi Long, Nguyen Quoc Phong, Pallav Shinghal, Pan Zhengyang, Pang Yan Han, Song Yangyu, Tan Cheng Yong Desmond, Tay Wenbin, Yang Mansheng, Zhao Yang, Zhou Yiming y otros dos alumnos que prefieren el anonimato.
- Los correctores de pruebas: seis de los alumnos del curso CS3233 del segundo semestre del curso 2011/2012 (aparecen subrayados) y Hubert Teo Hua Kian.
- Mis alumnos del curso CS3233 del segundo semestre del curso 2012/2012, que han realizado aportaciones, tanto técnicas como de diseño, en la tercera edición del libro. Por orden alfabético: Arnold Christopher Koroa, Cao Luu Quang, Lim Puay Ling Pauline, Erik Alexander Qwick Faxaa, Jonathan Darryl Widjaja, Nguyen Tan Sy Nguyen, Nguyen Truong Duy, Ong Ming Hui, Pan Yuxuan, Shubham Goyal, Sudhanshu Khemka, Tang Binbin, Trinh Ngoc Khanh, Yao Yujian, Zhao Yue y Zheng Naijia.
- El Centro de la UNS para el desarrollo de la enseñanza y el aprendizaje (CDTL), por aportar la financiación inicial para construir la página web de visualización de algoritmos.
- Mi esposa, Grace Suryani, y mi hija, Jane Angelina, por su amor y compañía.

Por un mejor futuro para la Humanidad,

STEVEN y FELIX HALIM

Singapur, 24 de mayo de 2013

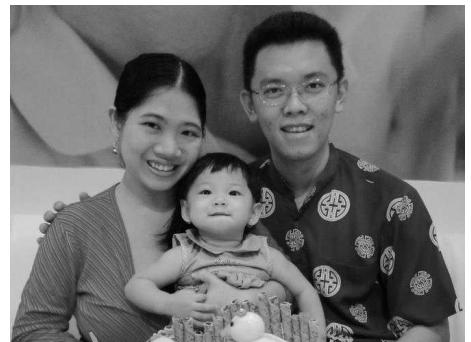
Perfil de los autores

Steven Halim, PhD³

stevenhalim@gmail.com

Steven Halim es profesor en la Escuela de Informática de la Universidad Nacional de Singapur, donde imparte varios cursos de programación, desde metodología básica hasta algoritmos y estructuras de datos intermedios, programación web y el módulo ‘Competitive Programming’, en el que utiliza este libro. Es entrenador de los equipos del ICPC de la UNS y del equipo nacional de la IOI de Singapur. Ha competido en varias fases regionales del ICPC como estudiante (Singapur 2001, Aizu 2003, Shanghai 2004). Hasta la fecha, junto a otros entrenadores de la UNS, ha llevado a dos equipos a competir en la final mundial del ICPC (2009-2010 y 2012-2013), así como ha logrado dos oros, seis platas y siete bronces en la IOI (2009-2012).

Está felizmente casado con Grace Suryani Tioso, con quien tiene una hija, Jane Angelina Halim.



Felix Halim, PhD⁴

felix.halim@gmail.com

Felix Halim es doctor por la Escuela de Informática de la Universidad Nacional de Singapur. En el ámbito de los concursos de programación, cuenta con un palmarés más vistoso que el de su hermano mayor. Fue concursante de la IOI de 2002 (representando a Indonesia). Sus equipos del ICPC (en la Universidad Bina Nusantara), compitieron en la fase regional de Manila en 2003, 2004 y 2005, logrando los puestos 10, 6 y 10, respectivamente. En su último año, su equipo finalmente ganó la fase regional del ICPC de Kaohsiung 2006, y alcanzó la final mundial en Tokio 2007 (puesto 44). Hoy día compite habitualmente en TopCoder, donde ha alcanzado la categoría de programador amarillo. En la actualidad, trabaja en Google, en Mountain View, Estados Unidos.



³Tesis doctoral: “An Integrated White+Black Box Approach for Designing and Tuning Stochastic Local Search Algorithms”, 2009.

⁴Tesis doctoral: “Solving Big Data Problems: from Sequences to Tables and Graphs”, 2012.

Abreviaturas

A*	: A asterisco
ACM	: Assoc of Computing Machinery
AC	: Aceptado
APSP	: Cam. más cort. entre todos los pares
AVL	: Adelson-Velskii Landis (BST)
BNF	: Notación de Backus-Naur
BFS	: Búsqueda en anchura
BI	: Big Integer
BIT	: Árbol indexado binario
BST	: Árbol de búsqueda binaria
CC	: Cambio de monedas
CCW	: En contra de las agujas del reloj
CF	: Frecuencia acumulada
CH	: Envoltorio convexa
CS	: Ciencias de la computación
CW	: A favor de las agujas del reloj
DAG	: Grafo acíclico dirigido
DAT	: Tabla de direccionamiento directo
D&C	: Divide y vencerás
DFS	: Búsqueda en profundidad
DLS	: Búsqueda de profundidad limitada
DP	: Programación dinámica
DS	: Estructura de datos
ED	: Distancia de edición
FIFO	: Primero en entrar, primero en salir
FT	: Árbol de Fenwick
GCD	: Máximo común divisor
ICPC	: Intl Collegiate Prog Contest
IDS	: Búsqueda de profundidad iterativa
IDA*	: A* de profundidad iterativa
IOI	: Intl Olympiad in Informatics
IPSC	: Internet Problem Solving Contest
KISS	: Hazlo breve y sencillo
LA	: Live Archive [33]
LCA	: Ancestro común mínimo

LCM	: Mínimo común múltiplo
LCP	: Prefijo común más largo
LCS₁	: Subsecuencia común más larga
LCS₂	: Subcadena común más larga
LIFO	: Último en entrar, primero en salir
LIS	: Subsecuencia creciente más larga
LRS	: Subcadena repetida más larga
LSB	: Bit menos significativo
MCBM	: Empar. bipart. de card. máxima
MCM	: Multiplic. de cadenas de matrices
MCMF	: Flujo máximo de coste mínimo
MIS	: Conjunto independiente máximo
MLE	: Límite de memoria superado
MPC	: Cobertura de caminos mínima
MSB	: Bit más significativo
MSSP	: Cam. más cort. de origen múltiple
MST	: Árbol de expansión mínimo
MWIS	: Conj. indep. ponderado máximo
MVC	: Cobertura de vértices mínima
OJ	: Online Judge
PE	: Error de presentación
RB	: Rojo-negro (BST)
RMQ	: Consulta de rango mín. (o máx.)
RSQ	: Consulta de suma de rango
RTE	: Error en tiempo de ejecución
SSSP	: Caminos más cortos de origen único
SA	: <i>Array</i> de sufijos
SPOJ	: Sphere Online Judge
ST	: Árbol de sufijos
STL	: Standard Template Library
TLE	: Límite de tiempo superado
USACO	: USA Computing Olympiad
UVa	: Universidad de Valladolid [49]
WA	: Respuesta incorrecta
WF	: Final mundial

Índice de tablas

1.	Materias <i>todavía</i> no incluídas en el temario de la IOI [20]	xvi
2.	Plan de estudios del curso CS3233 de Steven	xvii
1.1.	Tipos de problemas en los ICPC regionales de Asia recientes	6
1.2.	Tipos de problemas (versión compacta)	6
1.3.	Ejercicio: clasificar estos problemas del Online Judge	7
1.4.	Referencia rápida del ‘peor algoritmo AC’ para varios tamaños de entrada n	10
2.1.	Ejemplo de tabla de frecuencia acumulada	70
2.2.	Comparativa entre el árbol de segmentos y el árbol de Fenwick	75
3.1.	Ejecución del método de bisección en la función de ejemplo	101
3.2.	Tabla de decisión de programación dinámica	120
3.3.	UVa 108 - Maximum Sum	123
3.4.	Resumen de los problemas de DP clásicos de esta sección	136
3.5.	Comparativa de técnicas de resolución de problemas (orientativa)	142
4.1.	Lista de terminología de grafos importante	143
4.2.	Tabla de decisión de algoritmos de recorrido de grafos	161
4.3.	Tabla de DP de Floyd Warshall	187
4.4.	Tabla de decisión del algoritmo para SSSP/APSP	191
4.5.	Caracteres utilizados en UVa 11380	201
5.1.	Lista de <i>algunos</i> términos matemáticos utilizados en este capítulo	230
5.2.	Parte 1: búsqueda de $k\lambda$, $f(x) = (3 \times x + 1) \% 4$, $x_0 = 7$	268
5.3.	Parte 2: búsqueda de μ	268
5.4.	Parte 3: búsqueda de λ	269

6.1.	I/D: antes y después de ordenar; $k = 1$; aparece la ordenación inicial	305
6.2.	I/D: antes/después de ordenar; $k = 2$; se intercambian ‘GATAGACA’ y ‘GACA’	305
6.3.	Antes/Después de ordenar; $k = 4$; sin cambios	306
6.4.	Coincidencia de cadenas utilizando un <i>array</i> de sufijos	310
6.5.	Cálculo del LCP dado el SA de T = ‘GATAGACA\$’	311
6.6.	<i>Array</i> de sufijos, LCP y propietario de T = ‘GATAGACA\$CATA#’	312
9.1.	Reducción de LCA a RMQ	420
9.2.	Ejemplos de expresiones infijas, prefijas y posfijas	437
9.3.	Ejemplo de cálculo posfijo	438
9.4.	Ejemplo de ejecución del algoritmo del patio de maniobras	439

Índice de figuras

1.1. Ilustración de UVa 10911 - Forming Quiz Teams	2
1.2. Online Judge e ICPC Live Archive	19
1.3. Plataforma de entrenamiento de USACO y Sphere Online Judge	19
1.4. Algunas de las fuentes que han inspirado a los autores de este libro	36
2.1. Visualización de máscaras de bits	41
2.2. Ejemplo de BST	49
2.3. Visualización de un montículo (máximo)	51
2.4. Visualización de la estructura de datos de un grafo	58
2.5. Ejemplos de grafos implícitos	60
2.6. <code>unionSet(0, 1) → (2, 3) → (4, 3)</code> e <code>isSameSet(0, 4)</code>	63
2.7. <code>unionSet(0, 3) → findSet(0)</code>	63
2.8. Árbol de segmentos del array $A = \{18, 17, 13, 19, 15, 11, 20\}$ y $\text{RMQ}(1, 3)$	66
2.9. Árbol de segmentos del array $A = \{18, 17, 13, 19, 15, 11, 20\}$ y $\text{RMQ}(4, 6)$	67
2.10. Actualización del array A a $\{18, 17, 13, 19, 15, \mathbf{99}, 20\}$	67
2.11. Ejemplo de <code>rsq(6)</code>	72
2.12. Ejemplo de <code>rsq(3)</code>	72
2.13. Ejemplo de <code>adjust(5, 1)</code>	73
3.1. 8-Queens	87
3.2. UVa 10360 [49]	91
3.3. My Ancestor (los 5 caminos de la raíz a una hoja están ordenados)	99
3.4. Visualización de UVa 410 - Station Balance	105
3.5. UVa 410 - Observaciones	106
3.6. UVa 410 - Solución voraz	107
3.7. UVa 10382 - Watering Grass	108

3.8.	DP de abajo a arriba (las columnas 21 a 200 no aparecen por razón de espacio)	118
3.9.	Subsecuencia creciente máxima	125
3.10.	Cambio de monedas	128
3.11.	Un grafo completo	130
3.12.	Ilustración de Cutting Sticks	134
4.1.	Grafo de ejemplo	145
4.2.	UVa 11902	146
4.3.	Animación de ejemplo de BFS	147
4.4.	Ejemplo de un DAG	150
4.5.	Animación de un DFS cuando se ejecuta sobre el grafo de ejemplo de la figura 4.1	153
4.6.	Introducimos dos atributos adicionales a la DFS: <code>dfs_num</code> y <code>dfs_low</code>	155
4.7.	Búsqueda de puntos de articulación con <code>dfs_num</code> y <code>dfs_low</code>	156
4.8.	Búsqueda de puentes, también con <code>dfs_num</code> y <code>dfs_low</code>	156
4.9.	Ejemplo de un grafo dirigido y sus SCC	158
4.10.	Ejemplo de un problema de MST	163
4.11.	Animación del algoritmo de Kruskal para un problema de MST	165
4.12.	Animación del algoritmo de Prim en el grafo de la izquierda de la figura 4.10	166
4.13.	De izquierda a derecha: MST, ST ‘máximo’, SS ‘mínimo’, ‘bosque’ MS	167
4.14.	Segundo mejor árbol de expansión (de UVa 10600 [49])	168
4.15.	Búsqueda del segundo mejor árbol de expansión de un MST	168
4.16.	Minimax (UVa 10048 [49])	169
4.17.	Animación de Dijkstra en un grafo ponderado (de UVa 341 [49])	177
4.18.	Peso negativo	180
4.19.	Bellman Ford puede detectar la presencia de ciclos negativos (de UVa 558 [49])	181
4.20.	Explicación de Floyd Warshall 1	185
4.21.	Explicación de Floyd Warshall 2	186
4.22.	Explicación de Floyd Warshall 3	186
4.23.	Explicación de Floyd Warshall 4	187
4.24.	Ilustración del flujo máximo (UVa 820 [49] - problema E en la final ICPC 2000)	194
4.25.	El método Ford Fulkerson puede ser lento si se implementa con DFS	195
4.26.	¿Cuál es el valor del flujo máximo de estos tres grafos residuales?	197
4.27.	Grafo residual de UVa 259 [49]	198
4.28.	Técnica de división de vértices	200
4.29.	Algunos casos de prueba de UVa 11380	201

4.30. Modelado del grafo de flujo	201
4.31. Grafos especiales (izquierda a derecha): DAG, árbol, euleriano, bipartito	204
4.32. El camino más largo en este DAG	205
4.33. Ejemplo de conteo de caminos en un DAG, de abajo a arriba	206
4.34. Ejemplo de conteo de caminos en un DAG, de arriba a abajo	207
4.35. El grafo general dado (a la izquierda) se convierte en un DAG	208
4.36. El grafo/árbol general dado (a la izquierda) se convierte en un DAG	209
4.37. Cambio de monedas como caminos más cortos en un DAG	210
4.38. Mochila 0-1 como caminos más largos en un DAG	211
4.39. UVa 10943 como conteo de caminos en un DAG	211
4.40. A: SSSP (parte de APSP); B1-B2: diámetro del árbol	213
4.41. Euleriano	214
4.42. El problema de emparejamiento bipartito se puede reducir a uno de flujo máximo	216
4.43. Variantes de MCBM	216
4.44. Algoritmo de aumento de camino	218
5.1. Izquierda: triangulación de un polígono convexo, derecha: caminos monótonos	248
5.2. Árbol de decisión de una partida del ‘Euclid’s Game’	271
5.3. Árbol de decisión parcial de una partida de ‘A multiplication game’	272
6.1. Ejemplo: A = ‘ACAATCC’ y B = ‘AGCATGC’ (puntuación de la alineación = 7)	292
6.2. <i>Trie</i> de sufijos	296
6.3. Sufijos, <i>trie</i> de sufijos y árbol de sufijos de T = ‘GATAGACA\$’	298
6.4. Coincidencia de cadenas de T = ‘GATAGACA\$’ con varios patrones	299
6.5. Subcadena repetida más larga de T = ‘ <u>GATAGACA\$</u> ’	300
6.6. Árbol de sufijos generalizado de T_1 = ‘ <u>GATAGACA\$</u> ’ y T_2 = ‘ <u>CATA#</u> ’ y su LCS	301
6.7. Ordenación de los sufijos de T = ‘GATAGACA\$’	303
6.8. Árbol y <i>array</i> de sufijos de T = ‘GATAGACA\$’	303
7.1. Rotación de (10, 3) 180 grados contra las agujas del reloj, alrededor de (0, 0)	322
7.2. Distancia a la línea (izquierda) y al segmento (centro); producto vectorial (derecha)	325
7.3. Círculos	329
7.4. Un círculo por dos puntos y un radio	330
7.5. Triángulos	331
7.6. Circunferencias inscrita y exinscrita de un triángulo	333

7.7.	Cuadriláteros	334
7.8.	Izquierda: polígono convexo, derecha: polígono cóncavo	339
7.9.	Arriba-izquierda: dentro, arriba-derecha: también dentro, abajo: fuera	341
7.10.	Izquierda: antes del corte, derecha: después del corte	342
7.11.	Analogía de la goma elástica para encontrar la envolvente convexa	344
7.12.	Orden de un conjunto de 12 puntos por ángulo en relación al pivote (punto 0)	345
7.13.	La parte principal del método de Graham	346
7.14.	Explicación de un círculo que pasa por 2 puntos y un radio	351
8.1.	Problema 5 Queens: el estado inicial	355
8.2.	Problema 5 Queens: después de colocar la primera reina	356
8.3.	Problema 5 Queens: después de colocar la segunda reina	356
8.4.	Problema 5 Queens: después de colocar la tercera reina	357
8.5.	Problema 5 Queens: después de colocar las cuarta y quinta reinas	357
8.6.	Visualización de UVa 1098 - Robots on Ice	359
8.7.	Caso 1: ejemplo cuando s está a dos pasos de t	363
8.8.	Caso 2: ejemplo cuando s está a cuatro pasos de t	363
8.9.	Caso 3: ejemplo cuando s está a cinco pasos de t	364
8.10.	Juego del 15	364
8.11.	El camino de descenso	373
8.12.	Ilustración de ACM ICPC WF2010 - J - Sharing Chocolate	375
8.13.	Pista de atletismo (de UVa 11646)	380
8.14.	Ilustración de ACM ICPC WF2009 - A - A Careful Approach	385
9.1.	Los grafos de implicación de los ejemplos 1 (izquierda) y 2 (derecha)	395
9.2.	El problema del viajante estándar frente al bitónico	397
9.3.	Ejemplo del problema del cartero chino	400
9.4.	Las cuatro variantes de emparejamiento de grafos más comunes en concursos	408
9.5.	Caso de ejemplo de UVa 10746: 3 emparejamientos con coste mínimo = 40	409
9.6.	I: esfera, C: semiesfera y círculo máximo, D: $g\bar{c}Distance$ (arco $A - B$)	411
9.7.	Comparativa de caminos independientes y caminos arista-disjuntos máximos	413
9.8.	Ejemplo de un árbol T con raíz con $n = 10$ vértices	419
9.9.	Estrategia de construcción de un cuadrado mágico para un n impar	422
9.10.	Ejemplo del problema de flujo máximo de coste mínimo (MCMF) (UVa 10594 [49])	430
9.11.	Cobertura de caminos mínima en un DAG (de UVa 1201 [49])	432

9.12. Ejemplo de eliminación en un árbol AVL (se elimina el 7)	444
9.13. Explicación de RMQ(i, j)	451
A.1. Estadísticas de Steven el 24 de mayo de 2013	457
A.2. A la caza del siguiente problema más fácil mediante ‘dacu’	458
A.3. Podemos reproducir concursos pasados, mediante los ‘concursos virtuales’	458
A.4. Los ejercicios de programación de este libro están integrados en uHunt	459
A.5. Progreso de Steven y Felix en el Online Judge (2000-2013)	459

Capítulo 1

Introducción

¡Quiero competir en la final mundial del ICPC!
— Un estudiante aplicado

1.1 Programación competitiva

La premisa fundamental de la programación competitiva dice así: “enfrentarse a problemas de programación bien conocidos y resolverlos lo más rápidamente posible”.

Analicemos los términos. Mencionar ‘problemas de programación bien conocidos’ implica que, en la programación competitiva, nos enfrentamos a problemas *ya resueltos* y *no* a problemas de investigación (en los que la solución todavía no se conoce). Hay quien ya los ha resuelto antes (como mínimo, el autor del problema). ‘Resolverlos’ significa que debemos¹ mejorar nuestros conocimientos informáticos hasta el nivel requerido, para ser capaces de producir un código que funcione y que encuentre la solución, al menos, en términos de obtener la *misma* salida en los casos de prueba secretos² del autor del problema, dentro del tiempo límite estipulado. La necesidad de resolver los problemas ‘lo más rápidamente posible’ es, precisamente, donde reside el elemento competitivo, ya que la velocidad es un reto muy natural en los humanos.

Hay que tener en cuenta que destacar en la programación competitiva, es un medio, y *no* un fin en sí mismo. El auténtico objetivo es formar mejores programadores, que estén más preparados para producir mejor software y para enfrentarse, en un futuro, a retos de investigación más complejos. Los fundadores del International Collegiate Programming Contest (ICPC) [66] tienen esa visión y nosotros, los autores, estamos de acuerdo con ella. Con este libro, queremos hacer nuestra pequeña contribución a que las generaciones presentes y futuras sean más competitivas en la solución de los problemas de programación ya conocidos y que encontramos, frecuentemente, en las ediciones más recientes tanto del ICPC como de la International Olympiad in Informatics (IOI).

¹En algunos concursos de programación se compite en equipo, alentando así el trabajo en grupo ya que, en la vida real, los ingenieros de software no suelen trabajar solos.

²Al ocultar los casos de prueba finales en el planteamiento del problema, la programación competitiva alienta a quienes tratan de resolverlo a ejercitarse en su fortaleza mental, para tratar de encontrar todos los casos extremos que sean capaces, y comprobar que sus programas los resuelven. Ésta es una situación típica para los ingenieros de software, que deben probar sus programas en profundidad, hasta estar seguros de que cumplen con los requisitos establecidos por los clientes.

Ejemplo

Problema número 10911 del UVa Online Judge [49] (Forming Quiz Teams)

Descripción resumida del problema

Tenemos (x, y) como coordenadas de la casa de un estudiante en un plano bidimensional. Hay $2N$ estudiantes y queremos **emparejarlos** en N grupos. d_i será la distancia entre las casas de 2 estudiantes en el grupo i . Formar N grupos de forma que $cost = \sum_{i=1}^N d_i$ sea **mínimo**. Devolver el valor $cost$ mínimo. Restricciones: $1 \leq N \leq 8$ y $0 \leq x, y \leq 1000$.

Ejemplo de entrada

$N = 2$; las coordenadas de las $2N = 4$ casas son $\{1, 1\}$, $\{8, 6\}$, $\{6, 8\}$ y $\{1, 3\}$.

Ejemplo de salida

$cost = 4,83$.

¿Te ves capaz de resolver este problema? Y, si es así, ¿cuánto tiempo calculas que tardarás en tener el código funcionando? Piénsalo y trata de no seguir leyendo inmediatamente.

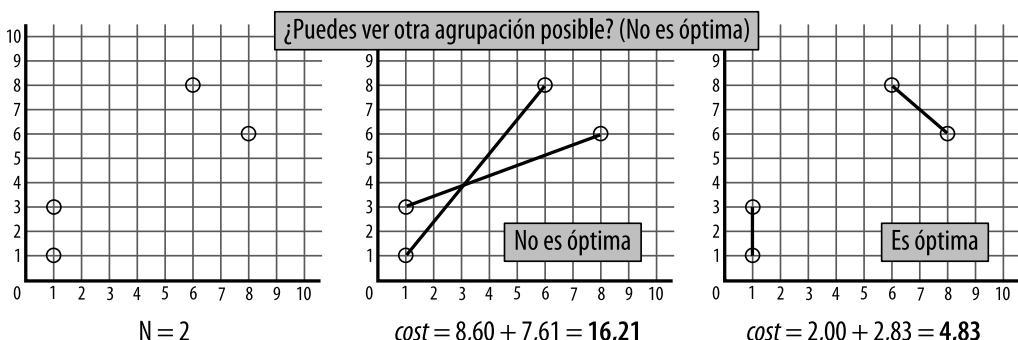


Figura 1.1: Ilustración de UVa 10911 - Forming Quiz Teams

Ahora, pregúntante con cuál de las siguientes descripciones te identificas más. Si todavía no estás familiarizado con toda la terminología utilizada en este capítulo, te recomendamos que lo vuelvas a leer una vez terminado el libro.

- Programador no competitivo A (alias *el difuso*):
Paso 1: lee el problema y se queda confuso (el problema es nuevo para él).
Paso 2: intenta programar algo a partir de las entradas y salidas de ejemplo.
Paso 3: se da cuenta de que *ninguno* de sus intentos es **aceptado (AC)**:
 - **Voraz** (sección 3.4): emparejar repetidamente a los dos estudiantes restantes con las distancias de separación más cortas será una **respuesta incorrecta (WA)**.
 - **Búsqueda completa** ingenua: utilizar *backtracking* recursivo (sección 3.2) y probar todas las parejas posibles terminará con **tiempo límite superado (TLE)**.
- Programador no competitivo B (abandona):
Paso 1: lee el problema y se da cuenta de que ya lo ha visto antes (sección 9.10). Pero también recuerda que nunca aprendió a resolverlo. No conoce la solución utilizando **programación dinámica (DP)** (sección 3.5).
Paso 2: ignora el problema y pasa al siguiente.

- (Otro) programador no competitivo C (lento):

Paso 1: lee el problema y se da cuenta de que es difícil: '**peso mínimo que se ajuste perfectamente a un grafo de peso general**'. Sin embargo, como el tamaño de entrada es pequeño, este problema se puede resolver con DP. El estado de la DP es una **máscara de bits**, que describe el estado de emparejamiento y, al emparejarse los estudiantes desemparejados i y j , se activarán dos bits i y j en la máscara (sección 8.3.1).

Paso 2: programa la rutina de E/S, escribe la DP recursiva, prueba, **depura...**

Paso 3: pasan *tres horas* y su solución logra AC (resuelve los casos de prueba secretos).
- Programador competitivo D:
Completa todos los pasos del programador no competitivo C en ≤ 30 minutos.
- Programador muy competitivo E:
Un programador muy competitivo (por ejemplo, los programadores con nivel rojo en TopCoder [32]) resolvería este problema 'bien conocido' en ≤ 15 minutos.

Ejercicio 1.1.1

La estrategia voraz del programador no competitivo A, mencionado antes, funciona para el caso de ejemplo mostrado en la Figura 1.1. Vamos a darle un ejemplo *mejor*.

Ejercicio 1.1.2

Analizar la complejidad de tiempo de la solución ingenua de búsqueda completa del programador no competitivo A, para entender por qué recibe un veredicto de TLE.

Ejercicio 1.1.3*

En realidad, una solución inteligente de *backtracking* recursivo *con poda* puede resolver este problema. Resuelve el problema sin utilizar una tabla de DP.

1.2 Consejos para ser competitivo

Si quieras convertirte en los programadores competitivos D o E mencionados, es decir, si quieras llegar a participar (a través de las selecciones locales → nacionales) y lograr una medalla en la IOI [34], ser uno de los miembros de los equipos que representen a tu universidad en el ICPC [66] (finales nacionales → regionales → mundiales) u obtener buenos resultados en otros concursos de programación, no hay duda de que este es tu libro.

En los próximos capítulos, aprenderás todo lo necesario desde el nivel básico al intermedio o,

incluso, avanzado³, sobre estructuras de datos y algoritmos que han aparecido en concursos de programación recientes, recopilados de diversas fuentes [47, 9, 57, 7, 40, 58, 42, 60, 1, 38, 8, 59, 41, 62, 48] (ver la Figura 1.4). No solo conocerás los conceptos que están detrás de las estructuras de datos y los algoritmos, sino que también descubrirás muchos consejos sobre programación, derivados de nuestra propia experiencia, que pueden resultar útiles en un concurso. Vamos a comenzar con algunos consejos generales.

1.2.1 Consejo 1: escribe más rápido

No es broma. Aunque este consejo puede no tener mucho sentido, dado que ni el ICPC ni (especialmente) la IOI son concursos de mecanografía, hemos visto equipos en el ICPC clasificados en las posiciones i e $i + 1$ separados por unos pocos minutos, así como concursantes de la IOI frustados al perder puntos importantes por no ser capaces de programar adecuadamente una solución de fuerza bruta en el último minuto. Cuando eres capaz de resolver el mismo número de problemas que tu contrincante, la diferencia vendrá dada por la capacidad de producir código preciso y robusto y por... la velocidad al escribirlo.

Puedes conocer tu velocidad con el teclado en <http://www.typingtest.com>, y seguir las instrucciones mostradas para mejorar tus capacidades en ese sentido. Steven logra ~85-95 palabras por minuto y Felix ~55-65. Si tu velocidad es muy inferior a estos números, deberías tomarte este consejo muy en serio.

Además de ser capaz de escribir caracteres alfanuméricos rápidamente y de forma correcta, deberás familiarizarte también con las posiciones de las teclas utilizadas más frecuentemente por los lenguajes de programación: paréntesis (), llaves {}, corchetes [] o los signos <>; el punto y coma ; o los dos puntos :, las comillas simples ' ' para los caracteres, las comillas dobles " " para las cadenas, el ampersand &, la barra vertical o ‘tubería’ |, el signo de exclamación !, etc.

A modo de práctica, teclea el siguiente código C++ lo más rápidamente posible.

```
1 #include <algorithm>           // si tienes problemas con este código C++,  
2 #include <cmath>             // deberías consultar tus libros sobre programación...  
3 #include <cstdio>  
4 #include <cstring>  
5 using namespace std;  
6 /* Forming Quiz Teams, la solución al problema UVA 10911 */  
7     // usar variables globales es una mala práctica en programación,  
8 int N, target;                // pero aceptable en el ámbito competitivo  
9 double dist[20][20], memo[1 << 16]; // 1 << 16 = 2^16, sabiendo que max N=8  
10  
11 double matching(int bitmask) {      // estado DP = máscara de bits  
12     // inicializamos 'memo' con -1 en la función main  
13     if (memo[bitmask] > -0.5)        // este estado ya se ha calculado  
14         return memo[bitmask];        // basta buscar en la tabla memo  
15     if (bitmask == target)          // todos los estudiantes están emparejados  
16         return memo[bitmask] = 0;       // el coste es 0
```

³La percepción del material presentado en este libro como intermedio o avanzado depende de los conocimientos previos del lector en lo relativo a programación, algorítmica y capacidad de resolución de problemas.

```

17
18     double ans = 2000000000.0;           // inicializar con un valor grande
19     int p1, p2;
20     for (p1 = 0; p1 < 2 * N; p1++)
21         if (!(bitmask & (1 << p1)))
22             break;                      // encontrar el primer bit apagado
23         for (p2 = p1 + 1; p2 < 2 * N; p2++)    // después intentar emparejar p1
24             if (!(bitmask & (1 << p2)))        // con otro bit p2 también apagado
25                 ans = min(ans,                // elegir el mínimo
26                               dist[p1][p2] + matching(bitmask | (1 << p1) | (1 << p2)));
27
28     return memo[bitmask] = ans; // guardar el resultado en una tabla y volver
29 }
30
31 int main() {
32     int i, j, caseNo = 1, x[20], y[20];
33     // freopen("10911.txt", "r", stdin);      // archivo de entrada a stdin
34
35     while (scanf("%d", &N), N) {            // sí, esto se puede hacer :)
36         for (i = 0; i < 2 * N; i++)
37             scanf("%*s %d %d", &x[i], &y[i]);      // '%*s' salta los nombres
38         for (i = 0; i < 2 * N - 1; i++)    // crear tabla de distancia de pares
39             for (j = i + 1; j < 2 * N; j++)        // ¡ya conocías 'hypot'?
40                 dist[i][j] = dist[j][i] = hypot(x[i] - x[j], y[i] - y[j]);
41
42         // utilizar DP para resolver el emparejamiento perfecto de peso mínimo
43         // en un grafo general pequeño
44         for (i = 0; i < (1 << 16); i++) memo[i] = -1.0; // fijar valores a -1
45         target = (1 << (2 * N)) - 1;
46         printf("Case %d: %.2lf\n", caseNo++, matching(0));
47     } } // return 0;

```

Para tu referencia, la explicación de esta solución de ‘programación dinámica con máscara de bits’ aparece en las secciones 2.2, 3.5 y 8.3.1. No te preocupes si todavía no la entiendes.

1.2.2 Consejo 2: identifica rápidamente el tipo de problema

En el ICPC, los concursantes (equipos) reciben un **conjunto** de $\approx 7\text{-}12$ problemas de diferentes tipos. De nuestra observación de esos conjuntos de problemas, en concursos regionales de Asia y en las finales mundiales recientes, podemos categorizar los tipos de problemas y su frecuencia de aparición en la siguiente tabla 1.1.

En la IOI, los concursantes reciben 6 tareas a lo largo de 2 días, que cubren los puntos 1-5 y 10, con un subconjunto *mucho más pequeño* de los puntos 6-10 de la tabla 1.1. Para más información, recomendamos consultar el temario de la IOI de 2009 [20] y la clasificación de problemas de las IOI 1989-2008 [67].

La clasificación de la tabla 1.1 está adaptada de [50] y no pretende ser exhaustiva. Algunas

Nº	Categoría	En este libro	Frecuencia
1.	<i>Ad hoc</i>	Sección 1.4	1-2
2.	Búsqueda completa (iterativa/recursiva)	Sección 3.2	1-2
3.	Divide y vencerás	Sección 3.3	0-1
4.	Voraz (normalmente los originales)	Sección 3.4	0-1
5.	Programación dinámica (normalmente los originales)	Sección 3.5	1-3
6.	Grafos	Capítulo 4	1-2
7.	Matemáticas	Capítulo 5	1-2
8.	Procesamiento de cadenas	Capítulo 6	1
9.	Geometría computacional	Capítulo 7	1
10.	Algunos problemas difíciles/poco habituales	Capítulos 8-9	1-2
Total en el conjunto			8-17 ($\approx \leq 12$)

Tabla 1.1: Tipos de problemas en los ICPC regionales de Asia recientes

técnicas, como la ordenación, no aparecen aquí, ya que se consideran triviales y, normalmente, serán utilizadas como subrutinas de un problema más amplio. Tampoco hemos incluído la recursividad, ya que está implícita en categorías como el *backtracking* recursivo o la programación dinámica. Omitimos las estructuras de datos, ya que el uso eficiente de las mismas se puede considerar como parte integral en la resolución de problemas más complejos. Por supuesto, en ocasiones, los problemas requieren técnicas mixtas: un problema puede estar clasificado en más de un tipo. Por ejemplo, el algoritmo de Floyd Warshall es tanto una solución para problemas de grafos, como el de los caminos más cortos entre todos los pares (APSP, sección 4.5), como un algoritmo de programación dinámica (sección 3.5). Los algoritmos de Prim y Kruskal son, igualmente, soluciones para problemas de árbol recubridor mínimo (MST, sección 4.3) y algoritmos voraces (sección 3.4). En la sección 8.4, veremos problemas (más difíciles) para cuya solución son necesarios más de un algoritmo y/o estructura de datos.

En un futuro (próximo) estas clasificaciones podrían cambiar. Un ejemplo significativo es la programación dinámica. Esta técnica, desconocida antes de la década de 1940, y poco utilizada en el ICPC o la IOI antes de mediados de la década de 1990, es considerada un requisito fundamental hoy día. Para ilustrarlo, sirva que en las finales mundiales del ICPC de 2010 hubo ≥ 3 problemas de DP (de un total de 11).

Sin embargo, el objetivo principal *no es* simplemente asociar problemas con las técnicas necesarias para resolverlos, como en la tabla 1.1. Una vez que conozcas la mayoría de los temas de este libro, deberías de ser capaz de catalogar los problemas en los tres tipos de la tabla 1.2.

Nº	Categoría	Confianza y expectativa de resolución
A.	Ya he resuelto este tipo	Estoy seguro de resolverlo otra vez (rápidamente)
B.	Ya he visto este tipo	Pero todavía no sé resolverlo
C.	No he visto este tipo	Seguir leyendo

Tabla 1.2: Tipos de problemas (versión compacta)

Para ser *competitivo*, es decir, obtener *un buen resultado* en un concurso de programación, deberás ser capaz de clasificar problemas frecuentemente y con confianza en el tipo A, y minimizar el número de problemas que quedan en el tipo B. En suma, debes adquirir los suficientes conocimientos de algorítmica y desarrollar tus capacidades de programación como para considerar sencillos la mayoría de los problemas clásicos. Sin embargo, para *ganar* un concurso de progra-

mación, deberás tener *capacidad de resolución de problemas* avanzada (por ejemplo, reduciendo un problema determinado a uno conocido, identificando pistas sutiles o propiedades especiales del problema, abordando el problema desde un punto de vista poco evidente, etc.) para que tú (o tu equipo) seas capaz de reconducir la solución requerida para un problema difícil/original del tipo C en la IOI o el ICPC, y hacerlo *durante* el concurso, no *después* de que el autor del problema, o el juez correspondiente, hagan pública la solución.

Ejercicio 1.2.1

UVa	Título	Tipo de problema	Más información
10360	Rat Attack	Búsqueda completa o DP	Sección 3.2
10341	Solve It		Sección 3.3
11292	Dragon of Loowater		Sección 3.4
11450	Wedding Shopping		Sección 3.5
10911	Forming Quiz Teams	DP con máscara de bits	Sección 8.3.1
11635	Hotel Booking		Sección 8.4
11506	Angry Programmer		Sección 4.6
10243	Fire! Fire!! Fire!!!		Sección 4.7.1
10717	Mint		Sección 8.4
11512	GATTACA		Sección 6.6
10065	Useless Tile Packers		Sección 7.3.7

Tabla 1.3: Ejercicio: clasificar estos problemas del Online Judge

Lee los problemas de UVa [49] de la tabla 1.3 y determina sus tipos. Dos de ellos ya aparecen identificados. Completar esta tabla será sencillo después de leer este libro, ya que todas las técnicas necesarias para resolver los problemas aparecen aquí descritas.

1.2.3 Consejo 3: analiza el algoritmo

Una vez que hayas diseñado un algoritmo que resuelva un problema en particular, en un concurso de programación, debes hacerte la siguiente pregunta: dado el límite superior máximo del tamaño de la entrada (indicado normalmente en el enunciado del problema, si está bien redactado), ¿es el algoritmo actual capaz de, con su complejidad de espacio/tiempo, resolver el problema en particular dentro de los límites de tiempo y memoria indicados?

En ocasiones, hay más de una forma de atacar un problema. Algunos de los abordajes pueden ser incorrectos, otros no lo suficientemente rápidos y otros ‘excesivos’. Una buena estrategia consiste en plantear los diferentes algoritmos y elegir la **solución más sencilla que funcione** (es decir, es lo bastante rápida para cumplir con los límites de tiempo y memoria y devuelve la respuesta correcta)⁴.

⁴Es cierto que, en los concursos de programación, elegir el algoritmo más sencillo que funciona es crucial para un buen resultado. Sin embargo, durante las *sesiones de entrenamiento*, donde no tenemos limitaciones de tiempo, puede ser beneficioso dedicar más esfuerzos a resolver un problema determinado utilizando *el mejor algoritmo posible*. De esta forma, estaremos mejor preparados y, si en el futuro encontramos una versión más compleja del problema, tendremos más posibilidades de obtener e implementar la solución correcta.

Los ordenadores modernos son muy rápidos y pueden procesar⁵ hasta $\approx 100M$ (o 10^8 ; $1M = 1000000$) operaciones por segundo. Esta información es útil para determinar si el algoritmo actual cumplirá con el límite de tiempo. Por ejemplo, si el tamaño máximo de entrada n es $100K$ (o 10^5 ; $1K = 1000$), y el algoritmo en cuestión tiene una complejidad de $O(n^2)$, el sentido común (o una calculadora) indicará que $(100K)^2$ o 10^{10} , es un número muy grande para el que el algoritmo necesitará (del orden de) cientos de segundos de ejecución. Por lo tanto, será necesario plantear un algoritmo más rápido (y también correcto) que resuelva el problema. Vamos a suponer que encontramos uno cuya complejidad sea de $O(n \log_2 n)$. En este caso, la calculadora nos informará de que $10^5 \log_2 10^5$ es solo $1,7 \times 10^6$ y el sentido común nos dictará que el algoritmo (que ahora se ejecutará en menos de un segundo) probablemente estará dentro del límite de tiempo.

Los límites del problema son tan importantes como la complejidad del algoritmo, para determinar si la solución es adecuada. Supongamos que solo podemos plantear un algoritmo relativamente simple de implementar, que implica una complejidad de $O(n^4)$. De entrada, puede parecer una solución inaceptable, pero si $n \leq 50$, habremos resuelto el problema. Podemos implementar el algoritmo de $O(n^4)$ impunemente, ya que 50^4 resulta ser $6,25M$, y eso únicamente ocupará alrededor de un segundo de tiempo de ejecución.

Hay que tener en cuenta, sin embargo, que el orden de complejidad no indica, necesariamente, el número real de operaciones que requerirá el algoritmo. Si cada iteración implica un número importante de operaciones (ya sean de coma flotante o con un número significativo de bucles anidados constantes), o si la implementación tiene un consumo ‘constante’ alto en su ejecución (bucles repetidos innecesariamente, pasadas múltiples o, incluso, sobrecargas en E/S o ejecución), el código puede tardar en ejecutarse más de lo esperado. De todas formas, esto no suele ser un problema grave, ya que los autores de los problemas deberían haber diseñado los límites de tiempo de forma que un algoritmo bien programado, con una complejidad adecuada, logre un veredicto de AC.

Al analizar la complejidad del algoritmo con los límites de entrada, tiempo y memoria dados, se puede decidir con más facilidad si se debería intentar implementar directamente (lo que ocupa un tiempo precioso en los concursos presenciales), intentar mejorarlo primero o, en último caso, ocuparse de un problema diferente.

Como se ha mencionado en el prefacio de este libro, vamos a intentar *no* tratar el concepto del análisis de algoritmos en detalle. *Asumimos* que el lector ya cuenta con esas habilidades. Hay muchos libros de referencia (por ejemplo, “Introduction to Algorithms” [7], “Algorithm Design” [38], “Algorithms” [8], etc.) que ayudarán a entender los siguientes conceptos y técnicas necesarios para el análisis de algoritmos:

- Análisis de complejidad de tiempo y espacio básico, para algoritmos iterativos y recursivos:
 - Un algoritmo con k bucles anidados de unas n iteraciones cada uno, tiene una complejidad de $O(n^k)$.
 - Si el algoritmo es recursivo, con b llamadas recursivas por nivel, y tiene L niveles, tendrá una complejidad aproximada de $O(b^L)$, pero este límite superior es muy variable. La complejidad real vendrá dada por las acciones que se realicen en cada nivel y dependiendo de si es posible realizar poda.

⁵Hay que tomarse esto como una aproximación, ya que el número puede variar entre diferentes equipos.

- Un algoritmo de programación dinámica, u otra rutina iterativa, que procese una matriz bidimensional $n \times n$ en $O(k)$ por celda, tiene un tiempo de ejecución de $O(k \times n^2)$. Este aspecto está mejor explicado en la sección 3.5.

■ Técnicas de análisis más avanzadas:

- Probar la validez de un algoritmo (especialmente los algoritmos voraces de la sección 3.4), para minimizar las posibilidades de obtener un veredicto de respuesta incorrecta.
- Realizar un análisis amortizado (ver, por ejemplo, el capítulo 17 de [7]), algo que raramente se hace en los concursos, para minimizar el riesgo de obtener un veredicto de tiempo límite superado o, peor aún, considerar el algoritmo demasiado lento e ignorar el problema cuando, en realidad, es lo suficientemente rápido una vez considerada la amortización.
- Realizar un análisis cuidadoso de la salida requerida, para analizar el algoritmo que (también) depende del tamaño de la salida y minimizar las posibilidades de obtener un veredicto de tiempo límite superado. Por ejemplo, un algoritmo que busque una cadena de longitud m dentro de una cadena más larga, con la ayuda de un árbol de sufijos (ya construido), se ejecuta en un tiempo de $O(m + occ)$. El tiempo que consume dicho algoritmo no depende únicamente del tamaño m de la entrada, sino también del de la salida, es decir, del número de apariciones de occ (se pueden ver más detalles en la sección 6.6).

■ Familiaridad con los siguientes límites:

- $2^{10} = 1024 \approx 10^3$, $2^{20} = 1048576 \approx 10^6$.
- Los enteros con signo de 32 (`int`) y 64 (`long long`) bits tienen unos límites superiores de $2^{31}-1 \approx 2 \times 10^9$ (se pueden utilizar con seguridad hasta con ≈ 9 dígitos decimales) y $2^{63}-1 \approx 9 \times 10^{18}$ (hasta ≈ 18 dígitos decimales), respectivamente.
- Los enteros sin signo se utilizarán, únicamente, cuando no se necesiten números negativos. Los enteros sin signo de 32 (`unsigned int`) y 64 (`unsigned long long`) bits tienen unos límites superiores de $2^{32}-1 \approx 4 \times 10^9$ y $2^{64}-1 \approx 1,8 \times 10^{19}$, respectivamente.
- Para almacenar enteros $\geq 2^{64}$, se utilizará la técnica de enteros grandes (sección 5.3).
- Existen $n!$ permutaciones y 2^n subconjuntos (o combinaciones) para n elementos.
- La complejidad de tiempo óptima para un algoritmo de ordenación, basado en la comparación, es $\Omega(n \log_2 n)$.
- El tamaño de entrada más grande en un concurso de programación típico debe ser $< 1M$. En tamaños superiores, el tiempo necesario para leer la entrada (la rutina de entrada/salida) será el cuello de botella.
- Normalmente, los algoritmos de complejidad $O(n \log_2 n)$ bastan para resolver la mayoría de los problemas de los concursos, por una sencilla razón: los algoritmos $O(n \log_2 n)$ y los, en teoría mejores $O(n)$, son difíciles de diferenciar *empíricamente* en el entorno de un concurso de programación, con tiempos límite estrictos y tamaños de entrada $n < 1M$.
- Una CPU normal del año 2013 puede procesar $100M = 10^8$ operaciones en unos pocos segundos.

Muchos programadores novatos ignoran esta fase e, inmediatamente, implementan el primer algoritmo que se les ocurre, para descubrir, a continuación, que la estructura de datos elegida y/o el propio algoritmo no son lo suficientemente eficientes (o están equivocados). Nuestro consejo para los concursantes del ICPC⁶: hay que evitar empezar a programar hasta que tengamos la seguridad de que el algoritmo es correcto y suficientemente rápido.

n	Peor algoritmo AC	Comentario
$\leq [10..11]$	$O(n!)$, $O(n^6)$	p.e. permutaciones enumeradas (sección 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	p.e. TSP con DP (sección 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	p.e. DP con máscara de bits (sección 8.3.1)
≤ 100	$O(n^4)$	p.e. DP con 3 dimensiones + bucle $O(n)$, $n C_{k=4}$
≤ 400	$O(n^3)$	p.e. Floyd Warshall (sección 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	p.e. 2 bucles anidados + DS con árbol (sección 2.3)
$\leq 10K$	$O(n^2)$	p.e. ordenación burbuja/selección/ inserción (sección 2.2)
$\leq 1M$	$O(n \log_2 n)$	p.e. ordenación por mezcla, árbol de segmentos (sección 2.3)
$\leq 100M$	$O(n)$, $O(\log_2 n)$, $O(1)$	mayoría de problemas con $n \leq 1M$ (cuello de botella en E/S)

Tabla 1.4: Referencia rápida de la complejidad de tiempo del ‘peor algoritmo AC’ para tamaños de entrada n en casos de prueba únicos, asumiendo que tu CPU pueda calcular $100M$ elementos en 3 segundos.

Para ayudarte a entender el crecimiento de varias complejidades de tiempo comunes y, con ello, ayudarte a determinar qué velocidad es ‘suficiente’, consulta la tabla 1.4. En muchos otros libros sobre estructuras de datos y algoritmos, se pueden encontrar variantes de esta tabla. En este caso, la hemos escrito desde la *perspectiva de un concurso de programación*. Las limitaciones del tamaño de la entrada se proporcionan, normalmente, en el enunciado del problema (si está bien escrito). Asumiendo que una CPU típica pueda ejecutar 100 millones de operaciones en unos tres segundos (el límite de tiempo habitual en la mayoría de los problemas UVa [49]), podemos predecir el ‘peor’ algoritmo que podría ejecutarse dentro de los límites. Normalmente el algoritmo más sencillo es el que tiene una complejidad de tiempo más deficiente, pero si nos sirve para resolver el problema, debemos utilizarlo.

En la tabla 1.4, podemos observar la importancia de utilizar buenos algoritmos con órdenes de crecimiento pequeños, ya que eso nos permite resolver problemas con tamaños de entrada mayores. Pero los algoritmos más rápidos no son, en general, sencillos y, en ocasiones, resultan muy difíciles de implementar. En la sección 3.2.3, veremos algunas pistas que pueden ayudar a utilizar un mismo tipo de algoritmo con tamaños de entrada mayores. En los siguientes capítulos, también explicaremos algoritmos eficientes aplicables a varios problemas de computación.

⁶A diferencia del ICPC, las tareas de la IOI se puede resolver (de forma parcial o total), normalmente, utilizando diferentes soluciones, cada una de ellas con complejidades de tiempo y puntuaciones de subtareas distintas. Para obtener más puntos, puede ser una buena idea utilizar inicialmente un algoritmo de fuerza bruta y, de esa forma, entender mejor el problema. No habrá una penalización significativa en el tiempo, ya que la IOI no es un campeonato de velocidad. Después, se puede ir mejorando la solución iterativamente, para incrementar la puntuación.

Ejercicio 1.2.2

Responde a las siguientes preguntas, utilizando tu conocimiento actual sobre algoritmos clásicos y sus complejidades de tiempo. Una vez hayas terminado de leer el libro, podría ser interesante repetir este ejercicio.

1. Tenemos n páginas web ($1 \leq n \leq 10M$). Cada página i tiene una clasificación r_i . Queremos elegir las 10 páginas con mejor clasificación. ¿Qué método es mejor?
 - a) Cargar la clasificación de las n páginas en memoria, ordenarlas (sección 2.2) de forma descendente y seleccionar las 10 primeras.
 - b) Utilizar una estructura de datos de cola de prioridad (un montículo) (sección 2.3, **ejercicio 2.3.10***).
2. Dada una lista L con $10K$ enteros, tienes que consultar *frecuentemente* $\text{sum}(i, j)$, es decir, la suma de $L[i] + L[i+1] + \dots + L[j]$. ¿Qué estructura de datos usarías?
 - a) *Array* sencillo (sección 2.2).
 - b) *Array* sencillo procesado previamente con DP (secciones 2.2 y 3.5).
 - c) Árbol de búsqueda binaria equilibrado (sección 2.3).
 - d) Montículo binario (sección 2.3).
 - e) Árbol de segmentos (sección 2.4.3).
 - f) Árbol binario indexado (Fenwick) (sección 2.4.4).
 - g) Árbol de sufijos (sección 6.6.2) o, quizás, *array* de sufijos (sección 6.6.4).
3. Dada una matriz de enteros Q , de tamaño $M \times N$ ($1 \leq M, N \leq 30$), determinar si existe una submatriz de Q de tamaño $A \times B$ ($1 \leq A \leq M, 1 \leq B \leq N$), donde $\text{media}(Q) = 7$. ¿Qué algoritmo no superará $1M$ de operaciones por caso de prueba en la peor de las situaciones?
 - a) Probar todas las submatrices posibles y comprobar si la media de cada una de ellas es 7.
 - b) Probar todas las submatrices posibles, pero en $O(M^2 \times N^2)$ usando la técnica:
_____.
4. Dado un conjunto S de N puntos distribuidos en un plano bidimensional ($2 \leq N \leq 1000$), encontrar dos puntos $\in S$ que tengan la mayor distancia euclídea de separación. ¿Es viable un algoritmo de búsqueda completa en $O(N^2)$ que compruebe todos los pares posibles?
 - a) Sí, esa búsqueda completa es posible.
 - b) No, hay que encontrar otro método. Utilicemos: _____.
5. Debes calcular el camino más corto entre dos vértices de un grafo acíclico dirigido ponderado (DAG) con $|V|, |E| \leq 100K$. ¿Qué algoritmos se pueden utilizar en un concurso de programación típico (esto es, con un límite de tiempo de aproximadamente 3 segundos)?

- a) Programación dinámica (secciones 3.5, 4.2.5 y 4.7.1).
- b) Búsqueda en anchura (secciones 4.2.2 y 4.4.2).
- c) Dijkstra (sección 4.4.3).
- d) Bellman Ford (sección 4.4.4).
- e) Floyd Warshall (sección 4.5).
6. ¿Qué algoritmo genera una lista de los $10K$ primeros números primos con la mejor complejidad de tiempo? (Sección 5.5.1)
- a) Criba de Eratóstenes (sección 5.5.1).
- b) Para cada número $i \in [1..10K]$, comprobar si `isPrime(i)` es verdadero (sección 5.5.1).
7. Quieres comprobar si el factorial de n , es decir, $n!$, es divisible por un entero m . $1 \leq n \leq 10000$. ¿Qué debes hacer?
- a) Comprobar si $n! \% m == 0$.
- b) Esta técnica ingenua no sirve, utilizar: _____ (sección 5.5.1).
8. Vuelve a la pregunta 4, pero ahora con un conjunto de puntos mayor: $N \leq 1M$, y una restricción añadida: los puntos están *distribuidos aleatoriamente* en un plano bidimensional.
- a) Se puede seguir usando la búsqueda completa mencionada en la pregunta 4.
- b) La técnica ingenua del punto anterior no funcionará, hay que utilizar _____ (sección 7.3.7).
9. Quieres enumerar todas las apariciones de una subcadena P (de longitud m) en una cadena (larga) T (de longitud n), si las hay. Tanto n como m tienen un máximo de $1M$ caracteres. ¿Qué algoritmo es más rápido?
- a) Utilizar el siguiente código C++:
- ```
1 for (int i = 0; i < n; i++) {
2 bool found = true;
3 for (int j = 0; j < m && found; j++)
4 if (i + j >= n || P[j] != T[i + j]) found = false;
5 if (found) printf("P is found at index %d in T\n", i);
6 }
```
- b) La técnica ingenua no sirve, utilizar: \_\_\_\_\_ (secciones 6.4 o 6.6).

## 1.2.4 Consejo 4: domina los lenguajes de programación

En el ICPC se permite utilizar varios lenguajes de programación<sup>7</sup>, incluyendo C/C++ y Java.

¿Qué lenguajes de programación se deben intentar dominar? Nuestra experiencia nos da esta respuesta: preferimos C++ acompañado de la Standard Template Library (STL) incluida, pero, además, es necesario tener conocimientos avanzados de Java. Aunque resulta más lento, Java incluye algunas bibliotecas y APIs muy potentes, como BigInteger/BigDecimal, GregorianCalendar, Regex, etc. Los programas en Java son más sencillos de depurar, con la capacidad de la máquina virtual de proporcionar un trazado de la pila cuando se produce un error de ejecución (a diferencia de los volcados o errores de segmentación de C/C++). Por otro lado, C/C++ también cuenta con méritos propios. Dependiendo del problema que nos ocupe, la mejor opción puede ser uno u otro lenguaje a la hora de implementar una solución en el menor tiempo posible.

Vamos a suponer que un problema requiere el cálculo de  $25!$  (el factorial de 25). La respuesta es muy grande: 15.511.210.043.330.985.984.000.000. Este número supera por mucho el tamaño del entero más grande de los tipos de datos incluídos en el lenguaje (`unsigned long long`:  $2^{64} - 1$ ). Como C/C++ todavía no incluye una biblioteca aritmética de precisión arbitraria, deberemos crear una nosotros mismos. El código en Java, sin embargo, es mucho más sencillo (hay más detalles en la sección 5.3). En este caso, el uso de Java acortará significativamente el tiempo empleado en la programación.

```
1 import java.util.Scanner;
2 import java.math.BigInteger;
3
4 class Main { // nombre estándar de clase Java en el UVa OJ
5 public static void main(String[] args) {
6 BigInteger fac = BigInteger.ONE;
7 for (int i = 2; i <= 25; i++)
8 fac = fac.multiply(BigInteger.valueOf(i)); // está en la biblioteca
9 System.out.println(fac);
10 } }
```

También es importante dominar y entender toda la capacidad del lenguaje de programación de preferencia. Veamos este problema con un formato de entrada no estándar: la primera línea de la entrada es un entero  $N$ . Le siguen  $N$  líneas, cada una de las cuales comienza con el carácter ‘0’, seguido de un punto ‘.’, y ello seguido de un número desconocido (de hasta 100 dígitos), finalmente terminado con tres puntos ‘...’. La tarea consiste en extraer los dígitos.

```
3
0.1227...
0.517611738...
0.7341231223444344389923899277...
```

<sup>7</sup>Opinión personal: según las últimas reglas de la IOI de 2013, Java todavía no está permitido en dicha competición. Los lenguajes de programación admitidos en la IOI con C, C++ y Pascal. Por otro lado, la final mundial del ICPC (y, por tanto, la mayoría de las regionales) permite el uso de C, C++ y Java. Por ello, parece claro que el ‘mejor’ lenguaje a dominar es C++, ya que está permitido en ambas competiciones y cuenta con las importantes aportaciones de la STL. Si los concursantes de la IOI deciden utilizar C++, tendrán la ventaja de poder utilizar el mismo lenguaje (aunque a un nivel superior) en el ámbito del ICPC.

Una posible solución sería:

```
1 #include <iostream>
2 using namespace std;
3
4 int N; // es buena estrategia utilizar variables globales en los concursos
5 char x[110]; // conviene dimensionar el array por encima de lo necesario
6
7 int main() {
8 scanf("%d\n", &N);
9 while (N--) { // hacemos un bucle desde N, N-1, N-2, ..., 0
10 scanf("0.%[0-9]...\\n", &x); // '&' es opcional cuando x es un array char
11 // nota: si te sorprende el truco anterior,
12 // consulta la sintaxis de scanf en www.cppreference.com
13 printf("the digits are 0.%s\\n", x);
14 } // return 0;
```



ch1\_01\_factorial.java



ch1\_02\_scnf.cpp

Muchos programadores de C/C++ no son conscientes de la potencia que ofrece el uso de expresiones regulares incluído en la biblioteca de entrada/salida estándar de C. Aunque `scanf/printf` son rutinas de E/S propias de C, se pueden utilizar en el código escrito en C++. Muchos programadores de C++ se *obligan* a sí mismos a utilizar `cin/cout` constantemente, a pesar de que no son tan flexibles como `scanf/printf`, y mucho más lentas.

En los concursos de programación, sobre todo en los ICPC, el tiempo empleado en la programación *no* debería ser un cuello de botella. Una vez que tengas claro cuál es el ‘peor algoritmo AC’ que se ejecutará dentro del tiempo límite estipulado, lo normal es que lo traslades rápidamente a un código libre de errores.

Ahora, intenta resolver alguno de los siguientes ejercicios. Si necesitas más de 10 líneas de código para cualquiera de ellos, deberías replantearte tu conocimiento de lenguajes de programación. Dominar los lenguajes que utilizas y los recursos que ofrecen es extraordinariamente importante y te ayudará en gran medida en los concursos de programación.

### Ejercicio 1.2.3

Escribe código funcional, *lo más conciso posible*, para realizar las siguientes tareas:

1. Usando **Java**, lee un `double` (por ejemplo, 1.4732, 15.324547327, etc.) y escríbelo, pero con una anchura de campo mínima de 7 caracteres y con 3 dígitos después del punto decimal (por ejemplo, `ss1.473`, donde ‘s’ indica un espacio, `s15.325`, etc.).
2. Dado un entero  $n$  ( $n \leq 15$ ), escribir  $\pi$  con  $n$  dígitos después del punto decimal (con redondeo). Por ejemplo, para  $n = 2$ , escribir 3.14; para  $n = 4$ , 3.1416; o, para  $n = 5$ , 3.14159.
3. Dada una fecha, determinar el día de la semana al que corresponde (lunes, ...).

domingo). Por ejemplo, el 9 de agosto de 2010, día de la publicación de la primera edición en inglés de este libro, fue lunes.

4. Dados  $n$  enteros aleatorios, escribir ordenados los enteros diferentes (únicos).
5. Dadas las fechas de nacimiento, distintas y válidas, de  $n$  personas, en grupos de tres datos (DD, MM, AAAA), ordenarlas primero por meses ascendentes (MM), después por días ascendentes (DD) y, finalmente, por edad ascendente.
6. Dada una lista de enteros *ordenados*  $L$ , de tamaño hasta  $1M$  elementos, determinar si un valor  $v$  existe en  $L$  sin utilizar más de 20 comparaciones (más detalles en la sección 2.2).
7. Generar todas las permutaciones posibles de {‘A’, ‘B’, ‘C’, ..., ‘J’}, las primeras  $N = 10$  letras del alfabeto (ver la sección 3.2.1).
8. Generar todos los subconjuntos posibles de {0, 1, 2, ...,  $N-1$ }, para  $N = 20$  (ver la sección 3.2.1).
9. Dada una cadena que representa un número en base  $X$ , convertirlo a una cadena equivalente en base  $Y$ ,  $2 \leq X, Y \leq 36$ . Por ejemplo, “FF” en base  $X = 16$  (hexadecimal) es “255” en base  $Y_1 = 10$  (decimal), y “11111111” en base  $Y_2 = 2$  (binario). Ver la sección 5.3.2.
10. Definamos una ‘palabra especial’ como letras minúsculas seguidas de dos dígitos. Dada una cadena, sustituir todas las ‘palabras especiales’ de longitud 3 con 3 asteriscos “\*\*\*”. Por ejemplo, S = “línea: a70 y z72 serán sustituidas, aa24 y a872 no” se transformará en S = “línea: \*\*\* y \*\*\* serán sustituidas, aa24 y a872 no”.
11. Dada una expresión matemática *válida* que utilice ‘+’, ‘-’, ‘\*’, ‘/’, ‘(’ y ‘)’, en una sola línea, evaluar dicha expresión. Por ejemplo, la expresión bastante complicada, pero válida,  $3 + (8 - 7.5) * 10 / 5 - (2 + 5 * 7)$ , debe dar como resultado  $-33.0$ , cuando se procesa según las reglas de precedencia estándar.

### 1.2.5 Consejo 5: domina el arte de probar el código

Pensabas que tenías resuelto un problema en particular. Has identificado el tipo de problema, diseñado el algoritmo que lo resuelve, verificado que el algoritmo (con sus estructuras de datos) se ejecutará dentro de los límites de tiempo (y memoria), teniendo en cuenta la complejidad temporal (y espacial), y has implementado el algoritmo pero, aun así, la solución no es aceptada.

Dependiendo del concurso de programación, es posible que recibas algún tipo de recompensa por la solución parcial de un problema. En el ICPC, sólo obtendrás los puntos de un problema si el código propuesto resuelve **todos** los casos de prueba secretos. Otros veredictos, como ‘error de presentación’ (PE), ‘respuesta incorrecta’ (WA), ‘límite de tiempo superado’ (TLE), ‘límite de memoria superado’ (MLE), ‘error en tiempo de ejecución’ (RTE), etc, no subirán al marcador. En el sistema actual de la IOI (2010-2013), se utiliza un sistema de puntuación por subtareas. Los casos de prueba se agrupan en diferentes tareas, normalmente variantes más sencillas de la tarea original, con límites de entrada más pequeños, o con aproximaciones más simplistas.

Solo se acreditará como resuelta una subtarea si el código resuelve todos sus casos de prueba. Recibirás *fichas*, que podrás utilizar (escasamente) durante el concurso, para ver la evaluación que el juez ha hecho de tu código.

En cualquier caso, necesitarás diseñar casos de prueba buenos, completos y complicados. El caso de ejemplo que se proporciona en el enunciado del problema suele ser trivial y, por ello, no es un buen método para determinar si tu código es correcto.

En vez de malgastar envíos (acumulando penalizaciones en tiempo o puntuación) en el ICPC o fichas en la IOI, quizás te resulte más conveniente diseñar casos de prueba que lleven tu código al límite en tu propio ordenador<sup>8</sup>. Asegúrate de que tu código es capaz de resolverlos (en caso contrario no tiene ningún sentido enviar la solución, ya que previsiblemente será incorrecta, salvo que quieras tantear los límites de los casos de prueba).

Algunos entrenadores animan a sus estudiantes a competir entre ellos diseñando casos de prueba. Si los casos del estudiante A son capaces de vencer al código del estudiante B, el estudiante A será recompensado. Quizás quieras probar este método cuando entrenes con tu equipo.

A continuación, incluimos algunas líneas generales de cómo diseñar buenos casos de prueba, según nuestra experiencia. Son los mismos pasos que suelen seguir los autores de los problemas:

1. Tus casos de prueba deben incluir los casos de ejemplo, ya que tenemos la garantía de que esos son correctos. Utiliza ‘fc’ en Windows, o ‘diff’ en UNIX, para comparar la salida de tu código con la salida de ejemplo. No conviene hacer comparaciones manuales, ya que los humanos somos propensos a cometer errores en este tipo de tareas, sobre todo en aquellos problemas donde el formato de salida es estricto (por ejemplo, incluir una línea en blanco *entre* los casos de prueba, en vez de *después* de cada caso de prueba). Para hacerlo, *copia y pega* los ejemplos de entrada y salida del enunciado del problema, guárdalos en archivos (llamados ‘entrada’ y ‘salida’, o cualquier otra cosa suficientemente autoexplicativa). Despues, tras compilar tu programa (vamos a asumir que el nombre del ejecutable es el predeterminado de ‘g++’, es decir, ‘a.out’), ejecútalo con un direccionamiento de E/S: ‘./a.out < entrada > misalida’. Por último, ejecuta ‘diff misalida salida’, para detectar cualquier diferencia, si es que existe.
2. En los problemas con casos de prueba múltiples en la misma ejecución (ver la sección 1.3.2), deberías incluir dos casos idénticos y consecutivos en la misma ejecución. Ambos deben producir la misma respuesta correcta, previamente conocida. Así, podrás determinar si has olvidado inicializar alguna variable. Si la primera aparición tiene una respuesta correcta, pero la segunda no, es muy posible que no hayas reiniciado correctamente las variables.
3. Los casos de prueba deben contener casos límite complicados. Ponte en la piel del autor del problema y trata de identificar la peor entrada posible para el algoritmo, identificando aquellos casos que están ‘ocultos’ o implícitos en la descripción del problema. Estos casos suelen estar incluidos en las pruebas del juez, pero *no* aparecen en los ejemplos de entrada y salida. Los casos límite, normalmente, aparecen en los valores extremos, como  $N = 0$ ,  $N = 1$ , valores negativos, números finales (y/o intermedios) grandes, que no caben en un entero con signo de 32 bits, etc.

---

<sup>8</sup>Los entornos de trabajo de los concursos de programación son diferentes de uno a otro. Esto puede ser un problema para los concursantes que dependen en exceso de entornos gráficos de desarrollo (IDEs) (por ejemplo Visual Studio, Eclipse, etc.) para depurar su código. No es mala idea tratar de programar utilizando solo un **editor de textos** y un **compilador**.

4. Tus casos de prueba deberían incluir casos *grandes*. Aumenta el tamaño de la entrada de forma incremental, hasta llegar a los límites máximos expresados en la descripción del problema. Utiliza casos de prueba grandes con estructuras triviales, que sean fáciles de verificar manualmente, y casos de prueba grandes y *aleatorios* para comprobar si tu código es capaz de terminar en el tiempo previsto y proporciona una salida razonable (ya que, en este caso, será difícil verificar que es correcta). En ocasiones, el programa podrá funcionar con casos pequeños, pero devolverá respuestas incorrectas, fallará o superará el límite de tiempo, al aumentar el tamaño de la entrada. Si ocurriese esto, comprueba desbordamientos y límites, o mejora tu algoritmo.
5. Aunque esto no es habitual en los concursos de programación modernos, no asumas que el formato de la entrada será perfecto si la descripción del problema no lo indica explícitamente (sobre todo en los problemas mal escritos). Prueba a insertar espacios en blanco adicionales (espacios o tabuladores) en la entrada, y verifica si tu código sigue funcionando sin fallos en ese caso.

Sin embargo, después de seguir todos los pasos, puede que sigas recibiendo veredictos de no aceptado. En el ICPC, es válido considerar el veredicto del juez y la clasificación (normalmente disponible durante las primeras cuatro horas) como elementos para determinar el siguiente paso. En las IOI 2010-2013, los concursantes tenían un número limitado de fichas para comprobar si sus códigos eran correctos. A medida que ganas experiencia, serás capaz de realizar mejores valoraciones y elecciones.

### Ejercicio 1.2.4

Conciencia de la situación (aplicable sobre todo al ICPC, no es relevante en la IOI).

1. Obtienes un veredicto de WA en un problema muy sencillo. ¿Qué deberías hacer?
  - a) Abandonar este problema y seguir con otro.
  - b) Mejorar el rendimiento de tu solución (optimizando el código/algoritmo).
  - c) Crear casos de prueba complejos para tratar de encontrar el error.
  - d) Si la competición es por equipos, pedir a un compañero que trate de resolverlo.
2. Recibes un veredicto TLE para tu solución  $O(N^3)$ . Sin embargo, el máximo de  $N$  es 100. ¿Qué deberías hacer?
  - a) Abandonar este problema y seguir con otro.
  - b) Mejorar el rendimiento de tu solución (mejor código/algoritmo).
  - c) Crear casos de prueba complejos para tratar de encontrar el error.
3. Ampliación a la pregunta 2: ¿qué ocurre si el máximo de  $N$  es 100.000?
4. Otra ampliación a la pregunta 2: ¿qué ocurre si el máximo de  $N$  es 1.000, la salida solo depende del tamaño de la entrada  $N$  y todavía quedan *cuatro horas* de concurso?
5. Recibes un veredicto de RTE. El código, aparentemente, se ejecuta perfectamente en tu sistema local. ¿Qué deberías hacer?

6. Treinta minutos después de comenzar el concurso, le echas un vistazo al marcador. Hay *muchos* equipos que han resuelto el problema *X*, que tu equipo todavía no ha intentado. ¿Qué deberías hacer?
7. A mitad del concurso le echas un vistazo al marcador. El equipo líder (asumiendo que no sea el tuyo) acaba de resolver el problema *Y*. ¿Qué deberías hacer?
8. Tu equipo le ha dedicado dos horas a un problema complicado. Ya habéis enviado varias implementaciones realizadas por diferentes miembros del equipo. Todos los envíos han resultado incorrectos. No tenéis ni idea de dónde está el fallo. ¿Qué deberías hacer?
9. Queda una hora para que termine el concurso. Tienes un código con veredicto WA y una idea nueva para *otro* problema. ¿Qué deberías hacer?
  - a) Abandonar el problema con el código WA e intentar el otro problema para tratar de resolver uno más.
  - b) Insistir en depurar el código WA. No queda tiempo para comenzar a trabajar en un nuevo problema.
  - c) En el ICPC, imprimir el código WA y pedir a los otros dos miembros del equipo que lo analicen, mientras tú comienzas a trabajar en el problema nuevo, en un intento de resolver *dos* problemas más.

### 1.2.6 Consejo 6: práctica y más práctica

Los programadores competitivos, al igual que los auténticos atletas, deben entrenar con regularidad para mantenerse ‘en forma’. Para este penúltimo consejo, proporcionaremos una lista de varias páginas web, con recursos que te ayudarán a mejorar tu capacidad de resolución de problemas. Creemos firmemente que el éxito viene de un esfuerzo de mejora continuo.

El Online Judge de la Universidad de Valladolid (UVa, en España) [49] contiene problemas de ediciones pasadas del ICPC (locales, regionales y de las finales mundiales), además de problemas de otras fuentes, incluyendo varios provenientes de concursos alojados en el juez UVa. Puedes resolver los problemas y enviar las soluciones al juez. El sistema evaluará tu programa, lo antes posible, y te entregará un veredicto. Trata de resolver los problemas mencionados en este libro y, quizás, algún día veas tu nombre en la lista de los 500 usuarios que más problemas han resuelto.

A fecha de 24 de mayo de 2013, hay que resolver  $\geq 542$  problemas para estar entre los 500 mejores usuarios. Steven está clasificado en el puesto 27 (habiendo resuelto 1674 problemas), mientras que Felix está en el 37 (con 1487 problemas), de un total de  $\approx 149008$  usuarios en el juez UVa (y un total de  $\approx 4097$  problemas).

La página ‘hermana’ del juez en línea de la UVa, es el ICPC Live Archive [33], que contiene *casi todos* los problemas de las regionales y finales mundiales recientes del ICPC, desde el año 2000. Puedes entrenar aquí, si quieres tener un buen rendimiento en ICPC futuros. Desde octubre de 2011, unos cientos de problemas del Live Archive, incluyendo los que aparecen en la segunda edición de este libro, se encuentran también en el UVa Online Judge.



Figura 1.2: I: Online Judge; D: ICPC Live Archive.

La USA Computing Olympiad tiene una página web de entrenamiento muy potente [50], con concursos en línea que te ayudarán a mejorar tus habilidades de programación y resolución de problemas. Está más orientada a concursos de la IOI que del ICPC. Puedes ir directamente a su página web y empezar a entrenar.

Otro juez en línea es el Sphere Online Judge [61], donde los usuarios debidamente cualificados pueden añadir problemas propios. Es muy popular en países como Polonia, Brasil y Vietnam. Nosotros utilizamos el SPOJ para publicar algunos de los problemas que hemos escrito.



Figura 1.3: I: Plataforma de entrenamiento de USACO; D: Sphere Online Judge

TopCoder organiza frecuentemente ‘partidas de ronda única’ (SRM) [32], que consisten en un conjunto de unos pocos problemas que hay que resolver en 1-2 horas. Después del concurso, se te ofrece la posibilidad de ‘retar’ al código de otros participantes, mediante casos de prueba límite. Este juez en línea utiliza un sistema de clasificación (programadores rojos, amarillos, azules, etc.) para recompensar, con un mejor nivel, a los concursantes con una capacidad alta de resolución de problemas, frente a aquellos más prácticos, que resuelven un número mayor de problemas fáciles.

### 1.2.7 Consejo 7: trabaja en equipo (para ICPC)

Este último consejo es más fácil de plantear que de llevar a la práctica, pero aquí proponemos algunas ideas que podrían mejorar el rendimiento de tu equipo:

- Practicar la programación sobre el papel (es muy útil cuando un compañero de equipo está utilizando el ordenador para que, al llegar tu turno, puedas introducir el código lo más rápidamente posible, en vez de perder tiempo pensando delante de la pantalla).
- La estrategia de ‘enviar e imprimir’: si el código recibe un veredicto AC, basta con ignorar la copia impresa pero, si todavía no has llegado a ese punto, puedes depurar el código utilizando la versión en papel, mientras otro miembro del equipo está en el ordenador ocupado con otro problema. Cuidado, porque depurar código sin ayuda del ordenador es una tarea sumamente complicada.
- Si un compañero de equipo está programando (y tú no estás ocupado con otro problema), aprovecha el tiempo en preparar casos de prueba límite (que ojalá el programa de tu compañero pueda resolver).

- El factor X: mantén una buena relación de amistad con tus compañeros de equipo *fuera* de las sesiones de entrenamiento y los concursos.

## 1.3 Primeros pasos: los problemas fáciles

Nota: si tienes experiencia como participante en concursos de programación, puedes ignorar esta sección, que está dirigida a los recién llegados al mundo de la programación competitiva.

### 1.3.1 Anatomía de un problema

Un problema de un concurso de programación contiene *normalmente* los siguientes elementos:

- **Enunciado del problema.** Los problemas más sencillos se suelen escribir de forma que *engaños* a los concursantes, aparentando ser difíciles, mediante la inclusión, por ejemplo, de ‘información extra’ para crear una distracción. Los concursantes deberían ser capaces de *filtrar* los detalles sin importancia y centrarse en los esenciales. Por ejemplo, todos los párrafos de introducción, con la excepción de la última frase, del problema UVa 579 (*ClockHands*) cuentan la historia del reloj, y no tienen nada que ver con el problema propiamente dicho. Sin embargo, los problemas más difíciles suelen estar planteados de forma más aséptica, puesto que ya incluyen suficientes retos sin necesidad de adornos.
- **Descripción de la entrada y salida.** En esta sección, se proporcionan detalles del formato que tendrán los datos de entrada y del que deberían tener los de salida. Esta parte suele estar escrita en un estilo formal. Un buen problema debería determinar, claramente, cuáles son sus límites en la entrada, ya que podría resolverse con diferentes algoritmos, en función de dichos límites (ver la tabla 1.4).
- **Ejemplos de entrada y salida.** Los autores de los problemas suelen proporcionar casos de prueba triviales a los concursantes. Los ejemplos de entrada y salida sirven para verificar que los concursantes han entendido el problema en lo más básico y para comprobar si el código es, al menos, capaz de procesar un caso mínimo, y proporcionar la salida correcta en el formato indicado. Nunca envíes código al juez si no es capaz de resolver como mínimo el caso de ejemplo. En la sección 1.2.5, se trata el tema de las pruebas que se deben realizar al código antes de su envío.
- **Pistas o notas al pie.** En algunos casos, los autores del problema pueden incluir pistas o notas al pie, para facilitar la comprensión del problema.

### 1.3.2 Rutinas típicas de entrada/salida

#### Casos de prueba múltiples

En un problema de un concurso de programación, la corrección del código viene normalmente determinada por la ejecución de dicho código contra *varios* casos de prueba. En vez de utilizar varios archivos de prueba individuales, los concursos de programación modernos suelen utilizar *un* archivo de prueba con varios casos incluidos. En esta sección utilizaremos un problema muy sencillo, como ejemplo para casos de prueba múltiples: dados dos enteros expresados en

una línea, devolver su suma en una línea. Ilustramos, a continuación, tres posibles formatos de entrada/salida:

1. El número de casos de prueba aparece en la primera línea de la entrada.
2. Los casos de prueba múltiples terminan con un valor especial (normalmente ceros).
3. Los casos de prueba múltiples terminan con la señal EOF (fin de archivo).

| Código fuente C/C++                                                                                                                                                                                                       | Ejemplo de entrada                     | Ejemplo de salida          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|----------------------------|
| <pre>int TC, a, b; scanf("%d", &amp;TC); // número de casos while (TC--) { // repetir hasta llegar a 0     scanf("%d %d", &amp;a, &amp;b); // calcular respuesta     printf("%d\n", a + b); // al vuelo }</pre>           | 3<br>1 2<br>5 7<br>6 3<br><b>EOF</b>   | 3<br>12<br>9<br><b>EOF</b> |
| <pre>int a, b; // para cuando los enteros sean 0 while (scanf("%d %d", &amp;a, &amp;b), (a    b))     printf("%d\n", a + b);</pre>                                                                                        | 1 2<br>5 7<br>6 3<br>0 0<br><b>EOF</b> | 3<br>12<br>9<br><b>EOF</b> |
| <pre>int a, b; // scanf devuelve el número de elementos while (scanf("%d %d", &amp;a, &amp;b) == 2) // o podemos comprobar el EOF, p.e. // while (scanf("%d %d", &amp;a, &amp;b) != EOF)     printf("%d\n", a + b);</pre> | 1 2<br>5 7<br>6 3<br><b>EOF</b>        | 3<br>12<br>9<br><b>EOF</b> |

### Números de caso y líneas en blanco

Algunos problemas, con casos de prueba múltiples, requieren que la salida de cada caso esté numerada de forma secuencial. Otros requieren, además, una línea en blanco *después* de cada caso de prueba. Vamos a modificar el problema anterior para incluir el número de caso en la salida (contando a partir de 1) con el siguiente formato: “Case [NÚMERO]: [RESPUESTA]” seguido de una línea en blanco por cada caso. Asumiendo que la entrada finaliza con la señal EOF, podemos utilizar el siguiente código:

| Código fuente C/C++                                                                                                                                    | Ejemplo de entrada              | Ejemplo de salida                                  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|----------------------------------------------------|
| <pre>int a, b, c = 1; while (scanf("%d %d", &amp;a, &amp;b) != EOF)     // ten en cuenta los dos '\n'     printf("Case %d: %d\n\n", c++, a + b);</pre> | 1 2<br>5 7<br>6 3<br><b>EOF</b> | Case 1: 3<br>Case 2: 12<br>Case 3: 9<br><b>EOF</b> |

Otros problemas nos pedirán que escribamos una línea en blanco solo *entre* casos de prueba. Si utilizamos la solución anterior, tendremos una línea extra al final de nuestra salida, lo que resultará en un veredicto de error de presentación (PE). En su lugar, deberíamos utilizar el siguiente código:

| Código fuente C/C++                                                                                                                                   | Ejemplo de entrada       | Ejemplo de salida                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|---------------------------------------------|
| int a, b, c = 1;<br>while (scanf("%d %d", &a, &b) != EOF) {<br>if (c > 1) printf("\n"); // caso 2 y sig.<br>printf("Case %d: %d\n", c++, a + b);<br>} | 1 2<br>5 7<br>6 3<br>EOF | Case 1: 3<br>Case 2: 12<br>Case 3: 9<br>EOF |
|                                                                                                                                                       |                          |                                             |

### Número de entradas variable

Modifiquemos ligeramente el problema anterior. Ahora, para cada caso de prueba (cada línea de entrada) recibiremos un entero  $k$  ( $k \geq 1$ ), seguido de  $k$  enteros. Nuestra tarea consistirá en mostrar la suma de esos  $k$  enteros. Si damos por hecho que la entrada terminará con la señal EOF y que no debemos numerar los casos, nuestro código podría quedar así:

| Código fuente C/C++                                                                                                                       | Ejemplo de entrada                                         | Ejemplo de salida              |
|-------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|--------------------------------|
| int k, ans, v;<br>while (scanf("%d", &k) != EOF) {<br>ans = 0;<br>while (k--) { scanf("%d", &v); ans += v; }<br>printf("%d\n", ans);<br>} | 1 1<br>2 3 4<br>3 8 1 1<br>4 7 2 9 3<br>5 1 1 1 1 1<br>EOF | 1<br>7<br>10<br>21<br>5<br>EOF |
|                                                                                                                                           |                                                            |                                |

### Ejercicio 1.3.1\*

¿Qué pasaría si el autor del problema decidiese hacer que la entrada fuese *un poco más* enrevesada? En lugar de un entero  $k$  al principio de cada caso de prueba, ahora habrá que sumar todos los enteros de cada caso (cada línea). Consejo: consultar la sección 6.2.

### Ejercicio 1.3.2\*

Reescribe en Java todo el código fuente C/C++ de la sección 1.3.2.

### 1.3.3 Empieza el viaje

No hay mejor forma de empezar el viaje en el mundo de la programación competitiva que resolviendo problemas. Para ayudarte a elegir con cuáles empezar, del total de  $\approx 4097$  problemas ofrecidos en el juez en línea de la UVa [49], hemos preparado una lista con algunos de los problemas *ad hoc* más sencillos. Más sobre los problemas *ad hoc* en la sección 1.4.

- **Súper fáciles:** deberías de poder resolver estos problemas<sup>9</sup> en menos de 7 minutos<sup>10</sup>. Si acabas de llegar a la programación competitiva, te recomendamos que comiences re-

<sup>9</sup>No te frustres si no es así, ya que hay muchos motivos por los que podrías no obtener un veredicto AC.

<sup>10</sup>Lo cual es una estimación. Algunos de estos problemas se pueden resolver con una sola línea de código.

solviendo algunos problemas de esta categoría tras haber leído la pasada sección 1.3.2. Nota: como cada categoría tiene numerosos problemas para resolver, hemos *marcado con un asterisco (\*)* un máximo de tres, que deben considerarse **obligatorios**, en cada una de ellas. Consideramos que esos problemas son los más interesantes o de mayor calidad.

- **Fáciles:** hemos dividido la categoría ‘fácil’ en otras dos. Estos problemas siguen siendo sencillos, pero ‘un poco’ más difíciles que los ‘súper fáciles’.
- **Medianos (un poco por encima de los fáciles):** aquí incluimos otros problemas *ad hoc*, que pueden ser un poco más difíciles (o largos), que los de la categoría ‘fácil’.

## Ejercicios de programación

Problemas súper fáciles en el UVa Online Judge (se resuelven en menos de 7 minutos):

- |                                              |                                                                         |
|----------------------------------------------|-------------------------------------------------------------------------|
| 1. UVa 00272 - TEX Quotes                    | (sustituir todas las comillas dobles por comillas de estilo TeX)        |
| 2. UVa 01124 - Celebrity Jeopardy            | (LA 2681, basta con reimprimir la entrada)                              |
| 3. UVa 10550 - Combination Lock              | (sencillo, hacer lo que se pide)                                        |
| 4. UVa 11044 - Searching for Nessy           | (existe fórmula/código de una línea)                                    |
| 5. <b>UVa 11172 - Relational Operators *</b> | ( <i>ad hoc</i> , muy fácil, una línea)                                 |
| 6. UVa 11364 - Parking                       | (barrido lineal para obtener $l & r$ ; respuesta = $2 \times (r - l)$ ) |
| 7. <b>UVa 11498 - Division of Nlogonia *</b> | (utilizar expresiones if-else)                                          |
| 8. UVa 11547 - Automatic Answer              | (existe una solución $O(1)$ de una línea)                               |
| 9. <b>UVa 11727 - Cost Cutting *</b>         | (ordenar los tres números y obtener la mediana)                         |
| 10. UVa 12250 - Language Detection           | (LA 4995, KualaLumpur10; comprobación if-else)                          |
| 11. UVa 12279 - Emoogle Balance              | (barrido lineal simple)                                                 |
| 12. UVa 12289 - One-Two-Three                | (basta con expresiones if-else)                                         |
| 13. UVa 12372 - Packing for Holiday          | (comprobar si todos $L, W, H \leq 20$ )                                 |
| 14. UVa 12403 - Save Setu                    | (evidente)                                                              |
| 15. UVa 12577 - Hajji-e-Akbar                | (evidente)                                                              |

Fáciles (solo ‘un poco’ más difíciles que los ‘súper fáciles’):

- |                                            |                                                                                            |
|--------------------------------------------|--------------------------------------------------------------------------------------------|
| 1. UVa 00621 - Secret Research             | (análisis de casos para solo 4 salidas posibles)                                           |
| 2. <b>UVa 10114 - Loansome Car Buyer *</b> | (basta simular el proceso)                                                                 |
| 3. UVa 10300 - Ecological Premium          | (ignorar el número de animales)                                                            |
| 4. UVa 10963 - The Swallowing Ground       | (para que dos bloques se puedan combinar, los huecos entre sus columnas deben ser iguales) |
| 5. UVa 11332 - Summing Digits              | (recursividad simple)                                                                      |
| 6. <b>UVa 11559 - Event Planning *</b>     | (una pasada lineal)                                                                        |
| 7. UVa 11679 - Sub-prime                   | (comprobar si tras la simulación todos los bancos tienen una reserva $\geq 0$ )            |
| 8. UVa 11764 - Jumping Mario               | (un barrido lineal para contar los saltos alto y bajo)                                     |
| 9. <b>UVa 11799 - Horror Dash *</b>        | (un barrido lineal para encontrar el valor máximo)                                         |
| 10. UVa 11942 - Lumberjack Sequencing      | (comprobar si la entrada tiene orden ascendente o descendente)                             |
| 11. UVa 12015 - Google is Feeling Lucky    | (recorrer la lista dos veces)                                                              |
| 12. UVa 12157 - Tariff Plan                | (LA 4405, KualaLumpur08, calcular y comparar)                                              |
| 13. UVa 12468 - Zapping                    | (fácil; solo hay 4 posibilidades)                                                          |
| 14. UVa 12503 - Robot Instructions         | (simulación fácil)                                                                         |

- |                                    |                           |
|------------------------------------|---------------------------|
| 15. UVa 12554 - A Special ... Song | (simulación)              |
| 16. IOI 2010 - Cluedo              | (usar 3 punteros)         |
| 17. IOI 2010 - Memory              | (usar 2 pasadas lineales) |

**Medianos:** un poco por encima de los fáciles (se resuelven en 15-30 minutos, pero no son excesivamente difíciles):

- |                                                 |                                                                |
|-------------------------------------------------|----------------------------------------------------------------|
| 1. UVa 00119 - Greedy Gift Givers               | (simular el proceso de entrega y recepción)                    |
| 2. <b>UVa 00573 - The Snail *</b>               | (simulación, cuidado con los casos límite)                     |
| 3. UVa 00661 - Blowing Fuses                    | (simulación)                                                   |
| 4. <b>UVa 10141 - Request for Proposal *</b>    | (se resuelve con un barrido lineal)                            |
| 5. UVa 10324 - Zeros and Ones                   | (simplificar con un array unidimensional: cambiar el contador) |
| 6. UVa 10424 - Love Calculator                  | (hacer lo que se pide)                                         |
| 7. UVa 10919 - Prerequisites?                   | (procesar los requisitos según se lee la entrada)              |
| 8. <b>UVa 11507 - Bender B. Rodriguez ... *</b> | (simulación; if-else)                                          |
| 9. UVa 11586 - Train Tracks                     | (TLE con fuerza bruta, buscar el patrón)                       |
| 10. UVa 11661 - Burger Time?                    | (barrido lineal)                                               |
| 11. <i>UVa 11683 - Laser Sculpture</i>          | (una pasada lineal es suficiente)                              |
| 12. UVa 11687 - Digits                          | (simulación; evidente)                                         |
| 13. UVa 11956 - Brain****                       | (simulación; ignorar los '.' )                                 |
| 14. <i>UVa 12478 - Hardest Problem ...</i>      | (probar con uno de los ocho nombres)                           |
| 15. IOI 2009 - Garage                           | (simulación)                                                   |
| 16. IOI 2009 - POI                              | (ordenación)                                                   |

## 1.4 Los problemas *ad hoc*

Finalizaremos este capítulo tratando el primer tipo de problema en los ICPC y en las IOI: los problemas *ad hoc*. Según la USACO [50], los problemas *ad hoc* son aquellos que ‘no pueden clasificarse en ninguna otra categoría’, ya que cada enunciado y su correspondiente solución son ‘únicos’. Muchos problemas *ad hoc* son fáciles (como los que aparecen en la sección 1.3), pero esto no se aplica necesariamente a todos ellos.

Los problemas *ad hoc* aparecen frecuentemente en los concursos de programación. En el ICPC,  $\approx 1\text{-}2$  de cada  $\approx 10$  son *ad hoc*. Si el problema es fácil, será normalmente el primero que resuelvan los equipos participantes en el concurso. Sin embargo, hay casos en los que las soluciones a los problemas *ad hoc* son demasiado complicadas de implementar, lo que provoca que, por estrategia, algunos equipos prefieran abordarlos al final del concurso. En una competición regional del ICPC con unos 60 equipos, tu equipo se clasificará en la mitad baja si *solo* conseguís resolver los problemas *ad hoc* (fáciles).

En las IOI de 2009 y 2010, ha habido una tarea fácil en cada día de competición<sup>11</sup>, normalmente una tarea *ad hoc*. Si eres concursante de la IOI, puedes tener la certeza de que no ganarás ninguna medalla resolviendo únicamente las dos tareas *ad hoc* fáciles en los dos días de competición. Sin embargo, cuanto más rápido puedas librarte de esas dos tareas fáciles, más tiempo tendrás para trabajar en las otras  $2 \times 3 = 6$  tareas más complejas.

<sup>11</sup>Ya no ocurrió en las IOI 2011-2013, donde las puntuaciones fáciles se integran en la subtarea 1 de cada tarea.

En las siguientes categorías, hemos hecho una lista de **muchos** problemas *ad hoc* que hemos resuelto en el UVa Online Judge [49]. Consideramos que es posible resolver la mayoría de estos problemas *sin* utilizar estructuras de datos avanzadas o los algoritmos que estudiaremos en los siguientes capítulos. Muchos de estos problemas *ad hoc* son ‘sencillos’, pero algunos pueden ‘tener truco’. Intenta resolver alguno de cada categoría antes de pasar al próximo capítulo.

Nota: un pequeño conjunto de problemas, aunque aparece como parte del capítulo 1, puede requerir conocimientos de los siguientes capítulos como, por ejemplo, las estructuras de datos lineales (*arrays*) de la sección 2.2, el *backtracking* de la sección 3.2, etc. Puedes revisitar los problemas *ad hoc* más difíciles una vez que entiendas los conceptos necesarios.

Las categorías:

- **Juegos (naipes):**

Hay muchos problemas *ad hoc* relacionados con juegos populares, bastantes de ellos de naipes. Normalmente, tendrás que procesar las cadenas de entrada (ver la sección 6.3) como naipes que tienen tanto palos (D/Diamantes/, T/Tréboles/, C/Corazones/ y P/Picas/) como números (normalmente:  $2 < 3 < \dots < 9 < D < Diez < J < Jota < Q < Reina < K < Rey < A < \text{Ás}^{12}$ ). Puede ser una buena idea convertir estas cadenas a índices de enteros, para evitar problemas. Por ejemplo, un mapa posible sería convertir D2 → 0, D3 → 1, ..., DA → 12, T2 → 13, T3 → 14, ..., PA → 51. A partir de ahí, se puede trabajar con índices enteros.

- **Juegos (ajedrez):**

El ajedrez es otro popular juego que, en ocasiones, aparece en los problemas de los concursos de programación. Algunos de estos problemas son *ad hoc* y aparecen en esta sección. Otros son de combinatoria, con tareas como contar cuántas formas hay de colocar 8 reinas en un tablero de ajedrez de  $8 \times 8$ . Esos los trataremos en el capítulo 3.

- **Juegos (otros), fáciles y difíciles (o más tediosos):**

Aparte de los juegos de naipes y de ajedrez, muchos otros juegos populares han encontrado un lugar en los concursos de programación: tres en raya, piedra-papel-tijera, serpientes y escaleras, bingo, bolos, etc. Conocer las reglas de estos juegos puede resultar útil aunque, la mayoría de las veces, la información necesaria aparecerá en el enunciado del problema, para evitar perjudicar a los concursantes que no estén familiarizados con ellos.

- **Problemas relacionados con palíndromos:**

Estos también son un clásico dentro de los concursos de programación. Un palíndromo es una palabra (o una secuencia) que se lee de la misma forma en cualquier dirección. La estrategia más común para comprobar si una palabra es palíndroma, es hacer un bucle desde el primer carácter hasta el carácter *central* y comprobar si estos coinciden con los de la posición correspondiente, empezando por el final. Por ejemplo, ‘ABCDCBA’ es un palíndromo. Para resolver algunos problemas de palíndromos más complicados, deberás leer la sección 6.5, donde encontrarás soluciones de programación dinámica.

- **Problemas relacionados con anagramas:**

Otro clásico más. Un anagrama es una palabra (o frase) cuyas letras se pueden reordenar para obtener otra palabra (o frase). La estrategia más común, para comprobar si dos palabras son anagramas entre ellas es ordenar sus letras y comparar los resultados. Por ejemplo, tomemos *palabraA* = ‘cab’, *palabraB* = ‘bca’. Después de la ordenación,

---

<sup>12</sup>En ocasiones A/Ás < 2.

encontraremos que `palabraA` = ‘abc’ y `palabraB` = ‘abc’, por lo que son anagramas. Puedes encontrar varias técnicas de ordenación en la sección 2.2.

- Problemas interesantes **de la vida real**, fáciles y difíciles (o más tediosos):

Ésta es una de las categorías más relevantes de entre las que encontramos en el UVa Online Judge. Creemos que los problemas de la vida real, como estos, son interesantes para cualquier recién llegado al mundo de la informática. El hecho de programar para resolver problemas de la vida real, puede ser una motivación adicional. Quién sabe, quizás puedas aprender algo nuevo (e interesante) al leer el enunciado del problema.

- Problemas *ad hoc* relacionados con el **tiempo**:

Estos problemas utilizan conceptos temporales como fechas, horas y calendarios. También son problemas de la vida real. Como ya hemos mencionado, estos problemas pueden resultar un poco más interesantes de resolver. Algunos de ellos son mucho más fáciles de abordar si dominamos la clase de Java `GregorianCalendar`, ya que cuenta con muchas funciones para tratar con el tiempo.

- Problemas para ‘perder el tiempo’:

Estos problemas *ad hoc* están escritos, expresamente, para hacer que la solución sea larga y aburrida. Si aparecen en un concurso de programación, determinarán qué equipo cuenta con el programador más *eficiente*, alguien que es capaz de implementar soluciones complicadas y precisas dentro de un límite de tiempo. Los entrenadores deberían incluir este tipo de problemas en sus sesiones de preparación.

- Problemas *ad hoc* en **otros capítulos**:

Hay muchos problemas *ad hoc* que hemos desplazado a otros capítulos, ya que su solución requiere conocimientos que están por encima de las capacidades de programación básicas, pero podría ser una buena idea echarles un vistazo al terminar este capítulo 1.

- Problemas *ad hoc* que requieren del uso de estructuras de datos lineales básicas (sobre todo *arrays*), en la sección 2.2.
- Problemas *ad hoc* que requieren cálculos matemáticos, en la sección 5.2.
- Problemas *ad hoc* que requieren procesamiento de cadenas, en la sección 6.3.
- Problemas *ad hoc* que requieren geometría básica, en la sección 7.2.
- Problemas *ad hoc*, en el capítulo 9.

### Consejo

Después de resolver algunos problemas de programación, empezarás a encontrar un patrón en tus soluciones. Algunos elementos se utilizan con la suficiente frecuencia en la programación competitiva como para que resulte útil implementar atajos. Desde una perspectiva de C/C++, estos elementos podrían ser: bibliotecas que se deben incluir (`cstdio`, `cmath`, `cstring`, etc.), tipos de datos (`ii`, `vii`, `vi`, etc.), rutinas básicas de E/S (`freopen`, formatos de entrada múltiple, etc.), macros para bucles (por ejemplo, `#define REP(i, a, b) for (int i = int(a); i <= int(b); i++)`, etc.) y algunos otros. Un programador competitivo que utilice C/C++, puede almacenar todos estos elementos en un archivo de cabecera como ‘`competitive.h`’. Con este archivo a mano, la solución

para cada problema empezará con un sencillo `#include<competitive.h>`. Sin embargo, este consejo no debe ser tenido en cuenta más allá del mundo de la programación competitiva, especialmente en la industria del software.

## Ejercicios de programación

### Ejercicios de programación relacionados con problemas *ad hoc*:

#### Juegos (naipes)

- |                                              |                                                                                                    |
|----------------------------------------------|----------------------------------------------------------------------------------------------------|
| 1. UVa 00162 - Beggar My Neighbour           | (simulación de juego de naipes; evidente)                                                          |
| 2. <b>UVa 00462 - Bridge Hand Evaluator*</b> | (simulación; naipes)                                                                               |
| 3. UVa 00555 - Bridge Hands                  | (juego de naipes)                                                                                  |
| 4. UVa 10205 - Stack 'em Up                  | (juego de naipes)                                                                                  |
| 5. UVa 10315 - Poker Hands                   | (problema tedioso)                                                                                 |
| 6. <b>UVa 10646 - What is the Card?*</b>     | (barajar los naipes mediante alguna regla y obtener uno concreto)                                  |
| 7. UVa 11225 - Tarot scores                  | (otro juego de naipes)                                                                             |
| 8. UVa 11678 - Card's Exchange               | (en realidad, es un problema de manipulación de <i>arrays</i> )                                    |
| 9. <b>UVa 12247 - Jollo*</b>                 | (interesante juego de naipes, pero requiere una buena lógica para pasar todos los casos de prueba) |

#### Juegos (ajedrez)

- |                                           |                                                                                                               |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| 1. UVa 00255 - Correct Move               | (comprobar la validez de los movimientos de ajedrez)                                                          |
| 2. <b>UVa 00278 - Chess*</b>              | ( <i>ad hoc</i> ; ajedrez; existe una fórmula cerrada)                                                        |
| 3. <b>UVa 00696 - How Many Knights*</b>   | ( <i>ad hoc</i> ; ajedrez)                                                                                    |
| 4. UVa 10196 - Check The Check            | (juego de ajedrez <i>ad hoc</i> ; tedioso)                                                                    |
| 5. <b>UVa 10284 - Chessboard in FEN *</b> | (FEN = Notación Forsyth-Edwards, es una notación estándar para describir posiciones de un tablero de ajedrez) |
| 6. UVa 10849 - Move the bishop            | (ajedrez)                                                                                                     |
| 7. UVa 11494 - Queen                      | ( <i>ad hoc</i> , ajedrez)                                                                                    |

#### Juegos (otros), más fáciles

- |                                           |                                                                               |
|-------------------------------------------|-------------------------------------------------------------------------------|
| 1. UVa 00340 - Master-Mind Hints          | (determinar los emparejamientos fuertes y débiles)                            |
| 2. <b>UVa 00489 - Hangman Judge*</b>      | (hacer lo que se pide)                                                        |
| 3. UVa 00947 - Master Mind Helper         | (similar a UVa 340)                                                           |
| 4. <b>UVa 10189 - Minesweeper*</b>        | (simular el buscaminas; similar a UVa 10279)                                  |
| 5. UVa 10279 - Mine Sweeper               | (un <i>array</i> bidimensional ayuda; similar a UVa 10189)                    |
| 6. UVa 10409 - Die Game                   | (basta simular el movimiento de la muerte)                                    |
| 7. UVa 10530 - Guessing Game              | (utilizar un <i>array</i> de etiquetas unidimensional)                        |
| 8. <b>UVa 11459 - Snakes and Ladders*</b> | (simularlo; similar a UVa 647)                                                |
| 9. UVa 12239 - Bingo                      | (probar las $90^2$ parejas; ver si están todos los números de $[0 \dots N]$ ) |

#### Juegos (otros), más difíciles (tediosos)

- |                                    |                                                |
|------------------------------------|------------------------------------------------|
| 1. UVa 00114 - Simulation Wizardry | (simulación de una máquina de <i>pinball</i> ) |
|------------------------------------|------------------------------------------------|

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>2. UVa 00141 - The Spot Game</li> <li>3. UVa 00220 - Othello</li> <li>4. UVa 00227 - Puzzle</li> <li>5. UVa 00232 - Crossword Answers</li> <li>6. UVa 00339 - SameGame Simulation</li> <li>7. UVa 00379 - HI-Q</li> <li>8. <b>UVa 00584 - Bowling *</b></li> <li>9. UVa 00647 - Chutes and Ladders</li> <li>10. UVa 10363 - Tic Tac Toe</li> <li>11. <b>UVa 10443 - Rock, Scissors, Paper *</b></li> <li>12. <b>UVa 10813 - Traditional BINGO *</b></li> <li>13. UVa 10903 - Rock-Paper-Scissors ...</li> </ol> | <p>(simulación; comprobación de patrones)</p> <p>(seguir las reglas del juego; un poco tedioso)</p> <p>(procesar la entrada; manipulación de <i>arrays</i>)</p> <p>(problema de manipulación de <i>arrays</i> compleja)</p> <p>(seguir la descripción del problema)</p> <p>(seguir la descripción del problema)</p> <p>(simulación; juegos; comprensión lectora)</p> <p>(juego de tablero infantil; ver también UVa 11459)</p> <p>(comprobar la validez de la partida de tres en raya; no evidente)</p> <p>(manipulación de <i>arrays</i> bidimensionales)</p> <p>(seguir la descripción del problema)</p> <p>(contar victorias y derrotas, mostar la media de victorias)</p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Palíndromos

- |                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. UVa 00353 - Pesky Palindromes</li> <li>2. <b>UVa 00401 - Palindromes *</b></li> <li>3. UVa 10018 - Reverse and Add</li> <li>4. <b>UVa 10945 - Mother Bear *</b></li> <li>5. <b>UVa 11221 - Magic Square Palindrome *</b></li> <li>6. UVa 11309 - Counting Chaos</li> </ol> | <p>(procesar las subcadenas por fuerza bruta)</p> <p>(comprobación de palíndromos sencilla)</p> <p>(<i>ad hoc</i>; matemáticas; comprobación de palíndromos)</p> <p>(palíndromos)</p> <p>(tratamos con una matriz)</p> <p>(comprobación de palíndromos)</p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Anagramas

- |                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. UVa 00148 - Anagram Checker</li> <li>2. <b>UVa 00156 - Ananagram *</b></li> <li>3. <b>UVa 00195 - Anagram *</b></li> <li>4. <b>UVa 00454 - Anagrams *</b></li> <li>5. UVa 00630 - Anagrams (III)</li> <li>6. UVa 00642 - Word Amalgamation</li> <li>7. UVa 10098 - Generating Fast, Sorted ...</li> </ol> | <p>(utiliza <i>backtracking</i>)</p> <p>(más fácil con <code>algorithm::sort</code>)</p> <p>(más fácil con <code>algorithm::next_permutation</code>)</p> <p>(anagramas)</p> <p>(<i>ad hoc</i>; cadenas)</p> <p>(recorrer el pequeño diccionario dado para ver la lista de posibles anagramas)</p> <p>(muy similar a UVa 195)</p> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Problemas interesantes de la vida real, más fáciles

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. <b>UVa 00161 - Traffic Lights *</b></li> <li>2. UVa 00187 - Transaction Processing</li> <li>3. UVa 00362 - 18,000 Seconds Remaining</li> <li>4. <b>UVa 00637 - Booklet Printing *</b></li> <li>5. <b>UVa 00857 - Quantiser</b></li> <li>6. UVa 10082 - WERTYU</li> <li>7. UVa 10191 - Longest Nap</li> <li>8. UVa 10528 - Major Scales</li> <li>9. UVa 10554 - Calories from Fat</li> <li>10. <b>UVa 10812 - Beat the Spread *</b></li> <li>11. UVa 11530 - SMS Typing</li> <li>12. <b>UVa 11945 - Financial Management</b></li> <li>13. UVa 11984 - A Change in Thermal Unit</li> </ol> | <p>(una situación típica en la carretera)</p> <p>(un problema de contabilidad)</p> <p>(situación típica de descarga de archivos)</p> <p>(aplicación con el controlador de una impresora)</p> <p>(MIDI; aplicación sobre música generada por ordenador)</p> <p>(a veces cometemos este error tipográfico)</p> <p>(quizá te interese en tu vida diaria)</p> <p>(hay conocimiento musical en la descripción del problema)</p> <p>(¿te preocupa tu peso?)</p> <p>(cuidado con los casos límite)</p> <p>(solía pasar con los teléfonos móviles antiguos)</p> <p>(un poco de formato en la salida)</p> <p>(conversión de °F a °C y viceversa)</p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

14. UVa 12195 - *Jingle Composing*  
 15. UVa 12555 - *Baby Me*  
 (contar el número de compases correctos)  
 (una de las primeras preguntas cuando nace un bebé;  
 hay que procesar un poco la entrada)

### Problemas interesantes de la vida real, más difíciles (tediosos)

1. UVa 00139 - *Telephone Tangles*  
 2. UVa 00145 - *Gondwanaland Telecom*  
 3. UVa 00333 - *Recognizing Good ISBNs*  
 (calcular la factura del teléfono; manipulación de cadenas)  
 (similar en concepto a UVa 139)  
 (nota: este problema tiene casos de prueba 'erróneos' con líneas  
 en blanco que pueden causar, potencialmente, muchos 'errores  
 de presentación')
4. UVa 00346 - *Getting Chorded*  
**5. UVa 00403 - Postscript \***  
 6. UVa 00447 - *Population Explosion*  
 7. UVa 00448 - *OOPS*  
 8. UVa 00449 - *Majoring in Scales*  
 9. UVa 00457 - *Linear Cellular Automata*  
 (acorde musical; mayor/menor)  
 (emulación de un controlador de impresora; tedioso)  
 (modelo de simulación de la vida)  
 (conversión tediosa de 'hexadecimal' a 'ensamblador')  
 (más fácil si tienes conocimientos de música)  
 (simulación simplificada del 'juego de la vida'; idea similar a UVa  
 447; busca el término en internet)
10. UVa 00538 - *Balancing Bank Accounts*  
**11. UVa 00608 - Counterfeit Dollar \***  
 12. UVa 00706 - *LC-Display*  
**13. UVa 01061 - Consanguine Calculations \***  
 (la premisa del problema es muy cierta)  
 (problema clásico)  
 (lo que vemos en las pantallas digitales antiguas)  
 (LA 3736 - WorldFinals Tokyo07; consanguinidad = sangre; este  
 problema cuestiona posibles combinaciones de grupos  
 sanguíneos y factor RH; se resuelve probando los ocho tipos  
 posibles de sangre+RH, con la información proporcionada en la  
 descripción del problema)
14. UVa 10415 - *Eb Alto Saxophone Player*  
 15. UVa 10659 - *Fitting Text into Slides*  
 16. UVa 11223 - *0: dah, dah, dah*  
 17. UVa 11743 - *Credit Check*  
 (sobre instrumentos musicales)  
 (los programas de presentación típicos hacen esto)  
 (conversión tediosa a código morse)  
 (algoritmo de Luhn para comprobar números de tarjetas de  
 crédito; busca más información en internet)
18. UVa 12342 - *Tax Calculator*  
 (el cálculo de impuestos tiene sus complicaciones)

### Tiempo

1. UVa 00170 - *Clock Patience*  
 2. UVa 00300 - *Maya Calendar*  
**3. UVa 00579 - Clock Hands \***  
**4. UVa 00893 - Y3K \***  
 5. UVa 10070 - *Leap Year or Not Leap ...*  
 6. UVa 10339 - *Watching Watches*  
 7. UVa 10371 - *Time Zones*  
 8. UVa 10683 - *The decadary watch*  
 9. UVa 11219 - *How old are you?*  
 10. UVa 11356 - *Dates*  
 11. UVa 11650 - *Mirror Clock*  
 12. UVa 11677 - *Alarm Clock*  
 (simulación; tiempo)  
 (ad hoc; tiempo)  
 (ad hoc; tiempo)  
 (utilizar GregorianCalendar de Java; similar a UVa 11356)  
 (más que los años bisiestos normales)  
 (encontrar la fórmula)  
 (seguir la descripción del problema)  
 (conversión simple de sistemas de relojes)  
 (cuidado con los casos límite)  
 (muy fácil usando GregorianCalendar de Java)  
 (se requieren matemáticas)  
 (idea similar a UVa 11650)

- |                                            |                                                                                                                                          |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| 13. <b>UVa 11947 - Cancer or Scorpio</b> * | (más fácil con <code>GregorianCalendar</code> de Java)                                                                                   |
| 14. UVa 11958 - Coming Home                | (cuidado con ‘después de medianoche’)                                                                                                    |
| 15. UVa 12019 - Doom’s Day Algorithm       | ( <code>GregorianCalendar</code> ; obtener <code>DAY_OF_WEEK</code> )                                                                    |
| 16. UVa 12136 - Schedule of a Married Man  | (LA 4202; Dhaka08, comprobar tiempo)                                                                                                     |
| 17. <i>UVa 12148 - Electricity</i>         | (fácil con <code>GregorianCalendar</code> ; usar el método ‘add’ para añadir un día a la fecha anterior y ver si coincide con la actual) |
| 18. <i>UVa 12439 - February 29</i>         | (inclusión-exclusión; muchos casos límite; cuidado)                                                                                      |
| 19. <i>UVa 12531 - Hours and Minutes</i>   | (ángulos entre dos agujas de un reloj)                                                                                                   |

### Problemas para ‘perder el tiempo’

- |                                                   |                                                                                     |
|---------------------------------------------------|-------------------------------------------------------------------------------------|
| 1. UVa 00144 - Student Grants                     | (simulación)                                                                        |
| 2. <i>UVa 00214 - Code Generation</i>             | (basta simular el proceso; cuidado al restar (-), dividir (/) y negar (@); tedioso) |
| 3. UVa 00335 - Processing MX Records              | (simulación)                                                                        |
| 4. UVa 00337 - Interpreting Control ...           | (simulación; relativo a la salida)                                                  |
| 5. UVa 00349 - Transferable Voting (II)           | (simulación)                                                                        |
| 6. UVa 00381 - Making the Grade                   | (simulación)                                                                        |
| 7. UVa 00405 - Message Routing                    | (simulación)                                                                        |
| 8. <b>UVa 00556 - Amazing</b> *                   | (simulación)                                                                        |
| 9. <i>UVa 00603 - Parking Lot</i>                 | (simular el proceso requerido)                                                      |
| 10. <i>UVa 00830 - Shark</i>                      | (muy difícil conseguir AC; al menor error = WA)                                     |
| 11. <i>UVa 00945 - Loading a Cargo Ship</i>       | (simular el proceso de carga de mercancía dado)                                     |
| 12. UVa 10033 - Interpreter                       | ( <i>ad hoc</i> , simulación)                                                       |
| 13. <i>UVa 10134 - AutoFish</i>                   | (hay que ser muy cuidadoso con los detalles)                                        |
| 14. UVa 10142 - Australian Voting                 | (simulación)                                                                        |
| 15. UVa 10188 - Automated Judge Script            | (simulación)                                                                        |
| 16. UVa 10267 - Graphical Editor                  | (simulación)                                                                        |
| 17. <i>UVa 10961 - Chasing After Don Giovanni</i> | (simulación tediosa)                                                                |
| 18. UVa 11140 - Little Ali’s Little Brother       | ( <i>ad hoc</i> )                                                                   |
| 19. UVa 11717 - Energy Saving Micro...            | (simulación complicada)                                                             |
| 20. <b>UVa 12060 - All Integer Average</b> *      | (LA 3012; Dhaka04; formato de salida)                                               |
| 21. <b>UVa 12085 - Mobile Casanova</b> *          | (LA 2189; Dhaka06; cuidado con los PE)                                              |
| 22. <i>UVa 12608 - Garbage Collection</i>         | (simulación con varios casos límite)                                                |

## 1.5 Soluciones a los ejercicios no resaltados

**Ejercicio 1.1.1:** un caso de prueba sencillo para hacer fallar a algoritmos voraces es  $N = 2$ ,  $\{(2, 0), (2, 1), (0, 0), (4, 0)\}$ . Un algoritmo voraz emparejará de forma incorrecta  $\{(2, 0), (2, 1)\}$  y  $\{(0, 0), (4, 0)\}$  con un coste de 5,000, mientras que la solución óptima es emparejar  $\{(0, 0), (2, 0)\}$  y  $\{(2, 1), (4, 0)\}$  con un coste de 4,236.

**Ejercicio 1.1.2:** para realizar una búsqueda completa ingenua, como la descrita en el texto, son necesarias hasta  ${}_{16}C_2 \times {}_{14}C_2 \times \dots \times {}_2C_2$  operaciones para el caso más grande, con  $N = 8$ , lo que es demasiado. Sin embargo, hay formas de podar el espacio de búsqueda para que la búsqueda completa pueda ser operativa. También puedes intentar el **ejercicio 1.1.3\***.

**Ejercicio 1.2.1:** la tabla 1.3 completa es la siguiente.

| UVa   | Título               | Tipo de problema                                  | Más información |
|-------|----------------------|---------------------------------------------------|-----------------|
| 10360 | Rat Attack           | Búsqueda completa o DP                            | Sección 3.2     |
| 10341 | Solve It             | Divide y vencerás (método de bisección)           | Sección 3.3     |
| 11292 | Dragon of Loowater   | Voraz (no clásico)                                | Sección 3.4     |
| 11450 | Wedding Shopping     | DP (no clásico)                                   | Sección 3.5     |
| 10911 | Forming Quiz Teams   | DP con máscara de bits (no clásico)               | Sección 8.3.1   |
| 11635 | Hotel Booking        | Grafos (descomposición: Dijkstra + BFS)           | Sección 8.4     |
| 11506 | Angry Programmer     | Grafos (corte mínimo/flujo máximo)                | Sección 4.6     |
| 10243 | Fire! Fire!! Fire!!! | DP en árbol (cobertura de vértices mínima)        | Sección 4.7.1   |
| 10717 | Mint                 | Descomposición: búsqueda completa y matemáticas   | Sección 8.4     |
| 11512 | GATTACA              | Cadenas ( <i>array</i> de sufijos, LCP, LRS)      | Sección 6.6     |
| 10065 | Useless Tile Packers | Geometría (envolvente convexa y área de polígono) | Sección 7.3.7   |

**Ejercicio 1.2.2:** las respuestas son:

1. (b) Utilizar una estructura de datos de cola de prioridad (montículo) (sección 2.3).
2. Si la lista  $L$  es estática, (b) *array* sencillo procesado previamente con programación dinámica (secciones 2.2 y 3.5). Si la lista  $L$  es dinámica, entonces (f) Árbol binario indexado (Fenwick), es una respuesta mejor (más fácil de implementar que (e) árbol de segmentos).
3. (b) Usar consulta de suma de rangos bidimensional (sección 3.5.2).
4. (a) Sí, esa búsqueda completa es posible (sección 3.2).
5. (a) Programación dinámica en  $O(V + E)$  (secciones 3.5, 4.2.5 y 4.7.1). Sin embargo, también es posible (c) algoritmo de Dijkstra en  $O((V+E) \log V)$ , ya que el factor adicional  $O(\log V)$  sigue siendo ‘pequeño’ para un valor de  $V$  hasta  $100K$  y, en un concurso de programación, es difícil separar este factor log.
6. (a) criba de Eratóstenes (sección 5.5.1).
7. (b) La técnica ingenua del punto anterior no funcionará. Debemos descomponer en factores (primos)  $n!$  y  $m$  y ver si los factores de  $m$  están entre los de  $n!$  (sección 5.5.5).
8. (b) No, hay que encontrar otra forma. En primer lugar, buscar la envolvente convexa de los  $N$  puntos en  $O(n \log n)$  (sección 7.3.7). Digamos que hay  $CH(S) = k$  puntos. Como están distribuidos aleatoriamente,  $k$  será mucho más pequeño que  $N$ . Después, encontrar los dos puntos más alejados, examinando todos los pares de puntos de  $CH(S)$  en  $O(k^2)$ .
9. (b) La técnica ingenua es muy lenta. Utilizar KMP o *array* de sufijos (secciones 6.4 o 6.6).

**Ejercicio 1.2.3:** a continuación, se reproduce el código Java:

```
1 // Código Java para la tarea 1, asumiendo todos los import necesarios
2 class Main {
3 public static void main(String[] args) {
4 Scanner sc = new Scanner(System.in);
5 double d = sc.nextDouble();
```

```

6 System.out.printf("%7.3f\n", d); // sí, Java también tiene printf
7 }
8
9 // Código C++ para la tarea 2, asumiendo todos los include necesarios
10 int main() {
11 double pi = 2 * acos(0.0); // una forma más precisa de calcular pi
12 int n; scanf("%d", &n);
13 printf("%.1lf\n", n, pi); // así se manipula la anchura del campo
14 }
15
16 // Código Java para la tarea 3, asumiendo todos los import necesarios
17 class Main {
18 public static void main(String[] args) {
19 String[] names = new String[]
20 { "", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
21 Calendar calendar = new GregorianCalendar(2010, 7, 9); // 9 agosto 2010
22 // los meses empiezan en 0, así que ponemos 7 en vez de 8
23 System.out.println(names[calendar.get(Calendar.DAY_OF_WEEK)]); // "Mon"
24 }
25
26 // Código C++ para la tarea 4, asumiendo todos los include necesarios
27 #define ALL(x) x.begin(), x.end()
28 #define UNIQUE(c) (c).resize(unique(ALL(c)) - (c).begin())
29
30 int main() {
31 int a[] = {1, 2, 2, 2, 3, 3, 2, 2, 1};
32 vector<int> v(a, a + 9);
33 sort(ALL(v)); UNIQUE(v);
34 for (int i = 0; i < (int)v.size(); i++) printf("%d\n", v[i]);
35 }
36
37 // Código C++ para la tarea 5, asumiendo todos los include necesarios
38 typedef pair<int, int> ii; // utilizaremos el orden natural
39 typedef pair<ii, int> iii; // de los tipos de datos que hemos emparejado
40
41 int main() {
42 iii A = make_pair(ii(5, 24), -1982); // reordenar DD/MM/AAAA
43 iii B = make_pair(ii(5, 24), -1980); // o MM, DD,
44 iii C = make_pair(ii(11, 13), -1983); // y utilizar AAAA NEGATIVO
45 vector<iii> birthdays;
46 birthdays.push_back(A); birthdays.push_back(B); birthdays.push_back(C);
47 sort(birthdays.begin(), birthdays.end()); // y ya está :)
48 }
49
50 // Código C++ para la tarea 6, asumiendo todos los include necesarios
51 int main() {
52 int n = 5, L[] = {10, 7, 5, 20, 8}, v = 7;

```

```

53 sort(L, L + n);
54 printf("%d\n", binary_search(L, L + n, v));
55 }
56
57 // Código C++ para la tarea 7, asumiendo todos los include necesarios
58 int main() {
59 int p[10], N = 10; for (int i = 0; i < N; i++) p[i] = i;
60 do {
61 for (int i = 0; i < N; i++) printf("%c ", 'A' + p[i]);
62 printf("\n");
63 }
64 while (next_permutation(p, p + N));
65 }
66
67 // Código C++ para la tarea 8, asumiendo todos los include necesarios
68 int main() {
69 int p[20], N = 20;
70 for (int i = 0; i < N; i++) p[i] = i;
71 for (int i = 0; i < (1 << N); i++) {
72 for (int j = 0; j < N; j++)
73 if (i & (1 << j)) // si el bit j está activo
74 printf("%d ", p[j]); // esto es parte del conjunto
75 printf("\n");
76 }
77
78 // Código Java para la tarea 9, asumiendo todos los import necesarios
79 class Main {
80 public static void main(String[] args) {
81 String str = "FF"; int X = 16, Y = 10;
82 System.out.println(new BigInteger(str, X).toString(Y));
83 }
84
85 // Código Java para la tarea 10, asumiendo todos los import necesarios
86 class Main {
87 public static void main(String[] args) {
88 String S = "line: a70 and z72 will be replaced, aa24 and a872 will not";
89 System.out.println(S.replaceAll("\\b[a-z][0-9][0-9]\\b", "***"));
90 }
91
92 // Código Java para la tarea 11, asumiendo todos los import necesarios
93 class Main {
94 public static void main(String[] args) throws Exception {
95 ScriptEngineManager mgr = new ScriptEngineManager();
96 ScriptEngine engine = mgr.getEngineByName("JavaScript"); // "trampas"
97 Scanner sc = new Scanner(System.in);
98 while (sc.hasNextLine()) System.out.println(engine.eval(sc.nextLine()));
99 }

```

**Ejercicio 1.2.4:** las consideraciones situacionales están entre paréntesis:

1. Obtienes un veredicto de WA en un problema muy sencillo. ¿Qué deberías hacer?
  - a) Abandonar este problema y seguir con otro. (**No es buena idea, perderéis.**)
  - b) Mejorar el rendimiento de tu solución. (**No es útil.**)
  - c) Crear casos de prueba complejos para tratar de encontrar el error. (**Lo lógico.**)
  - d) Si la competición es por equipos, pedirle a un compañero de equipo que trate de resolverlo. (**Esta opción podría ser factible si no has entendido bien el problema. Por lo tanto, deberías evitar darle a tu compañero cualquier explicación sobre el mismo, o tu solución. Aun así, tu equipo perderá un tiempo precioso.**)
2. Recibes un veredicto de TLE para tu solución  $O(N^3)$ . Sin embargo, el  $N$  máximo es 100. ¿Qué deberías hacer?
  - a) Abandonar este problema y seguir con otro. (**No es buena idea, perderéis.**)
  - b) Mejorar el rendimiento de tu solución. (**No sirve ya que no deberíamos tener un TLE en un algoritmo  $O(N^3)$  si  $N \leq 400$ .**)
  - c) Crear casos de prueba complejos para tratar de encontrar el error. (**Esta es la respuesta, quizás el programa entre accidentalmente en un bucle infinito en algunos casos de prueba.**)
3. Ampliación a la pregunta 2: ¿qué ocurre si el máximo de  $N$  es 100.000?  
(**Si  $N > 400$  puede que no tengas otra opción que mejorar el rendimiento del algoritmo actual, o utilizar otro más rápido.**)
4. Otra ampliación a la pregunta 2: ¿qué ocurre si el máximo de  $N$  es 1.000, la salida solo depende del tamaño de la entrada  $N$  y todavía quedan *cuatro horas* de concurso?  
(**Si la salida solo depende de  $N$ , podrías llegar a calcular previamente todas las soluciones posibles ejecutando el algoritmo  $O(N^3)$  localmente, mientras otros miembros del equipo utilizan el ordenador. Una vez que la solución  $O(N^3)$  haya terminado de ejecutarse, tendrás todas las respuestas posibles. Ahora, podrás enviar la solución  $O(1)$  en su lugar, siempre que no supere el tamaño máximo del código fuente impuesto por el juez.**)
5. Recibes un veredicto de RTE. El código, aparentemente, se ejecuta perfectamente en tu sistema local. ¿Qué deberías hacer?  
(**La mayoría de los veredictos RTE vienen provocados por tamaños de array que son demasiado pequeños o errores de desbordamiento de pila o recursión infinita. Diseña casos de prueba que provoquen estos errores en tu código.**)
6. Treinta minutos después de comenzar el concurso, le echas un vistazo al marcador. Hay *muchos* equipos que han resuelto el problema  $X$ , que tu equipo todavía no ha intentado. ¿Qué deberías hacer?  
(**Un miembro del equipo debería de tratar de resolver el problema  $X$  inmediatamente, ya que probablemente será muy fácil. La situación no es nada buena para tu equipo, ya que supondrá un inconveniente a la hora de conseguir una buena clasificación en el concurso.**)

7. A mitad del concurso, le echas un vistazo al marcador. El equipo líder (asumiento que no sea el tuyo) acaba de resolver el problema *Y*. ¿Qué deberías hacer?  
**(Si tu equipo no es el que va ‘marcando el ritmo’, no es mala idea ‘ignorar’ lo que va haciendo el equipo líder y concentrarse en resolver los problemas que consideráis abordables. A esas alturas del concurso, tu equipo ya debería haber leído todos los problemas e identificado los que podéis resolver dentro de vuestras posibilidades.)**
8. Tu equipo le ha dedicado dos horas a un problema complicado. Ya habéis enviado varias implementaciones realizadas por diferentes miembros del equipo. Todos los envíos han resultado incorrectos. No tenéis ni idea de dónde está el fallo. ¿Qué deberías hacer?  
**(Es hora de abandonar este problema. No ocupes el ordenador y permite que otro compañero de equipo pueda resolver otro problema. O bien no habéis entendido el problema o, en casos muy raros, la solución que utiliza el juez es errónea. En cualquier caso, no es una buena situación para el equipo.)**
9. Queda una hora para que termine el concurso. Tienes un código con veredicto WA y una idea nueva para *otro* problema. ¿Qué deberías hacer?  
**(Utilizando terminología de ajedrez, estás ante una situación de ‘final’.)**
- a) Abandonar el problema con el código WA e intentar el otro problema para tratar de resolver uno más. **(Buena opción en concursos individuales como la IOI.)**
  - b) Insistir en depurar el código WA. No queda tiempo para comenzar a trabajar en un nuevo problema. **(Si la idea para nuestro problema implica una solución compleja y tediosa, centrarse en el código con respuesta incorrecta puede ser una buena vía para evitar tener dos soluciones no aceptadas.)**
  - c) En el ICPC, imprimir el código WA y pedir a los otros dos miembros del equipo que lo analicen, mientras tú comienzas a trabajar en el problema nuevo, en un intento de resolver *dos* problemas más. **(Si la solución al otro problema se puede implementar en menos de 30 minutos, hazlo mientras tus compañeros intentan encontrar, sobre el código en papel, el error que causa el WA.)**

## 1.6 Notas del capítulo

Este capítulo, así como los siguientes, se apoya en muchos libros de texto (ver la figura 1.4, en la página siguiente) y recursos disponibles en internet. A continuación, incluimos algunas referencias adicionales:

- Para mejorar tu velocidad en mecanografía, como se indica en el consejo 1, puedes jugar a alguno de los muchos juegos sobre el tema que están disponibles en internet.
- El consejo 2 viene del texto de introducción de la plataforma de USACO [50].
- Se pueden encontrar más detalles sobre el consejo 3 en muchos libros de ciencias de la computación, como por ejemplo en los capítulos 1 a 5 y 17 de [7].
- Referencias en línea para el consejo 4: para la STL de C++, [www.cppreference.com](http://www.cppreference.com) y [www.sgi.com/tech/stl/](http://www.sgi.com/tech/stl/), y para la API de Java, [docs.oracle.com/javase/8/docs/api/](http://docs.oracle.com/javase/8/docs/api/). No es necesario memorizar todas las funciones de las bibliotecas, pero resulta útil aprender bien las que utilices con mayor frecuencia.

- Para mejorar las técnicas de prueba del código (consejo 5), sería aconsejable la consulta de algunos libros sobre ingeniería de software.
- Hay muchos otros jueces en línea, aparte de los mencionados en el consejo 6, como:
  - Codeforces, <http://codeforces.com/>
  - Peking University Online Judge, (POJ) <http://poj.org>.
  - Zhejiang University Online Judge, (ZOJ) <http://acm.zju.edu.cn>.
  - Tianjin University Online Judge, <http://acm.tju.edu.cn/toj>.
  - Ural State University (Timus) Online Judge, <http://acm.timus.ru>.
  - URI Online Judge, <http://www.urionlinejudge.edu.br>, etc.
- Para un comentario en relación a los concursos por equipos (consejo 7), ver [16].

En este capítulo, te hemos presentado el mundo de la programación competitiva. Sin embargo, un programador competitivo debe ser capaz de resolver, en un concurso, problemas diferentes a los llamados *ad hoc*. Esperamos que disfrutes del viaje y alimentes tu entusiasmo leyendo y aprendiendo nuevos conceptos en los *otros* capítulos de este libro. Una vez que hayas terminado de leer todo el texto, hazlo una vez más desde el principio. En esa segunda vez, intenta resolver los  $\approx 238$  ejercicios escritos y los  $\approx 1675$  ejercicios de programación.



Figura 1.4: Algunas de las fuentes que han inspirado a los autores de este libro

## Capítulo 2

---

# Estructuras de datos y bibliotecas

*Si he logrado ver más allá, ha sido únicamente gracias a mirar desde los hombros de gigantes.*

— Isaac Newton

## 2.1 Introducción y motivación

Una estructura de datos (DS) es un mecanismo de almacenamiento y organización de datos. Cada tipo de estructura de datos tiene sus fortalezas y sus debilidades, por lo que, al diseñar un algoritmo, es importante elegir aquella que permita inserciones, búsquedas, eliminaciones y/o actualizaciones eficientes, dependiendo de las necesidades concretas que se presenten. Aunque una estructura de datos no resolverá por sí misma un problema (de un concurso de programación), utilizar la que resulte más eficiente puede marcar la diferencia entre cumplir con el límite de tiempo establecido o superarlo. Existen muchas formas de organizar unos datos determinados y, en ocasiones, unas son mejores que otras, dependiendo del contexto. Podremos comprobarlo en numerosas ocasiones en el presente capítulo. Familiarizarse a fondo con las estructuras de datos y las bibliotecas tratadas en este capítulo, resultará fundamental para entender los algoritmos que las utilizarán en el resto del libro.

Como se decía en el prefacio, **asumimos** que estás *familiarizado* con las estructuras de datos básicas que aparecen en las secciones 2.2 y 2.3, por lo que **no** las estudiaremos en este libro. En su lugar, pondremos de relieve el hecho de que existen implementaciones integradas de estas estructuras de datos elementales en la STL de C++ y en el API de Java<sup>1</sup>. Si no te sientes familiarizado con alguno de los términos o estructuras de datos mencionados en las secciones 2.2 y 2.3, te recomendamos que refresques esos conceptos en particular en alguno de los libros de referencia<sup>2</sup> que las tratan, incluyendo clásicos como “Introduction to Algorithms” [7], “Data Abstraction and Problem Solving” [5, 54], “Data Structures and Algorithms” [12], etc. Continúa leyendo este libro solo cuando entiendas, al menos, los *conceptos básicos* detrás de las estructuras de datos.

---

<sup>1</sup>Aunque en este texto utilizamos principalmente código C++ para ilustrar las técnicas de implementación, se pueden encontrar los códigos Java equivalentes en la página web del libro.

<sup>2</sup>Las materias de las secciones 2.2 y 2.3 se estudian, normalmente, en el primer o segundo año de ingeniería informática. Los estudiantes de secundaria que deseen participar en la IOI, deberían estudiar de forma independiente dichas materias.

Hay que tener en cuenta que, en la programación competitiva, basta con conocer las estructuras de datos solo hasta el punto de ser capaz de elegir y *utilizar* la que sea más adecuada para cada situación. Deberías entender las fortalezas, debilidades y complejidades de tiempo/espacio de las más habituales. Aun cuando la teoría sobre la que se sustentan las estructuras de datos puede resultar una lectura interesante, su conocimiento no resulta imprescindible, ya que las bibliotecas integradas en los lenguajes proporcionan implementaciones fiables y listas para usar de las mismas. Aunque ésta *no sea* la mejor aproximación al tema, descubriremos que, normalmente, es suficiente. Muchos concursantes (jóvenes) son capaces de utilizar, eficientemente (con complejidad de  $O(\log n)$  en la mayoría de las operaciones), la implementación de `map` de la STL de C++ (o `TreeMap` en Java), para almacenar colecciones dinámicas de parejas ‘clave-dato’ sin entender que la estructura de datos subyacente es un *árbol de búsqueda binaria equilibrado*, o utilizan `priority_queue` de la STL de C++ (o `PriorityQueue` de Java) para ordenar una cola de elementos, sin entender que la estructura que lo sustenta es un *montículo (normalmente binario)*. Ambas estructuras de datos se estudian, normalmente, en el primer (o segundo) año de ingeniería informática.

Este capítulo se divide en tres partes. La sección 2.2 contiene estructuras de datos *lineales* básicas y las operaciones más elementales que soportan. La sección 2.3 cubre estructuras de datos *no lineales* básicas, como árboles de búsqueda binaria (BST) equilibrados, montículos (binarios) y tablas *hash*, así como sus operaciones más importantes. La discusión de cada estructura de datos en las secciones 2.2 y 2.3 es breve, poniendo el acento en las *rutinas de la biblioteca* más importantes para su manipulación. Sin embargo, una estructura de datos especial, que es común en los concursos de programación, es la máscara de bits (y sus diversas técnicas de manipulación de bits, ver la figura 2.1), que será tratada en más detalle, dada la importancia de sus aplicaciones en el mundo de la programación competitiva. La sección 2.4 contiene una explicación más extensa que la que podemos encontrar en las secciones 2.2 y 2.3.

### Características de valor añadido de este libro

Como este capítulo es el primero que se sumerge en el núcleo de la programación competitiva, aprovechamos la oportunidad para poner de manifiesto algunas de las características de valor añadido de este libro, que podrás ver en este y en los siguientes capítulos.

Una característica fundamental de este libro es la colección de *ejemplos de código eficientes y totalmente implementados* que le acompaña, tanto en C/C++ como en Java, algo de lo que adolecen tantos libros de ingeniería informática, que se detienen al nivel del pseudo-código en sus demostraciones de algoritmos y estructuras de datos. Esta característica se ha encontrado en este libro desde su primera edición. Las partes más importantes del código están impresas en el propio libro<sup>3</sup> y todos los códigos completos se encuentran disponibles en [sites.google.com/site/stevenhalim/home/material](http://sites.google.com/site/stevenhalim/home/material). La referencia a cada archivo de código aparece en el texto, indicado por los iconos que se muestran a continuación:



Otra fortaleza de este libro es la colección de ejercicios, tanto escritos como de programación (la mayoría de ellos por medio del UVa Online Judge [49] e integrados en uHunt, como se indica

<sup>3</sup>Sin embargo, hemos decidido no incluir el código de las secciones 2.2 y 2.3, porque resultaría trivial para la mayoría de los lectores excepto, quizás, en algunos trucos útiles.

en el apéndice A). En la presente edición, hemos añadido *muchos más* ejercicios escritos. Los hemos clasificado como *no resaltados* y *resaltados* (\*). Los no resaltados sirven, principalmente, para realizar autoevaluaciones, y sus soluciones se encuentran al final de cada capítulo. Los resaltados sirven como retos adicionales para los que no proporcionamos una solución y, en su lugar, damos algunas pistas útiles.

También en esta edición hemos añadido visualizaciones<sup>4</sup> para muchas de las estructuras de datos y algoritmos tratados en el texto [27]. Creemos que dichas visualizaciones proporcionarán un importante beneficio para aquellos lectores que prefieran el aprendizaje visual. En este momento, las visualizaciones se alojan en [visualgo.net](http://visualgo.net). La referencia de cada visualización viene incluida en el texto junto a un ícono como el que se muestra a continuación:



## 2.2 Estructuras de datos lineales con bibliotecas integradas

Una estructura de datos se considera *lineal* si sus elementos forman una secuencia lineal, es decir, están colocados de izquierda a derecha (o de arriba a abajo). Dominar las siguientes estructuras de datos lineales básicas es imprescindible en los concursos de programación actuales:

- *Array* estático (soporte nativo en C/C++ y en Java).

Sin lugar a dudas, es la estructura de datos más utilizada en los concursos de programación. Siempre que tenemos que almacenar una colección de datos secuenciales para, posteriormente, acceder a ellos utilizando sus *índices*, el *array* estático será la forma más natural de hacerlo. Como el tamaño máximo de la entrada suele estar mencionado en el enunciado del problema, es posible declarar el tamaño del *array* para que lo albergue completamente, con un pequeño espacio adicional (centinela) de seguridad, para evitar veredictos RTE innecesarios, por haber calculado mal la dimensión inicial. Normalmente, en los concursos de programación se utilizan *arrays* de una, dos o tres dimensiones, y es raro que sean necesarias dimensiones adicionales. Las operaciones típicas con un *array* incluyen acceder a sus elementos por medio del índice, ordenar los elementos, y realizar barridos lineales o búsquedas binarias en un *array* ordenado.

- *Array* dinámico: *vector* en la STL de C++ (*ArrayList* (más rápido) o *Vector* en Java).

Esta estructura de datos es similar al *array* estático, con la excepción de que está diseñado de forma nativa para ser redimensionable en tiempo de ejecución. Es mejor utilizar un *vector*, frente a un *array* estático, siempre que el tamaño de la secuencia de elementos sea desconocida en el momento de la implementación. Normalmente, inicializaremos el tamaño (*reserve()* o *resize()*) con una estimación de las necesidades, para obtener un mejor rendimiento. Las operaciones típicas de la STL de C++ con *vector*, que se aplican a la programación dinámica, incluyen *push\_back()*, *at()*, el operador *[]*, *assign()*, *clear()*, *erase()* e *iterator*, para recorrer el contenido de un *vector*.

<sup>4</sup>VisuAlgo está desarrollado con tecnologías web modernas, como canvas de HTML5, SVG, CSS3, JavaScript, la biblioteca D3.js, etc.



ch2\_01\_array\_vector.cpp



ch2\_01\_array\_vector.java

Es interesante, en este punto, detenernos a estudiar dos operaciones realizadas habitualmente sobre *arrays*: **ordenación** y **búsqueda**. Ambas operaciones están bien soportadas en C++ y Java.

Hay *muchos* algoritmos de ordenación mencionados en libros de ingeniería informática [7, 5, 54, 12, 40, 58], por ejemplo:

1. Algoritmos de ordenación basados en la comparación, con complejidad  $O(n^2)$ : burbuja, selección, inserción, etc. Estos algoritmos son (terriblemente) lentos y, normalmente, se deben evitar en los concursos de programación, aunque su comprensión puede ser útil para resolver ciertos problemas.
2. Algoritmos de ordenación basados en la comparación, con complejidad  $O(n \log n)$ : mezcla, montículo, *quick*, etc. Estos algoritmos son la elección natural en concursos de programación, ya que la complejidad  $O(n \log n)$  resulta óptima para la ordenación basada en comparación. Por lo tanto, se ejecutan consumiendo el ‘menor tiempo posible’ en la mayoría de los casos (después trataremos algunos algoritmos de ordenación para casos especiales). Además, estos algoritmos son bien conocidos y, por ello, no es necesario ‘reinventar la rueda’<sup>5</sup>, bastará con utilizar `sort`, `partial_sort` o `stable_sort` en `algorithm` de la STL de C++ (o `Collections.sort` en Java), para realizar tareas normales de ordenación. Es suficiente con especificar la función de comparación requerida y las rutinas de la biblioteca se ocuparán del resto.
3. Algoritmos de ordenación específicos, con complejidad  $O(n)$ : cuentas, *radix*, casilleros, etc. Aunque utilizados en pocas ocasiones, es importante conocer estos algoritmos de uso especial, ya que pueden reducir el tiempo de ordenación, si los datos cuentan con determinadas características especiales. Por ejemplo, la ordenación por cuentas se puede aplicar a un pequeño rango de números enteros (ver la sección 9.32).

Hay tres métodos habituales para buscar un elemento en un *array*:

1. Búsqueda lineal con complejidad  $O(n)$ : considera cada elemento desde el índice 0 hasta el índice  $n - 1$  (evitar siempre que sea posible).
2. Búsqueda binaria con complejidad  $O(\log n)$ : en la STL de C++, utilizar `lower_bound`, `upper_bound` o `binary_search` (o `Collections.binarySearch` en Java). Si el *array* de entrada está desordenado, será necesario ordenarlo al menos una vez (utilizando uno de los algoritmos  $O(n \log n)$  mostrados antes) antes de ejecutar una, o *varias*, búsquedas binarias.
3. *Hashes* con complejidad  $O(1)$ : esta técnica resulta útil cuando se requiere un acceso rápido a valores conocidos. Si se ha seleccionado una función *hash* adecuada, la probabilidad de que se produzca una colisión es mínima. Aun así, se utiliza en raras ocasiones, y podríamos prescindir de ella totalmente<sup>6</sup> en la mayoría de los problemas.

<sup>5</sup>Sin embargo, a veces sí que es necesario ‘reinventar la rueda’ en ciertos problemas sobre ordenación, por ejemplo en el problema de índice de inversión de la sección 9.14.

<sup>6</sup>En las entrevistas de trabajo suelen aparecer preguntas sobre *hashing*.



ch2\_02\_algorithm\_collections.cpp



ch2\_02\_algorithm\_collections.java

- *Array* de booleanos: `bitset` en la STL de C++ (`BitSet` en Java).

Si nuestro *array* va a contener, únicamente, valores booleanos (1/verdadero y 0/falso), podemos utilizar una estructura de datos alternativa: el `bitset` de la STL de C++. El `bitset` admite operaciones tan útiles como `reset()`, `set()`, el operador `[]` y `test()`.



ch5\_06\_primes.cpp



ch5\_06\_primes.java

**ver también la sección 5.5.1**

- Máscaras de bits o pequeños conjuntos de booleanos (soporte nativo en C/C++/Java).

Un número entero se almacena en la memoria de un ordenador como una secuencia de bits. Por lo tanto, podemos utilizar enteros para representar un pequeño conjunto de valores booleanos. Todas las operaciones se realizarán a través de la manipulación de los bits del entero correspondiente, lo que supone una elección *mucho más eficiente*, en comparación a las opciones `vector<bool>`, `bitset` o `set<int>` de la STL de C++. La diferencia de velocidad es importante en la programación competitiva. A continuación, se muestran *algunas* de las operaciones más importantes que aparecerán en este libro.

```
Comprobar si el bit j (desde la derecha) de S está activo
{F D B } (activo)
S=42 (dec) = 101010 (bin)
j=3, 1<j=8 (dec) = 001000 (bin)
----- (AND)
T=8 (dec) = 001000 (bin)
{ D } (activo)
```

Figura 2.1: Visualización de máscaras de bits

1. Representación: un entero *con signo* de 32 (o 64) bits nos sirve para representar hasta 32 (o 64) elementos<sup>7</sup>. Sin perder el ámbito generalista, todos los ejemplos mostrados a continuación utilizan un entero con signo de 32 bits, llamado *S*.

Ejemplo:            5| 4| 3| 2| 1| 0 <- índice 0 desde la derecha

                      32|16| 8| 4| 2| 1 <- potencia de 2

*S* = 34 (base 10) = 1| 0| 0| 0| 1| 0 (base 2)

F| E| D| C| B| A <- etiqueta alfabética alternativa

<sup>7</sup>Para evitar problemas con la representación complementaria de los doses, se deberían utilizar los enteros de 32 o 64 bits solo para máscaras de bits de 30 o 62 elementos, respectivamente.

En el ejemplo anterior, el entero  $S = 34$ , o  $100010$  en binario, también representa un pequeño conjunto  $\{1, 5\}$  con un esquema de indexación basado en el 0, según se incrementa la significancia del dígito (o  $\{B, F\}$ , utilizando la etiqueta alfabética alternativa), ya que los bits segundo y sexto (desde la derecha) de  $S$  están activados.

- Para multiplicar, o dividir, un entero por 2, basta con desplazar los bits hacia la izquierda, o la derecha, respectivamente. Esta operación (especialmente el desplazamiento a la izquierda) resultará importante en los siguientes ejemplos. Hay que tener en cuenta que el desplazamiento a la derecha provoca un truncado que supone el redondeo automático a la baja en la división, por ejemplo,  $17/2 = 8$ .

```

S = 34 (base 10) = 100010 (base 2)
S = S<<1 = S*2 = 68 (base 10) = 1000100 (base 2)
S = S>>2 = S/4 = 17 (base 10) = 10001 (base 2)
S = S>>1 = S/2 = 8 (base 10) = 1000 (base 2) <- LSB desaparece
 (LSB = bit menos significativo)

```

- Para activar el elemento  $j$ -ésimo (con una indexación basada en 0) en el conjunto, utilizamos la operación de bits OR  $S |= (1 << j)$ .

```

S = 34 (base 10) = 100010 (base 2)
j = 3, 1<<j = 001000 <- el bit '1' se mueve a la izquierda 3 veces
 ----- OR (cierto si cualquiera de los bits lo es)
S = 42 (base 10) = 101010 (base 2) // actualiza S al nuevo valor 42

```

- Para comprobar si el elemento  $j$ -ésimo está activado, utilizamos la operación de bits AND  $T = S \& (1 << j)$ . Si  $T = 0$ , entonces el elemento  $j$ -ésimo está desactivado. Si  $T != 0$  (para ser exactos,  $T = (1 << j)$ ), entonces el elemento  $j$ -ésimo está activado. En la figura 2.1 hay un ejemplo de esto.

```

S = 42 (base 10) = 101010 (base 2)
j = 3, 1<<j = 001000 <- el bit '1' se mueve a la izquierda 3 veces
 ----- AND (cierto si los dos bits lo son)
T = 8 (base 10) = 001000 (base 2) -> no es 0, tercer elemento activado

S = 42 (base 10) = 101010 (base 2)
j = 2, 1<<j = 000100 <- el bit '1' se mueve a la izquierda 2 veces
 ----- AND
T = 0 (base 10) = 000000 (base 2) -> 0, segundo elemento desactivado

```

- Para limpiar/desactivar el elemento  $j$ -ésimo del conjunto, utilizamos<sup>8</sup> la operación de bits AND  $S \&= \sim(1 << j)$ .

```

S = 42 (base 10) = 101010 (base 2)
j = 1, \sim(1<<j) = 111101 <- '\sim' es el operador de bits NOT
 ----- AND
S = 40 (base 10) = 101000 (base 2) // actualiza S al nuevo valor 40

```

- Para comutar (invertir el estado) el elemento  $j$ -ésimo del conjunto, utilizamos la operación de bits XOR  $S ^= (1 << j)$ .

---

<sup>8</sup>Usamos siempre paréntesis en las operaciones de manipulación de bits, para evitar errores accidentales debido a la precedencia de operadores.

```

S = 40 (base 10) = 101000 (base 2)
j = 2, (1<<j) = 000100 <- el bit '1' se mueve a la izquierda 2 veces
----- XOR <- cierto si ambos bits son diferentes
S = 44 (base 10) = 101100 (base 2) // actualiza S al nuevo valor 44

S = 40 (base 10) = 101000 (base 2)
j = 3, (1<<j) = 001000 <- el bit '1' se mueve a la izquierda 3 veces
----- XOR <- cierto si ambos bits son diferentes
S = 32 (base 10) = 100000 (base 2) // actualiza S al nuevo valor 32

```

7. Para obtener el valor del bit menos significativo de S que está activado (el primero por la derecha), utiliza  $T = (S \& (-S))$ .

```

S = 40 (base 10) = 000...000101000 (32 bits, base 2)
-S = -40 (base 10) = 111...111011000 (complemento de dos)
----- AND
T = 8 (base 10) = 000...000001000 (el tercer bit está activado)

```

8. Para activar *todos* los bits en un conjunto de tamaño  $n$ , utiliza  $S = (1 \ll n) - 1$  (cuidado con los desbordamientos).

Ejemplo para  $n = 3$

```

S+1 = 8 (base 10) = 1000 <- el bit '1' se mueve a la izquierda 3 veces
 1

 -
S = 7 (base 10) = 111 (base 2)

```

Ejemplo para  $n = 5$

```

S+1 = 32 (base 10) = 100000 <- el bit '1' a la izquierda 5 veces
 1

 -
S = 31 (base 10) = 11111 (base 2)

```

9. Para contar cuántos bits están activados en S, podemos usar

```
_builtin_popcount(S) <- integrado en el compilador GNU C++
```



**ch2\_03\_bit\_manipulation.cpp**



**ch2\_03\_bit\_manipulation.java**

Muchas operaciones de manipulación de bits están escritas como macros del preprocesador de C/C++ en nuestro código de ejemplo (pero están escritas directamente en el código Java, ya que este lenguaje no admite macros).

- Lista enlazada: `list` en la STL de C++ (`LinkedList` en Java).

Aunque esta estructura de datos aparece casi siempre en los libros de texto sobre estructuras de datos y algoritmos, la lista enlazada se suele evitar en los problemas más típicos. Esto es debido a la ineficiencia al acceder a los elementos (se debe realizar un barrido

lineal del principio al final de la lista), y el uso de punteros puede producir errores en tiempo de ejecución, si no se implementa correctamente. Casi todas las formas de lista enlazada que aparecen en este libro han sido sustituidas por el `vector` de la STL de C++ (`Vector` en Java), mucho más flexible.

La única excepción la encontramos, probablemente, en el problema UVa 11988 - Broken Keyboard (a.k.a. Beiju Text), donde se requiere mantener dinámicamente una lista (enlazada) de caracteres, e insertar eficientemente un nuevo carácter en *cualquier lugar* de la lista, por ejemplo al principio (cabecera), posición actual o final (cola) de la lista (enlazada). De los 1903 problemas de UVa resueltos por los autores, este es, probablemente, el único problema de lista enlazada puro que hemos encontrado.

- Pila: `stack` en la STL de C++ (`Stack` en Java).

Esta estructura de datos se utiliza normalmente como parte de algoritmos que resuelven ciertos problemas (por ejemplo, emparejamiento de paréntesis en la sección 9.4, calculadora posfija y conversión de infija a posfija en la sección 9.27, encontrar componentes fuertemente conexos en la sección 4.2.9 y el barrido de Graham en la sección 7.3.7). Una pila solo permite la inserción (`push`), de complejidad  $O(1)$ , y la eliminación (`pop`), de complejidad  $O(1)$ , de su parte superior. Este comportamiento se conoce habitualmente como ‘último en entrar, primero en salir’ (LIFO) y guarda paralelismo con las pilas del mundo real. Las operaciones típicas de la `stack` de la STL de C++ incluyen `push()`/`pop()` (insertar/eliminar de la parte superior de la pila), `top()` (obtener contenido de la parte superior de la pila) y `empty()`.

- Cola: `queue` en las STL de C++ (`Queue` en Java<sup>9</sup>).

Esta estructura de datos se utiliza en algoritmos como búsqueda en anchura (BFS), en la sección 4.2.2. Una cola permite la inserción con complejidad  $O(1)$  en el final (cola), y la eliminación con complejidad  $O(1)$  en el principio (cabecera). Este comportamiento se conoce también como ‘primero en entrar, primero en salir’ (FIFO), como las colas del mundo real. Las operaciones típicas de la `queue` de la STL de C++ incluyen `push()`/`pop()` (insertar al final/eliminar del principio de la cola), `front()`/`back()` (obtener contenido del principio/final de la cola) y `empty()`.

- Cola de dos extremos (`deque`): `deque` en la STL de C++ (`Deque` en Java<sup>10</sup>).

Esta estructura de datos es muy similar al `array` redimensionable (`vector`) y cola, salvo que las colas de dos extremos soportan inserciones y eliminaciones rápidas, de complejidad  $O(1)$ , en ambos extremos de las mismas. Esta característica es importante para ciertos algoritmos, como, por ejemplo, los de ventanas correderas de la sección 9.31. Las operaciones típicas de la `deque` de la STL de C++ incluyen `push_back()`, `pop_front()` (como en una cola normal), `push_front()` y `pop_back()` (específicos para la `deque`).



ch2\_04\_stack\_queue.cpp



ch2\_04\_stack\_queue.java

<sup>9</sup>Queue en Java es solo un *interfaz* que, normalmente, se instancia con una `LinkedList`.

<sup>10</sup>Deque en Java también es un *interfaz* que, normalmente, se instancia con una `LinkedList`.

### Ejercicio 2.2.1\*

Supongamos que tienes un *array sin ordenar*  $S$  de  $n$  enteros con signo de 32 bits. Resuelve cada una de las tareas que se indican a continuación, con el mejor algoritmo posible que se te ocurra, y analiza su complejidad de tiempo. Vamos a asumir los siguientes límites:  $1 \leq n \leq 100K$ , lo que supone que las soluciones  $O(n^2)$  serían, teóricamente, impracticables en el entorno de un concurso.

1. Determina si  $S$  contiene una o más parejas de enteros duplicados.
- 2\*. Dado un entero  $v$ , busca dos enteros  $a, b \in S$ , de forma que  $a + b = v$ .
- 3\*. Siguiendo con la tarea 2: ¿qué ocurre si el *array S ya está ordenado*?
- 4\*. Escribe ordenados los enteros en  $S$  que estén en un rango  $[a \dots b]$  (ambos inclusive).
- 5\*. Determina la longitud del *subarray* más largo incremental y *contiguo* en  $S$ .
6. Determina la mediana (el percentil 50) de  $S$ . Se asume que  $n$  es impar.

### Ejercicio 2.2.2

Existen otros trucos posibles en las técnicas de manipulación de bits, pero raramente se utilizan. Implementa estas tareas con manipulación de bits:

1. Obtén el resto (módulo) de  $S$  cuando es dividido por  $N$  ( $N$  es potencia de 2), por ejemplo,  $S = (7)_{10} \% (4)_{10} = (111)_2 \% (100)_2 = (11)_2 = (3)_{10}$ .
2. Determina si  $S$  es una potencia de 2, por ejemplo,  $S = (7)_{10} = (111)_2$  no es potencia de 2, pero  $(8)_{10} = (1000)_2$  sí que lo es.
3. Desactiva el último bit de  $S$ , por ejemplo,  $S = (40)_{10} = (10\underline{1}000)_2 \rightarrow S = (32)_{10} = (100\underline{0}000)_2$ .
4. Activa el último cero de  $S$ , por ejemplo,  $S = (41)_{10} = (1010\underline{0}1)_2 \rightarrow S = (43)_{10} = (1010\underline{1}1)_2$ .
5. Desactiva la última aparición consecutiva de unos en  $S$ , por ejemplo,  $S = (39)_{10} = (100\underline{1}11)_2 \rightarrow S = (32)_{10} = (100\underline{0}00)_2$ .
6. Activa la última aparición consecutiva de ceros en  $S$ , por ejemplo,  $S = (36)_{10} = (1001\underline{0}0)_2 \rightarrow S = (39)_{10} = (1001\underline{1}1)_2$ .
- 7\*. Resuelve el problema UVa 11173 - Grey Codes, con una expresión de manipulación de bits de *una línea de código* para cada caso de prueba, es decir, encuentra el código gris  $k$ -ésimo.
- 8\*. Vamos a invertir el problema UVa 1173 anterior. Dado un código gris, encuentra su posición  $k$ , utilizando manipulación de bits.

## Ejercicio 2.2.3\*

También podemos utilizar un *array redimensionable* (`vector` en la STL de C++ o `Vector` en Java) para implementar una pila eficiente. Descubre cómo hacerlo. Pregunta posterior: ¿es posible utilizar, en su lugar, un *array estático*, una lista enlazada o una cola de dos extremos? ¿Por qué o por qué no?

## Ejercicio 2.2.4\*

Podemos utilizar una lista enlazada (`list` en la STL de C++ o `LinkedList` en Java) para implementar una cola (o cola de dos extremos) eficiente. Descubre cómo hacerlo. Pregunta posterior: ¿es posible utilizar un *array redimensionable* en su lugar? ¿Por qué o por qué no?

## Ejercicios de programación

### Ejercicios de programación sobre estructuras de datos lineales con bibliotecas:

#### Manipulación de arrays 1D: *array*, `vector` en la STL de C++ (o `Vector`/`ArrayList` en Java)

1. UVa 00230 - *Borrowers* (procesamiento de cadenas; ordenar los libros por autor y, después, por título; tamaño de la entrada pequeño; no es necesario un BST equilibrado)
2. UVa 00394 - *Mapmaker* (cualquier *array* de  $n$  dimensiones se almacena en memoria como unidimensional; seguir la descripción del problema)
3. UVa 00414 - *Machined Surfaces* (obtener la secuencia más larga de Bs)
4. UVa 00467 - *Synching Signals* (barido lineal; etiqueta booleana 1D)
5. UVa 00482 - *Permutation Arrays* (puede requerir trocear cadenas porque no se indica el tamaño del *array*)
6. UVa 00591 - *Box of Bricks* (sumar todos los elementos; obtener la media; sumar el total de diferencias absolutas de cada elemento con la media dividida por dos)
7. UVa 00665 - *False Coin* (usar etiquetas booleanas 1D; cada '=','<' o '>' nos proporciona información; comprobar si, al final, solo queda una moneda falsa posible)
8. UVa 00755 - 487-3279 (tabla de direccionamiento directo; convertir las letras, salvo Q y Z, a 2-9; mantener 0-9 como 0-9; ordenar los enteros; encontrar duplicados, si hay)
9. **UVa 10038 - *Jolly Jumpers*\*** (etiquetas booleanas 1D para comprobar  $[1..n-1]$ )
10. UVa 10050 - *Hartals* (etiqueta booleana 1D)
11. UVa 10260 - *Soundex* (tabla de direccionamiento directo para mapear el código *soundex*)
12. UVa 10978 - *Let's Play Magic* (manipulación de *array* de cadenas 1D)
13. UVa 11093 - *Just Finish it up* (barido lineal; *array* circular; un poco complicado)
14. UVa 11192 - *Group Reverse* (*array* de caracteres)
15. UVa 11222 - *Only I did it* (usar varios *arrays* 1D para simplificar el problema)
16. **UVa 11340 - *Newspaper*\*** (tabla de direccionamiento directo)
17. UVa 11496 - *Musical Loop* (almacenar información en *array* 1D; contar los picos)
18. UVa 11608 - *No Problem* (usar tres *arrays*: creados, requeridos, disponibles)

19. UVa 11850 - Alaska  
 20. UVa 12150 - Pole Position  
**21. UVa 12356 - Army Buddies \***
- (para cada ubicación entera de 0 a 1322, ¿puede llegar Brenda a (menos de 200 millas de) cualquier estación de carga?)  
 (manipulación sencilla)  
 (similar a la eliminación en listas enlazadas dobles, pero se puede usar un array 1D para la estructura de datos subyacente)

### Manipulación de *arrays* bidimensionales

1. UVa 00101 - The Blocks Problem  
 (simulación de una pila; pero necesitamos acceder al contenido de cada pila, así que es mejor usar un *array* bidimensional)
2. UVa 00434 - Matty's Blocks  
 (un tipo de problema de visibilidad en geometría; se resuelve con manipulación de un *array* bidimensional)  
 (funciones principales: rotar y reflejar)
3. UVa 00466 - Mirror Mirror  
 (contar el número de unos en cada fila/columna, que debe ser par; si hay error, ver si el número impar de unos aparece en la misma fila y columna)  
 (tedioso)
4. UVa 00541 - Error Correction  
 (usar un *array* booleano bidimensional de tamaño  $500 \times 500$ )  
 (array de cadenas; rotación de 90 grados a la derecha)
5. UVa 10016 - Flip-flop the Squarelotron  
 (simular el proceso)  
 (manipulación de *array* bidimensional no trivial)
6. UVa 10703 - Free spots  
 (usar `long long` para evitar problemas)
- 7. UVa 10855 - Rotated squares \***  
 (hacer lo que se pide)
- 8. UVa 10920 - Spiral Tap \***  
 (hacer lo que se pide)
9. UVa 11040 - Add bricks in the wall  
 (simular el proceso)
10. UVa 11349 - Symmetric Matrix  
 (hacer lo que se pide)
11. UVa 11360 - Have Fun with Matrices  
 (hacer lo que se pide)
- 12. UVa 11581 - Grid Successors \***  
 (hacer lo que se pide; un poco tedioso)
13. UVa 11835 - Formula 1  
 (simular hacia atrás; no olvidar hacer `mod 10`)
14. UVa 12187 - Brothers  
 (simular el proceso)
15. UVa 12291 - Polyomino Composer  
 (hacer lo que se pide)
16. UVa 12398 - NumPuzz I

### algoritm de la STL de C++ (Collections en Java)

1. UVa 00123 - Searching Quickly  
 (función de comparación modificada; usar `sort`)
- 2. UVa 00146 - ID Codes \***  
 (usar `next_permutation`)  
 (este comando se utiliza mucho en UNIX)
3. UVa 00400 - Unix ls  
 (problema de ordenación tedioso)
4. UVa 00450 - Little Black Book  
 (ordenación por campos múltiples; usar `sort`; similar a UVa 10258)
5. UVa 00790 - Head Judge Headache  
 (ordenación; mediana)
6. UVa 00855 - Lunch in Grid City  
 (LA 3173 - Manila06; `next` y `prev_permutation` de la STL)
7. UVa 01209 - Wordfish  
 (implica la mediana; usar `sort`, `upper_bound`, `lower_bound` de la STL y algunas comprobaciones)
8. UVa 10057 - A mid-summer night ...  
 (encontrar la mediana de una lista de enteros *creciente/dinámica*; se resuelve con llamadas múltiples a `nth_element` en `algorithm`)
9. **UVa 10107 - What is the Median? \***  
 (ordenación por campos múltiples; usar `sort`)
10. UVa 10194 - Football a.k.a. Soccer  
 (ordenación por campos múltiples; usar `sort`; similar a UVa 790)
- 11. UVa 10258 - Contest Scoreboard \***  
 (ordenación por campos múltiples; usar `sort`)  
 (usar `sort`)
12. UVa 10698 - Football Sort  
 (función de comparación modificada; usar `sort`)
13. UVa 10880 - Colin and Ryan  
 (usar `sort` y contar los signos diferentes)
14. UVa 10905 - Children's Game
15. UVa 11039 - Building Designing

- |                                                                                                                                                                                 |                                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 16. UVa 11321 - Sort Sort and Sort<br>17. UVa 11588 - Image Coding<br>18. UVa 11777 - Automate the Grades<br>19. UVa 11824 - A Minimum Land Price<br>20. UVa 12541 - Birthdates | (cuidado con el módulo negativo)<br>(sort simplifica el problema)<br>(sort simplifica el problema)<br>(sort simplifica el problema)<br>(LA 6148 - HatYai12; sort; elegir al más joven y al más viejo) |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Manipulación de bits (bitset de la STL de C++ (BitSet en Java) y máscara de bits)

- |                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00594 - One Little, Two Little ...<br>2. UVa 00700 - Date Bugs<br>3. UVa 01241 - Jollybee Tournament<br><b>4. UVa 10264 - The Most Potent Corner *</b><br>5. UVa 11173 - Grey Codes<br>6. UVa 11760 - Brother Arif, ...<br><b>7. UVa 11926 - Multitasking *</b><br><b>8. UVa 11933 - Splitting Numbers *</b><br>9. IOI 2011 - Pigeons | (manipular la cadena de bits con bitset)<br>(se resuelve con bitset)<br>(LA 4147 - Jakarta08; fácil)<br>(manipulación de máscara de bits intensa)<br>(patrón de divide y vencerás o manipulación de bits en una línea)<br>(separar las comprobaciones fila y columna; usar dos bitset)<br>(usar bitset de 1M para comprobar si un hueco está libre)<br>(ejercicio de manipulación de bits)<br>(este problema es más sencillo con manipulación de bits, pero la solución final requiere mucho más que eso) |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### C++ STL list (Java LinkedList)

- |                                             |                                        |
|---------------------------------------------|----------------------------------------|
| <b>1. UVa 11988 - Broken Keyboard ... *</b> | (problema de listas enlazadas atípico) |
|---------------------------------------------|----------------------------------------|

### C++ STL stack (Java Stack)

- |                                                                                                                                                                                               |                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00127 - "Accordian" Patience<br><b>2. UVa 00514 - Rails *</b><br><b>3. UVa 00732 - Anagram by Stack *</b><br><b>4. UVa 01062 - Containers *</b><br>5. UVa 10858 - Unique Factorization | (mezclar un stack)<br>(usar stack para simular el proceso)<br>(usar stack para simular el proceso)<br>(LA 3752 - WorldFinals Tokyo07; simulación con stack; la respuesta máxima es 26 pilas; existe una solución $O(n)$ )<br>(usar stack) |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Ver también: stack implícitos en llamadas recursivas a funciones y conversión/evaluación posfija en la sección 9.27.

### queue y deque de la STL de C++ (Queue y Deque en Java)

- |                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00540 - Team Queue<br><b>2. UVa 10172 - The Lonesome Cargo ... *</b><br><b>3. UVa 10901 - Ferry Loading III *</b><br>4. UVa 10935 - Throwing cards away I<br><b>5. UVa 11034 - Ferry Loading IV *</b><br>6. UVa 12100 - Printer Queue<br>7. UVa 12207 - This is Your Queue | ('cola' modificada)<br>(usar tanto queue como stack)<br>(simulación con queue)<br>(simulación con queue)<br>(simulación con queue)<br>(simulación con queue)<br>(usar tanto queue como deque) |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Ver también: queue en BFS (ver secciones 4.2.2 y 8.2.3).

## 2.3 Estructuras de datos no lineales con bibliotecas integradas

El almacenamiento lineal no es la mejor forma de organizar los datos en algunos problemas. Resulta más rápido operar con las implementaciones eficientes de estructuras de datos no lineales que aparecen a continuación, lo que acelera los algoritmos que dependen de ellas.

Por ejemplo, si necesitas una colección *dinámica*<sup>11</sup> de parejas (como “clave → valor”), utilizar `map` de la STL de C++ proporcionará un rendimiento de complejidad  $O(\log n)$  en las operaciones de inserción, búsqueda y eliminación, empleando unas pocas líneas de código (que tendrás que escribir), mientras que almacenar la misma información en un *array* estático de `struct` puede requerir una complejidad  $O(n)$ , y el código a implementar será más largo.

- Árbol de búsqueda binaria equilibrado (BST): `map/set` en STL (Java `TreeMap/TreeSet`)

El BST supone una forma de organización de datos en un estructura de árbol. En cada subárbol con raíz en  $x$  se mantiene la siguiente propiedad: los elementos del subárbol izquierdo de  $x$  son menores que  $x$ , y los elementos del subárbol derecho de  $x$  son mayores (o iguales) que  $x$ . Esto es, en esencia, una aplicación de la estrategia del “divide y vencerás” (ver también la sección 3.3). Organizar los datos de esta forma (ver la figura 2.2), permite realizar las operaciones `search(clave)`, `insert(clave)`, `findMin()/findMax()`, `successor(clave)/predecessor(clave)` y `delete(clave)` en  $O(\log n)$  ya que, en el peor de los casos, bastarán  $O(\log n)$  para realizar un barrido desde la raíz a las hojas (para más información, consultar [7, 5, 54, 12]). Sin embargo, esta afirmación solo es válida si el BST está equilibrado.

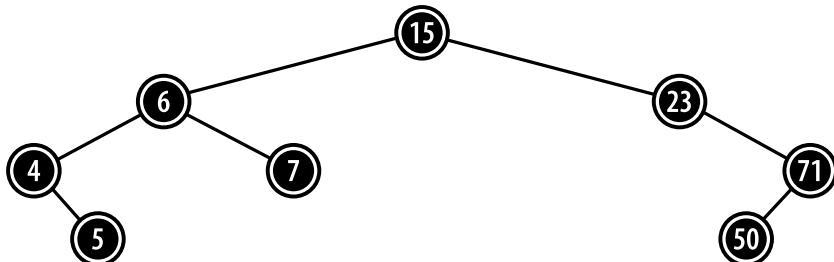


Figura 2.2: Ejemplo de BST

La implementación *libre de errores* de un BST equilibrado, como el Adelson-Velskii Landis (AVL)<sup>12</sup> o el Red-Black (RB)<sup>13</sup>, resulta una tarea tediosa y difícil de conseguir durante la duración de un concurso de programación (salvo que cuenten de antemano con una

<sup>11</sup>El contenido de una estructura de datos dinámica se modifica frecuentemente por medio de operaciones de inserción, eliminación y actualización.

<sup>12</sup>El árbol AVL fue el primer BST autoequilibrado que se inventó. Los árboles AVL consisten en, básicamente, un BST tradicional con una propiedad añadida: las alturas de los dos subárboles de cualquier vértice de un árbol AVL pueden tener una diferencia *máxima* de uno. Las operaciones de reequilibrio (rotaciones) se realizan (cuando es necesario) durante las inserciones y las eliminaciones, para mantener esta propiedad invariable, manteniendo así un árbol mayormente equilibrado.

<sup>13</sup>El árbol Red-Black es, además, un BST autoequilibrado, en el que cada vértice tiene un color: rojo o negro. En los árboles RB, el vértice raíz, los vértices hoja y todos los hijos de los vértices rojos son negros. Todo camino sencillo desde un vértice hasta cualquiera de sus hojas contiene *el mismo número de vértices negros*. A través de las inserciones y las eliminaciones, un árbol RB mantendrá estas características invariadas para conservar el equilibrio.

biblioteca de código, ver la sección 9.29). Por suerte, la STL de C++ tiene `map` y `set` (y Java tiene `TreeMap` y `TreeSet`), que son, *normalmente*, implementaciones del árbol RB, lo que garantiza que la mayoría de las operaciones del BST, como inserción, búsqueda o eliminación, se realizan en tiempo  $O(\log n)$ . Al dominar estas dos clases de la STL de C++ (o sus correspondientes en Java), puedes ahorrar mucho tiempo durante los concursos. La diferencia entre estas dos estructuras de datos es sencilla: el `map` de la STL de C++ (y el `TreeMap` de Java) almacena parejas (clave → valor), mientras que el `set` de la STL de C++ (y el `TreeSet` de Java) solo almacena la clave. En la mayoría de los problemas de los concursos utilizaremos `map` (para tener un auténtico mapa de la información) en vez de `set` (que solo resulta útil para determinar, de forma eficiente, la existencia de una clave concreta). Sin embargo, hay un pequeño inconveniente. Si utilizamos las implementaciones de las bibliotecas, resultará muy difícil o, incluso, imposible, aumentar (añadir información adicional) el BST. Intenta resolver el **ejercicio 2.3.5\*** y consulta la sección 9.29 para más información.



- Montículo: `priority_queue` en la STL de C++ (`PriorityQueue` en Java)

Un montículo es otra forma de organizar información en un árbol. El montículo (binario) es un árbol binario como el BST, con la diferencia de que debe ser un árbol completo<sup>14</sup>. Los árboles binarios completos se pueden almacenar, de forma eficiente, en un *array* compacto de un índice, de tamaño  $n + 1$ , lo que resulta preferible a una representación de un árbol completo. Por ejemplo, el *array*  $A = \{N/D, 90, 19, 36, 17, 3, 25, 1, 2, 7\}$  es la representación en un *array* compacto de la figura 2.3, ignorando el índice 0. Podemos desplazarnos desde un índice (vértice)  $i$  determinado a su padre, hermano izquierdo o hermano derecho, utilizando una manipulación de índices sencilla:  $\lfloor \frac{i}{2} \rfloor$ ,  $2 \times i$  y  $2 \times i + 1$ , respectivamente. Estas manipulaciones de los índices se pueden acelerar con técnicas de desplazamiento de bits (ver la sección 2.2):  $i >> 1$ ,  $i << 1$  e  $(i << 1) + 1$ , respectivamente.

En vez de forzar la propiedad de un BST, el montículo (máximo) se sustenta en la propiedad de montículo: en cada subárbol con raíz en  $x$ , los elementos a su izquierda y los subárboles a su derecha son menores (o iguales) que  $x$  (ver la figura 2.3). Esto resulta ser también una aplicación del concepto ‘divide y vencerás’ (ver la sección 3.3). Dicha propiedad garantiza que la cumbre (o raíz) del montículo es siempre el elemento máximo. En los montículos no existe el concepto de ‘búsqueda’ (a diferencia de en los BST). En su lugar, el montículo permite la extracción (eliminación) rápida del elemento máximo con `ExtractMax()` y la inserción de nuevos elementos con `Insert(v)`, pudiendo lograrse ambos fácilmente en  $O(\log n)$ , realizando un recorrido de la raíz a una hoja o de una hoja a la raíz, mediante operaciones de intercambio que mantengan la propiedad de montículo (máximo), siempre que sea necesario (para más información, consultar [7, 5, 54, 12]).

El montículo (máximo) es una estructura de datos muy útil en el modelado de una cola de prioridad, donde se puede extraer el elemento con mayor prioridad (el elemento máximo)

<sup>14</sup>Un árbol binario completo es aquel en el que cada nivel, con la posible excepción del último, está relleno completamente. También deben estar llenados todos los vértices del último nivel, de izquierda a derecha.

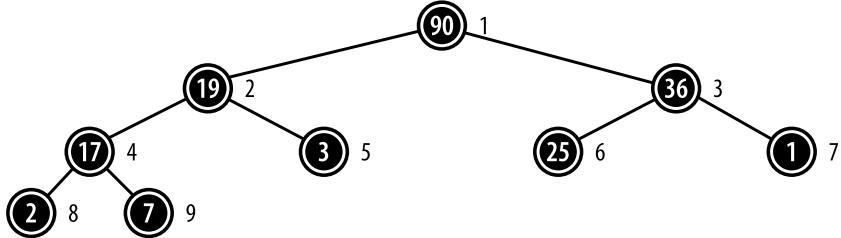


Figura 2.3: Visualización de un montículo (máximo)

mediante (`ExtractMax()`), y se puede añadir a la cola un nuevo elemento  $v$  mediante (`Insert(v)`), ambos en tiempo  $O(\log n)$ . La implementación<sup>15</sup> de `priority_queue` se encuentra en la biblioteca `queue` de la STL de C++ (o en `PriorityQueue` de Java). La cola de prioridad es un componente importante de algoritmos como los de Prim (y Kruskal), para el problema del árbol recubridor mínimo (MST) (ver la sección 4.3), Dijkstra para el problema del camino más corto de origen único (SSSP) (ver la sección 4.4.3) y la búsqueda A\* (ver la sección 8.2.5).

Esta estructura de datos tiene también uso para realizar el `partial_sort` de la biblioteca `algorithm` de la STL de C++. Una implementación posible consiste en procesar los elementos, de uno en uno, y crear un montículo máximo<sup>16</sup> de  $k$  elementos, eliminando el elemento mayor cuando su tamaño supere  $k$  ( $k$  es el número de elementos solicitado por el usuario). Se pueden obtener los  $k$  elementos más pequeños en orden descendente, al eliminar de la cola del montículo máximo los elementos restantes. Como cada operación de extracción de la cola supone  $O(\log k)$ , `partial_sort` tiene una complejidad de tiempo  $O(n \log k)$ <sup>17</sup>. Cuando  $k = n$ , este algoritmo es equivalente a una ordenación por montículos. Hay que tener en cuenta que, aunque la complejidad de tiempo de una ordenación por montículos es también  $O(n \log n)$ , éstas suelen ser más lentas que las ordenaciones *quick*, ya que las operaciones con montículos acceden a información almacenada en índices alejados y, por ello, no aprovechan bien la caché.



<sup>15</sup>La `priority_queue` predeterminada de la STL de C++ es un montículo máximo (la extracción de la cola toma los elementos en orden descendente de la clave), mientras que el `PriorityQueue` predeterminado de Java es un montículo mínimo (tomando los elementos en orden ascendente de la clave). Como consejo, cabe mencionar que un montículo máximo compuesto de números se puede transformar fácilmente en un montículo mínimo (y viceversa), convirtiendo los números en negativos. Esto se debe a que el orden de un conjunto de números se invertirá cuando sus valores se transforman en los negativos correspondientes. En este libro se pueden encontrar muchos ejemplos de este método. Sin embargo, si la cola de prioridad se utiliza para almacenar *enteros con signo de 32 bits*, se producirá un desbordamiento si se transforma  $-2^{31}$ , ya que  $2^{31} - 1$  es el valor máximo que puede alcanzar ese tipo de datos.

<sup>16</sup>El `partial_sort` predeterminado nos devuelve los  $k$  elementos más pequeños en orden ascendente.

<sup>17</sup>Quizá te hayas dado cuenta de que, en la complejidad  $O(n \log k)$ , esa  $k$  es el tamaño de la salida y  $n$  el de la entrada. Esto significa que el algoritmo es ‘sensible’ a la salida, ya que su tiempo de ejecución no depende únicamente de la entrada, sino también de los elementos que queremos obtener.

- Tabla *hash*: `unordered_map` en STL de C++11<sup>18</sup> (`HashMap`/`HashSet`/`HashTable` en Java)

La tabla de *hash* es otra estructura de datos no lineal, pero no recomendamos su uso en concursos de programación, salvo que sea estrictamente necesario. Diseñar una función de *hash* que proporcione buen rendimiento es complejo, y solo la STL del nuevo C++11 lo soporta (Java tiene clases relativas a los *hash*).

Además, los `map` y `set` de la STL de C++ (y los `TreeMap` o `TreeSet` de Java) son, normalmente, lo bastante rápidos, ya que el tamaño típico de la entrada de los problemas (de concursos de programación) no suele superar 1M. Dentro de esos límites, el rendimiento  $O(1)$  de una tabla de hash y el rendimiento  $O(\log 1M)$  de los BST equilibrados no se diferencia por mucho. Por ello, no trataremos las tablas de *hash* de forma detallada.

Sin embargo, sí existe un forma simplificada de las tablas de *hash* que sirven en concursos de programación. Se puede considerar que las ‘tablas de direccionamiento directo’ (DAT) son tablas de *hash*, cuando las propias claves son los índices, o cuando la ‘función de *hash*’ es la función de identidad. Por ejemplo, puede que necesitemos asignar todos los caracteres ASCII existentes [0-255] a valores enteros, como ‘a’ → ‘3’, ‘W’ → ‘10’, ..., ‘T’ → ‘13’. En este caso, no necesitaremos el `map` de la STL de C++, o cualquier otra forma de *hash*, pues la propia clave (el valor del carácter ASCII) es único y suficiente para determinar el índice correcto en un *array* de tamaño 256. En la sección 2.2, hemos incluido algunos ejercicios que contemplan el uso de DAT.

### Ejercicio 2.3.1

Alguien ha sugerido que es posible almacenar pares ‘clave → valor’ en un *array ordenado* de `struct`, de forma que podamos utilizar la búsqueda binaria en  $O(\log n)$  con el problema de ejemplo anterior. ¿Es esto posible? Si no, ¿qué lo impide?

### Ejercicio 2.3.2

En este libro no trataremos la parte más elemental de las operaciones con BST. En su lugar, utilizaremos una serie de subtareas para comprobar que entiendes los conceptos relacionados. Usaremos la figura 2.2 como *referencia inicial* en todas, excepto en la 2.

1. Muestra los pasos dados por `search(71)`, `search(7)` y `search(22)`.
2. Comenzando con un BST *vacío*, muestra los pasos dados por `insert(15)`, `insert(23)`, `insert(6)`, `insert(71)`, `insert(50)`, `insert(4)`, `insert(7)` e `insert(5)`, uno detrás de otro.
3. Muestra los pasos dados por `findMin()` (y `findMax()`).
4. Indica el *recorrido inorder* de este BST. ¿Está la salida ordenada?
5. Indica los recorridos *preorden*, *postorden* y *orden de nivel* de este BST.

<sup>18</sup>C++11 es un estándar más moderno de C++, los compiladores más antiguos podrían no soportarlo.

6. Muestra los pasos dados por `successor(23)`, `successor(7)` y `successor(71)`. Lo mismo para `predecessor(23)`, `predecessor(7)` y `predecessor(71)`.
7. Muestra los pasos dados por `delete(5)` (una hoja), `delete(71)` (un nodo interno con un hijo) y `delete(15)` (un nodo interno con dos hijos), uno detrás de otro.

### Ejercicio 2.3.3\*

Supón que tienes una referencia a la raíz  $R$  de un árbol binario  $T$ , que contiene  $n$  vértices. Puedes acceder a los vértices izquierdo, derecho y padre, así como a su clave, a través de su referencia. Resuelve cada una de las siguientes tareas con los mejores algoritmos que se te ocurran, y analiza sus complejidades de tiempo. Asumimos los siguientes límites:  $1 \leq n \leq 100K$ , por lo que las soluciones  $O(n^2)$  son, teóricamente, inviables en un concurso de programación.

1. Comprueba si  $T$  es un BST.
- 2\*. Muestra los elementos de  $T$  dentro de un rango  $[a..b]$ , en orden ascendente.
- 3\*. Muestra el contenido de las *hojas* de  $T$ , en *orden descendente*.

### Ejercicio 2.3.4\*

Se sabe que el recorrido inorden (ver también la sección 4.7.2) de un BST estándar (no necesariamente equilibrado) devuelve los elementos ordenados y se ejecuta en  $O(n)$ . ¿Devolverá también el siguiente código los elementos del BST ordenados? ¿Se puede conseguir que se ejecute en un tiempo total de  $O(n)$ , en vez de en  $O(\log n + (n-1) \times \log n) = O(n \log n)$ ? Y, si es así, ¿cómo?

```
x = findMin(); output x
for (i = 1; i < n; i++)
 x = successor(x); output x // ¿este bucle es O(n log n)?
```

### Ejercicio 2.3.5\*

Algunos problemas (difíciles) nos obligan a escribir *nuestra propia* implementación de un árbol de búsqueda binaria (BST) equilibrado, debido a la necesidad de aumentar el BST con datos adicionales (ver el capítulo 14 de [7]). El reto: resolver UVa 11849 - CD, que es un problema de BST equilibrado puro, con *tu propia* implementación de un BST equilibrado, para comprobar su rendimiento y validez.

### Ejercicio 2.3.6

En este libro no trataremos los conceptos elementales de operaciones con montículos. En su lugar, utilizaremos una serie de preguntas para comprobar tu comprensión de los conceptos de montículo.

1. Con la figura 2.3 como montículo inicial, muestra los pasos de `Insert(26)`.
2. Después de responder a la pregunta anterior, muestra los pasos de `ExtractMax()`.

### Ejercicio 2.3.7

¿Es la estructura representada por un *array* compacto basado en 1 (ignora el índice 0), ordenada de forma descendente, un montículo máximo?

### Ejercicio 2.3.8\*

Demuestra o rechaza esta afirmación: “el segundo elemento más grande en un montículo máximo, con  $n \geq 3$  elementos distintos, siempre es uno de los hijos directos de la raíz”. Siguiiente pregunta: ¿qué hay del tercer elemento más grande? ¿Cuáles son las ubicaciones potenciales del tercer elemento más grande en un montículo máximo?

### Ejercicio 2.3.9\*

Dado un *array* compacto, basado en 1,  $A$ , que contiene  $n$  enteros ( $1 \leq n \leq 100K$ ) con la garantía de que se satisface la propiedad de montículo máximo, mostrar los elementos de  $A$  que son mayores que un entero  $v$ . ¿Qué algoritmo es el mejor para ello?

### Ejercicio 2.3.10\*

Dado un *array* no ordenado  $S$  formado por  $n$  enteros distintos ( $2k \leq n \leq 100000$ ), encontrar los  $k$  enteros mayores y menores ( $1 \leq k \leq 32$ ) de  $S$  en  $O(n \log k)$ . Para este ejercicio, asumimos que un algoritmo  $O(n \log n)$  *no es* aceptable.

### Ejercicio 2.3.11\*

Una operación con montículos que *no está* integrada directamente en la `priority_queue` de la STL de C++ (ni en la `PriorityQueue` de Java) es `UpdateKey(indice, nuevaClave)`, que permite actualizar el elemento de un montículo (máximo) en un índice dado (incrementado o decrementado). Escribe *tu propia* implementación de un montículo (máximo) binario, incluyendo esta operación.

### Ejercicio 2.3.12\*

Otra operación con montículos que puede ser útil es `DeleteKey(indice)`, para eliminar elementos de un montículo (máximo) en un índice dado. Impleméntalo. ¿Qué casos debes tener ahora en consideración?

### Ejercicio 2.3.13\*

Supón que solo necesitamos la operación `DecreaseKey(indice, nuevaClave)`, es decir, una operación de `UpdateKey` donde la actualización *siempre* hace que `nuevaClave` sea menor que su valor anterior. ¿Podemos utilizar una técnica más sencilla que la del **ejercicio 2.3.11**? Consejo: utiliza eliminación perezosa, técnica que veremos en nuestro código de Dijkstra, en la sección 4.4.3.

### Ejercicio 2.3.14\*

¿Es posible utilizar un BST equilibrado (por ejemplo, `set` de la STL de C++ o `TreeSet` de Java) para implementar una cola de prioridad, con el mismo rendimiento de entrada y salida de la cola en  $O(\log n)$ ? Si la respuesta es afirmativa, ¿cómo? ¿Hay ventajas o desventajas potenciales con esta estrategia? Si no, ¿por qué?

### Ejercicio 2.3.15\*

¿Hay una forma mejor de implementar una cola de prioridad, si todas las claves son enteros dentro de un rango pequeño, por ejemplo,  $[0 \dots 100]$ ? Queremos un rendimiento de entrada y salida de la cola de  $O(1)$ . Si la respuesta es afirmativa, ¿cómo? Si es que no, ¿por qué? ¿Y si el rango es solo  $[0..1]$ ?

### Ejercicio 2.3.16

¿Qué estructura de datos no lineal utilizarías si tienes que trabajar con las tres siguientes operaciones dinámicas: 1) muchas inserciones, 2) muchas eliminaciones y 3) muchas peticiones de datos ordenados?

### Ejercicio 2.3.17

Hay  $M$  **cadenas**.  $N$  de ellas son únicas ( $N \leq M$ ). ¿Qué estructura de datos no lineal, que se ha tratado en esta sección, deberías usar si tienes que indexar (etiquetar) las  $M$  cadenas, con enteros en el rango  $[0..N-1]$ ? El criterio de indexación es el siguiente: la primera cadena debe tener el índice 0, la siguiente cadena diferente tendrá el 1 y así sucesivamente. Sin embargo, si vuelve a aparecer una cadena, debe tener el mismo índice que en la ocasión anterior. Una aplicación de esta tarea es la de construir el grafo de conexiones de una lista de nombres de ciudades (que no son índices de enteros) y otra lista de carreteras entre ellas (ver la sección 2.4.1). Para hacerlo, primero tenemos que vincular los nombres de las ciudades a índices de enteros (ya que son mucho más cómodos para trabajar).

## Ejercicios de programación

Ejercicios de programación que se resuelven con una biblioteca de estructuras de datos no lineales:

map en la STL de C++ (y TreeMap en Java)

- |                                            |                                                                                                                                          |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00417 - Word Index                  | (generar todas las palabras; añadir a map para ordenación automática)                                                                    |
| 2. UVa 00484 - The Department ...          | (mantener la frecuencia con map)                                                                                                         |
| 3. UVa 00860 - Entropy Text Analyzer       | (contar la frecuencia)                                                                                                                   |
| 4. UVa 00939 - Genes                       | (map de los nombres de los hijos a sus genes y nombres de los padres)                                                                    |
| 5. UVa 10132 - File Fragmentation          | (usar map; fuerza bruta)                                                                                                                 |
| 6. UVa 10138 - CDVII                       | (mapa de matrículas a facturas; hora de entrada y posición)                                                                              |
| 7. <b>UVa 10226 - Hardwood Species *</b>   | (map está bien, pero el <i>hashing</i> es mejor)                                                                                         |
| 8. UVa 10282 - Babelfish                   | (problema puro de diccionario; usar map)                                                                                                 |
| 9. UVa 10295 - Hay Points                  | (usar map para tratar el diccionario de <i>Hay Points</i> )                                                                              |
| 10. UVa 10686 - SQF Problem                | (usar map para gestionar los datos)                                                                                                      |
| 11. UVa 11239 - Open Source                | (usar map y set para comprobar cadenas anteriores)                                                                                       |
| 12. <b>UVa 11286 - Conformity *</b>        | (usar map para registrar las frecuencias)                                                                                                |
| 13. UVa 11308 - Bankrupt Baker             | (usar map y set para gestionar los datos)                                                                                                |
| 14. UVa 11348 - Exhibition                 | (usar map y set para comprobar que son únicos)                                                                                           |
| 15. <b>UVa 11572 - Unique Snowflakes *</b> | (usar map para registrar el índice de aparición de un tamaño de copo determinado; usarlo para determinar la respuesta en $O(n \log n)$ ) |
| 16. UVa 11629 - Ballot evaluation          | (usar map)                                                                                                                               |
| 17. UVa 11860 - Document Analyzer          | (usar set y map; barrido lineal)                                                                                                         |

- 18. UVa 11917 - Do Your Own Homework (usar map)
- 19. UVa 12504 - Updating a Dictionary (usar map; cadena a cadena)
- 20. UVa 12592 - Slogan Learning of Princess (usar map; cadena a cadena)

Ver también la frecuencia de aparición en la sección 6.3.

#### **set en la STL de C++ (TreeSet en Java)**

- 1. UVa 00501 - Black Box (usar multiset, con manipulación eficiente del iterador)
- 2. **UVa 00978 - Lemmings Battle \*** (simulación; usar multiset)
- 3. UVa 10815 - Andy's First Dictionary (usar set y string)
- 4. UVa 11062 - Andy's Second Dictionary (similar a UVa 10815, con algunas variantes)
- 5. **UVa 11136 - Hoax or what \*** (usar multiset)
- 6. **UVa 11849 - CD \*** (usar set para entrar en el tiempo límite; mejor: usar hashing)
- 7. UVa 12049 - Just Prune The List (manipulación de multiset)

#### **priority\_queue en la STL de C++ (PriorityQueue en Java)**

- 1. **UVa 01203 - Argus \*** (LA 3135 - Beijing04; usar priority\_queue)
- 2. **UVa 10954 - Add All \*** (usar priority\_queue; voraz)
- 3. **UVa 11995 - I Can Guess ... \*** (stack, queue y priority\_queue)

Ver también el uso de priority\_queue para ordenaciones topológicas (sección 4.2.1), algoritmos de Kruskal<sup>19</sup> (sección 4.3.2), Prim (sección 4.3.3), Dijkstra (sección 4.4.3) y búsqueda A\* (sección 8.2.5)

## 2.4 Estructuras de datos con nuestras propias bibliotecas

A fecha de 24 de mayo de 2013, las estructuras de datos importantes que se mencionan en esta sección no tienen todavía soporte integrado en la STL de C++ o en Java. Por ello, en aras de la competitividad, los concursantes deberán preparar implementaciones libres de errores para trabajar con ellas. En esta sección, trataremos las ideas clave y aportaremos ejemplos de implementaciones (se deben consultar también los códigos fuente relacionados) para su uso.

### 2.4.1 Grafo

Un grafo es una estructura ubicua que aparece en muchos problemas de las ciencias de la computación. En su forma básica, un grafo ( $G = (V, E)$ ), es, sencillamente, un conjunto de vértices ( $V$ ) y aristas ( $E$ , que almacena la información de conectividad entre los vértices en  $V$ ). Más adelante, en los capítulos 3, 4, 8 y 9, trataremos muchos problemas y algoritmos importantes relativos a los grafos. Para prepararnos, observaremos tres métodos básicos (existen otras estructuras de datos para almacenar grafos, pero son menos habituales) para representar un grafo  $G$  con  $V$  vértices y  $E$  aristas<sup>20</sup>.

<sup>19</sup>Ésta es otra forma de implementar la ordenación de aristas con el algoritmo de Kruskal. Nuestra implementación (en C++) de la sección 4.3.2 usa solo vector y sort, en vez de priority\_queue (montículos).

<sup>20</sup>La notación más apropiada para la cardinalidad de un conjunto  $S$  es  $|S|$ . Sin embargo, en este libro, aumentaremos habitualmente el significado de  $V$  y  $E$  para indicar también  $|V|$  y  $|E|$ , dependiendo del contexto.

| Matriz de adyacencia                                                                                                                     | Lista de adyacencia                                               | Lista de aristas                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <pre> 0 1 2 3 4 5 6 0 0 2 5 0 0 0 1 2 0 7 1 0 0 0 2 5 7 0 0 4 0 0 3 0 1 0 0 3 0 0 4 0 0 4 3 0 9 0 5 0 0 0 0 9 0 8 6 0 0 0 0 0 8 0 </pre> | <pre> 0: 1 2 1: 0 2 3 2: 0 1 4 3: 1 4 4: 2 3 5 5: 4 6 6: 5 </pre> | <pre> 0: 0 1 1: 0 2 2: 1 0 3: 1 2 4: 1 3 5: 2 0 6: 2 1 7: 2 4 8: 3 1 9: 3 4 10: 4 2 11: 4 3 </pre> |

Figura 2.4: Visualización de la estructura de datos de un grafo

A) La matriz de adyacencia, normalmente en la forma de un *array* bidimensional (figura 2.4).

En los problemas (de concursos) que implican grafos, normalmente conocemos el número de vértices  $V$ . Como consecuencia, podemos construir una ‘tabla de conectividad’ creando un *array* bidimensional estático: `int AdjMat[V][V]`. Esto tiene una complejidad *espacial*<sup>21</sup> de  $O(V^2)$ . Para un grafo no ponderado, establecemos  $\text{AdjMat}[i][j]$  a un valor distinto de cero (normalmente 1), si existe una arista entre los vértices  $i-j$ , o a cero en caso contrario. Una matriz de adyacencia no se puede utilizar para almacenar grafos múltiples. En un grafo sencillo sin bucles, la diagonal principal de la matriz solo contendrá ceros, es decir,  $\text{AdjMat}[i][i] = 0$ ,  $\forall i \in [0..V-1]$ .

La matriz de adyacencia es una buena solución si debemos consultar frecuentemente la conectividad de dos vértices en un *grafo denso pequeño*. Sin embargo, su uso no es recomendable en un *grafo disperso grande*, ya que ocuparía demasiado espacio ( $O(V^2)$ ) y causaría que demasiadas celdas de la matriz bidimensional quedasen vacías (valor cero). En el entorno competitivo resulta inasumible, normalmente, utilizar matrices de adyacencia cuando el  $V$  dado es mayor que  $\approx 1000$ . Otra desventaja de una matriz de adyacencia es que requiere un tiempo de  $O(V)$  para enumerar la lista de vecinos de un vértice  $v$ , lo que es una operación muy común en muchos algoritmos, aunque dicho vértice tenga muy pocos vecinos. A continuación, veremos una representación más compacta y eficiente de un grafo, la lista de adyacencia.

B) La lista de adyacencia, normalmente como un vector de vectores de parejas (figura 2.4).

Usando la STL de C++: `vector<vector<pair<int, int>> AdjList`, con `pair<int, int>` definido como:

```
typedef pair<int, int> ii; typedef vector<ii> vii; // tipos de datos
```

Usando Java: `Vector<Vector<IntegerPair>> AdjList`. `IntegerPair` es una clase de Java sencilla, que contiene parejas de enteros, como antes `ii`.

En las listas de adyacencia tenemos un `vector` de `vector` de parejas, donde almacenamos la lista de vecinos de cada vértice  $u$  como parejas de ‘información de la arista’. Cada pareja contiene dos unidades de información, el índice del vértice vecino y el peso de la arista. Si el

<sup>21</sup>Diferenciamos entre las complejidades de *espacio* y de *tiempo* de las estructuras de datos. La complejidad de *espacio* es una medida asintótica de los requisitos de memoria de una estructura de datos, mientras que la complejidad de *tiempo* lo es del tiempo necesario para ejecutar un algoritmo determinado o realizar una operación en la propia estructura de datos.

grafo no es ponderado, basta con almacenar el peso como 0, 1 o descartar la información<sup>22</sup> por completo. La complejidad de espacio de una lista de adyacencia es de  $O(V + E)$ , porque si hay  $E$  aristas bidireccionales en un grafo (sencillo), la lista solo almacenará  $2E$  parejas de ‘información de aristas’. Como, normalmente,  $E$  es mucho más pequeño que  $V \times (V - 1)/2 = O(V^2)$  (el número máximo de aristas en un grafo sencillo completo), las listas de adyacencia suelen resultar más eficientes que las matrices de adyacencia en lo que a espacio se refiere. Además, se puede utilizar una lista de adyacencia para almacenar un grafo múltiple.

Con las listas de adyacencia también podemos enumerar la lista de vecinos de un vértice  $v$  de forma eficiente. Si  $v$  tiene  $k$  vecinos, tal enumeración tendrá un coste en tiempo de  $O(k)$ . Como ésta es una de las operaciones más comunes en la mayoría de algoritmos de grafos, se recomienda el uso de listas de adyacencia como primera elección para la representación de grafos. Salvo que se indique lo contrario, la mayoría de los algoritmos de grafos tratados en este libro utilizarán listas de adyacencia.

- C) La lista de aristas, normalmente en forma de un vector de 3-tuplas (ver la figura 2.4).

Usando la STL de C++: `vector< pair<int, ii> > EdgeList`. Usando Java: `Vector< IntegerTriple > EdgeList`. `IntegerTriple` es una clase que contiene una 3-tupla de enteros, como antes `pair<int, ii>`.

En la lista de aristas, almacenamos una lista de todas las  $E$  aristas, siguiendo algún tipo de orden. En los grafos dirigidos, almacenamos las aristas bidireccionales dos veces, una por cada dirección. La complejidad en espacio es, claramente,  $O(E)$ . Esta representación de grafos resulta muy útil para el algoritmo de Kruskal para MST (sección 4.3.2), donde se debe ordenar la colección de aristas unidireccionales<sup>23</sup> por peso ascendente. Sin embargo, almacenar información del grafo en la lista de aristas complica muchos algoritmos que requieren la enumeración de las aristas incidentes a un vértice.



### Grafo implícito

En ocasiones *no es* necesario almacenar los grafos en un estructura de datos de grafos o generarlos explícitamente para que se pueda operar con ellos. Son los llamados grafos *implícitos*. Los encontrarás en los siguientes capítulos. Hay dos tipos de grafos implícitos:

1. Es fácil determinar las aristas.

Ejemplo 1: navegación de un mapa de una rejilla bidimensional (ver la figura 2.5.A). Los vértices son las celdas de la rejilla bidimensional, donde ‘.’ representa tierra y ‘#’

<sup>22</sup>Para simplificar la explicación, asumiremos que siempre existe el segundo atributo en todas las implementaciones de grafos que incluimos en el libro, aunque no se utilice.

<sup>23</sup>Los objetos `pair` de C++ son muy fáciles de ordenar. El criterio de ordenación predeterminado consiste en colocar el primer elemento y, después, el segundo, para deshacer empates. En Java, podemos escribir nuestra propia clase `IntegerPair/IntegerTriple`, que implemente `Comparable`.

representa un obstáculo. Es posible determinar las aristas con facilidad: hay una arista entre dos celdas adyacentes si comparten un borde N/S/E/O, y si ambas son ‘?’ (ver la figura 2.5.B).

Ejemplo 2: el grafo de los movimientos del caballo de ajedrez, en un tablero  $8 \times 8$ . Los vértices son las casillas del tablero. Dos casillas tienen una arista entre ellas si difieren por dos posiciones horizontalmente y una verticalmente (o viceversa). En la figura 2.5.C se muestran las primeras tres filas y cuatro columnas de un tablero de ajedrez (se excluyen muchos vértices y aristas).

- Las aristas se pueden determinar según alguna regla.

Ejemplo: un grafo contiene  $N$  vértices ( $[1..N]$ ). Hay una arista entre dos vértices  $i$  y  $j$  si  $(i + j)$  es un número primo. En la figura 2.5.D se muestra un grafo de ese tipo con  $N = 5$ , y hay varios ejemplos más en la sección 8.2.3.

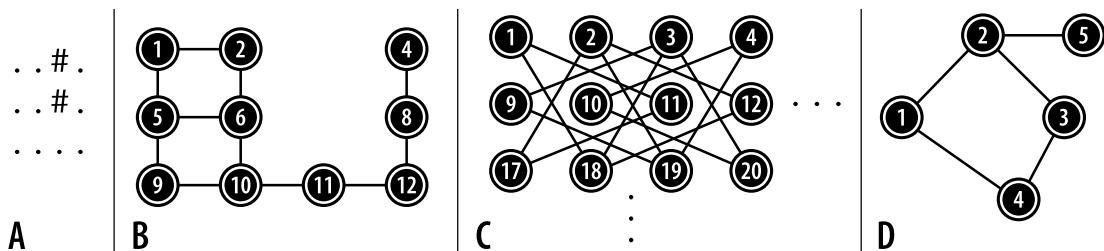


Figura 2.5: Ejemplos de grafos implícitos

### Ejercicio 2.4.1.1\*

Crea las representaciones de matriz de adyacencia, lista de adyacencia y lista de aristas de los grafos mostrados en las figuras 4.1 (sección 4.2.1) y 4.9 (sección 4.2.9). Consejo: utiliza la herramienta de visualización de estructuras de datos de grafos mostrada antes.

### Ejercicio 2.4.1.2\*

Dado un grafo (simple) con una representación (matriz de adyacencia/AM, lista de adyacencia/AL o lista de aristas/EL), *convíértela* en otra representación de la forma más eficiente posible. Tenemos seis conversiones posibles:  $AM \rightarrow AL$ ,  $AM \rightarrow EL$ ,  $AL \rightarrow AM$ ,  $AL \rightarrow EL$ ,  $EL \rightarrow AM$  y  $EL \rightarrow AL$ .

### Ejercicio 2.4.1.3

Si la matriz de adyacencia de un grafo (simple) tiene la propiedad de que es igual a su traspuesto, ¿qué implica esto?

### Ejercicio 2.4.1.4\*

Dado un grafo (simple), representado por una matriz de adyacencia, realiza las siguientes tareas de la forma más eficiente. Una vez que hayas descubierto cómo hacerlo con matrices de adyacencia, repite la tarea con listas de adyacencia y listas de aristas.

1. Cuenta el número de vértices  $V$  y aristas unidireccionales  $E$  (asumiendo que una arista bidireccional equivale a dos aristas unidireccionales) del grafo.
- 2\*. Cuenta los grados de entrada y salida de un vértice  $v$  determinado.
- 3\*. Realiza una traspósicion del grafo (invierte la dirección de las aristas).
- 4\*. Comprueba si el grafo es completo  $K_n$ . Nota: un grafo completo es un grafo simple no dirigido en el que *cada pareja* de vértices distintos está conectada por una sola arista.
- 5\*. Comprueba si el grafo es un árbol (conexo no dirigido con  $E = V - 1$  aristas).
- 6\*. Comprueba si el grafo es en estrella  $S_k$ . Un grafo en estrella  $S_k$ , es un grafo bipartito completo  $K_{1,k}$ . Un árbol con un único vértice interno y  $k$  hojas.

### Ejercicio 2.4.1.5\*

Investiga otros métodos posibles de representación de grafos, diferentes a los ya tratados, especialmente para almacenar grafos especiales.

## 2.4.2 Conjuntos disjuntos para unión-buscar

El conjunto disjunto para unión-buscar (UFDS) es una estructura de datos utilizada para modelar una colección de *conjuntos disjuntos*, con la capacidad de determinar de forma eficiente<sup>24</sup>, en  $\approx O(1)$ , a qué conjunto pertenece un elemento (o si dos elementos pertenecen al mismo conjunto) y para combinar dos conjuntos disjuntos en un conjunto mayor. Esta estructura de datos también se puede utilizar para resolver problemas de búsqueda de componentes conexos en un grafo no dirigido (sección 4.2.3). Inicializamos cada vértice en un conjunto disjunto separado, después enumeramos las aristas y uniones del grafo cada dos vértices/conjuntos disjuntos conectados por una arista. A continuación, podremos comprobar si dos vértices pertenecen al mismo componente/conjunto fácilmente. El número de conjuntos disjuntos que se pueden seguir con facilidad también denota el número de componentes conexos del grafo no dirigido.

Estas operaciones, aparentemente sencillas, no están incorporadas de forma *eficiente* al `set` de la STL de C++ (y al `TreeSet` de Java), que no está diseñado con este propósito. Recorrer completamente cada `set` en un `vector` y buscar a cuál pertenece determinado elemento, tiene

<sup>24</sup>  $M$  operaciones de la estructura de datos UFDS con heurística de ‘compresión de rutas’ y ‘unión por rango’ se ejecutan en  $O(M \times \alpha(n))$ . Sin embargo, como la función inversa de Ackermann  $\alpha(n)$  crece muy despacio, es decir, su valor es inferior a 5 en un tamaño de entrada típico en los concursos de programación de  $n \leq 1M$ , podemos tratar  $\alpha(n)$  como una constante.

un coste muy elevado. Tampoco será eficiente el `set_union` de la STL de C++ (en `algorithm`), aunque combina dos conjuntos en *tiempo lineal*, ya que todavía tendremos que lidiar con la mezcla del contenido del `vector` de `set`. Para realizar estas operaciones con eficacia, necesitamos una estructura de datos mejor, el UFDS.

La innovación principal de esta estructura de datos la encontramos en la elección de un elemento ‘padre’ representativo de un conjunto. Si podemos asegurarnos de que cada conjunto está representado por un único elemento, determinar si dos elementos pertenecen al mismo conjunto se vuelve mucho más sencillo: el elemento ‘padre’ representativo se puede utilizar como un tipo de identificador del conjunto. Para lograrlo, el conjunto disjunto para unión-buscar crea una estructura en árbol, donde los conjuntos disjuntos forman un bosque. Cada árbol corresponde a un conjunto disjunto. Queda determinada la raíz del árbol como el elemento representativo de un conjunto. Por lo tanto, el identificador del conjunto de un elemento se puede obtener, de forma sencilla, siguiendo la cadena de padres, hasta alcanzar la raíz del árbol y, como cada árbol solo puede tener una raíz, este elemento representativo se puede utilizar como el identificador único del conjunto.

Para llevar a cabo esta tarea de forma eficiente, almacenamos el índice del elemento padre y (el límite superior de) la altura del árbol de cada conjunto (`vi p` y `vi rank` en nuestra implementación). Recorda que `vi` es nuestro atajo para un vector de enteros. `p[i]` almacena el parente inmediato del elemento *i*. Si el elemento *i* es el representativo de un determinado conjunto disjunto, entonces `p[i] = i`, es decir, se refiere a sí mismo. `rank[i]` determina (el límite superior de) la altura del árbol con raíz en el elemento *i*.

En esta sección utilizaremos 5 conjuntos disjuntos  $\{0, 1, 2, 3, 4\}$  para ilustrar el uso de esta estructura de datos. Inicializaremos la estructura de forma que cada elemento sea un conjunto disjunto en sí mismo, con rango 0, y el parente de cada elemento será, inicialmente, él mismo.

Para combinar dos conjuntos disjuntos, hacemos que el elemento representativo (raíz) de uno de ellos sea el nuevo parente del elemento representativo del otro. Esto supone, efectivamente, la combinación de dos árboles en la representación del conjunto disjunto para unión-buscar. De esta forma, `unionSet(i, j)` provocará que ambos elementos *i* y *j* tengan el mismo elemento representativo, de forma directa o indirecta. Por razones de eficiencia, podremos utilizar la información contenida en `vi rank`, para hacer que el elemento representativo del conjunto disjunto con *mayor rango* sea el nuevo parente del de *menor rango*, *manteniendo* así el rango del árbol resultante. Si ambos rangos son iguales, podemos elegir uno u otro arbitrariamente como nuevo parente e incrementar el rango de la raíz resultante. Es la heurística de ‘unión por rango’. En la parte superior de la figura 2.6, `unionSet(0, 1)` establece `p[0]` a 1 y `rank[1]` también a 1. En la parte central de la figura 2.6, `unionSet(2, 3)` establece `p[2]` a 3 y `rank[3]` a 1.

De momento, vamos a asumir que la función `findSet(i)` se limita a llamar a `findSet(p[i])` de forma recursiva, para encontrar el elemento representativo de un conjunto, devolviendo `findSet(p[i])` siempre que `p[i] != i` e *i* en caso contrario. En la parte inferior de la figura 2.6, cuando llamamos a `unionSet(4, 3)`, tenemos que `rank[findSet(4)] = rank[4] = 0`, que resulta más pequeño que `rank[findSet(3)] = rank[3] = 1`, así que establecemos `p[4] = 3`, *sin cambiar* la altura del árbol resultante, lo que es un ejemplo de la heurística ‘unión por rango’ en funcionamiento. Al utilizar este método, el camino que va desde cualquier nodo a su elemento representativo, siguiendo la cadena de enlaces ‘padre’, queda minimizado.

En la parte inferior de la figura 2.6, `isSameSet(0, 4)` demuestra otra operación para esta estructura de datos. La función `isSameSet(i, j)` se limita a llamar a `findSet(i)` y `findSet(j)`, y comprueba si ambos se refieren al mismo elemento representativo. Si la respuesta es afir-

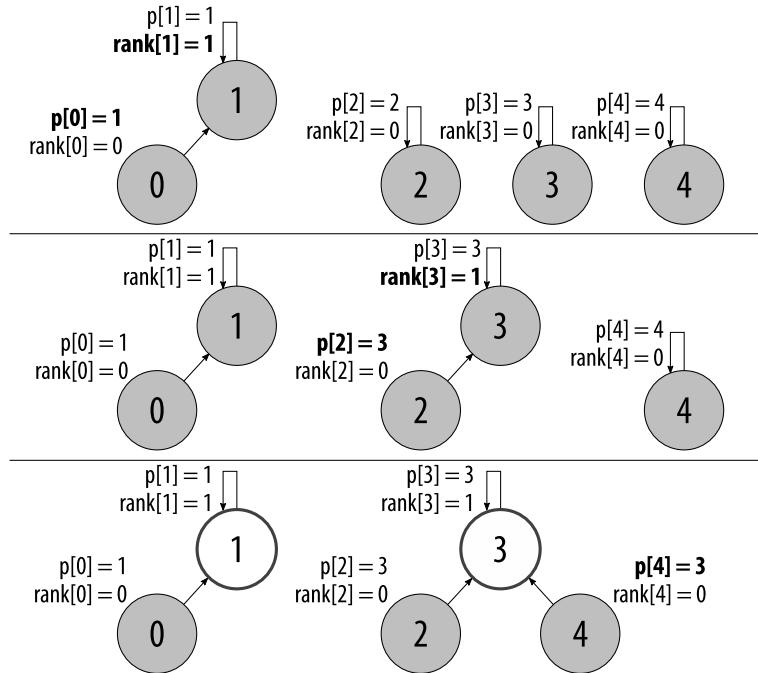


Figura 2.6:  $\text{unionSet}(0, 1) \rightarrow (2, 3) \rightarrow (4, 3)$  e  $\text{isSameSet}(0, 4)$

mativa, significa que tanto  $i$  como  $j$  pertenecen al mismo conjunto. En este caso, podemos ver que  $\text{findSet}(0) = \text{findSet}(p[0]) = \text{findSet}(1) = 1$  no es lo mismo que  $\text{findSet}(4) = \text{findSet}(p[4]) = \text{findSet}(3) = 3$ . Por lo tanto, los elementos 0 y 4 pertenecen a conjuntos disjuntos *diferentes*.

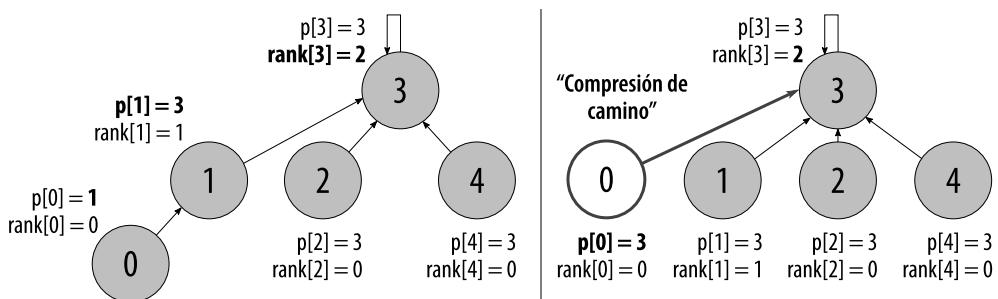


Figura 2.7:  $\text{unionSet}(0, 3) \rightarrow \text{findSet}(0)$

Existe una técnica que puede acelerar enormemente la función  $\text{findSet}(i)$ : la compresión de caminos. Siempre que encontramos el elemento representativo (raíz) de un conjunto disjunto, siguiendo la cadena de enlaces ‘padre’ desde un elemento determinado, podemos establecer que el padre de *todos los elementos* recorridos sea, directamente, la raíz. Cualquier llamada posterior a  $\text{findSet}(i)$ , sobre los elementos implicados, resultará en la consulta de un solo enlace. Esto provoca un cambio en la estructura del árbol (para hacer  $\text{findSet}(i)$  más eficiente), pero conserva la constitución real del conjunto disjunto. Como esto ocurrirá cada vez que se llame a  $\text{findSet}(i)$ , el efecto combinado supondrá la reducción del tiempo de ejecución de las  $M$

llamadas a `findSet(i)` a un tiempo amortizado  $O(M \times \alpha(n))$ , extremadamente eficiente. Para el caso de la programación competitiva, donde  $n$  es razonablemente pequeño, podemos tratar  $\alpha(n)$  como una operación constante en  $O(1)$ .

En la figura 2.7, mostramos la ‘compresión de camino’. Primero, llamamos a `unionSet(0, 3)`. Esta vez, establecemos  $p[1] = 3$  y actualizamos  $rank[3] = 2$ . Nos fijamos en que  $p[0]$  no cambia, es decir  $p[0] = 1$ . Esto es una referencia *indirecta* al elemento representativo (verdadero) del conjunto, quedando  $p[0] = 1 \rightarrow p[1] = 3$ . La función `findSet(i)` necesitará, de hecho, más de un paso para recorrer la cadena de enlaces ‘padre’ hasta la raíz. Sin embargo, una vez que se encuentra el elemento representativo (por ejemplo ‘x’) del conjunto, se *comprimirá el camino* al establecer  $p[i] = x$ , es decir, `findSet(0)` establece  $p[0] = 3$ . Por lo tanto, las siguientes llamadas a `findSet(i)` se realizarán en  $O(1)$ . Esta sencilla estrategia se llama, oportunamente, la heurística de ‘compresión de caminos’. Hay que fijarse en que  $rank[3] = 2$  ya no refleja la *altura real* del árbol. Por eso, `rank` solo hace referencia al *límite superior* de la verdadera altura del árbol. A continuación, mostramos nuestra implementación en C++:

```
1 class UnionFind { // estilo OOP
2 private: vi p, rank; // recuerda: vi es vector<int>
3 public:
4 UnionFind(int N) { rank.assign(N, 0); }
5 p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
6 int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
7 bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
8 void unionSet(int i, int j) {
9 if (!isSameSet(i, j)) { // si es de otro conjunto
10 int x = findSet(i), y = findSet(j);
11 if (rank[x] > rank[y]) p[y] = x; // rank mantiene el árbol corto
12 else { p[x] = y;
13 if (rank[x] == rank[y]) rank[y]++; }
14 } } };
```



ch2\_08\_unionfind\_ds.cpp



ch2\_08\_unionfind\_ds.java

### Ejercicio 2.4.2.1

Hay otras dos consultas que se suelen realizar en esta estructura de datos. Actualiza el código de esta sección para operar con ellas de forma eficiente: `int numDisjointSets()`, que devuelve el número de conjuntos disjuntos que están en la estructura; e `int sizeOfSet(int i)`, que devuelve el tamaño del conjunto que contiene el elemento  $i$ .

### Ejercicio 2.4.2.2\*

Dados 8 conjuntos disjuntos  $\{0, 1, 2, \dots, 7\}$ , crear una secuencia de operaciones `unionSet(i, j)` que genere un árbol de rango = 3. ¿Es posible para rango = 4?

#### Perfiles de los inventores de estructuras de datos

**George Boole** (1815-1864) fue un matemático, filósofo y lógico inglés. Conocido principalmente entre los ingenieros informáticos como el creador de la lógica booleana, la base de los ordenadores digitales modernos. Se le considera el fundador del campo de las ciencias de la computación.

**Rudolf Bayer** (nacido en 1939) ha sido profesor (emérito) de informática en la Universidad Técnica de Munich. Inventó el árbol Red-Black (RB) utilizado en `map/set` de la STL de C++.

**Georgii Adelson-Velskii** (nacido en 1922) es un matemático y científico de la computación soviético. Junto con Evgenii Mikhailovich Landis, inventó el árbol AVL en 1962.

**Evgenii Mikhailovich Landis** (1921-1997) fue un matemático soviético. El nombre del árbol AVL es un acrónimo de los apellidos de sus dos inventores: Adelson-Velskii y el propio Landis.

#### 2.4.3 Árbol de segmentos

En esta subsección, trataremos una estructura de datos que puede responder, de forma eficiente, a consultas de rango *dinámicas*<sup>25</sup>. Una de esas consultas de rango la encontramos en el problema de buscar el índice del elemento mínimo en un *array* dentro del rango  $[i..j]$ . Esta cuestión se conoce como el problema de la consulta de rango mínimo (RMQ). Por ejemplo, si tenemos un array A de tamaño  $n = 7$ , como el que aparece a continuación,  $RMQ(1, 3) = 2$ , ya que el índice 2 contiene el elemento mínimo entre  $A[1], A[2]$  y  $A[3]$ . Para comprobar tu comprensión de la RMQ, verifica que en este *array* A  $RMQ(3, 4) = 4$ ,  $RMQ(0, 0) = 0$ ,  $RMQ(0, 1) = 1$  y  $RMQ(0, 6) = 5$ . En los siguientes párrafos asumiremos que A es el mismo *array*.

| Array | Valores | 18 | 17 | 13 | 19 | 15 | 11 | 20 |
|-------|---------|----|----|----|----|----|----|----|
| A     | Índices | 0  | 1  | 2  | 3  | 4  | 5  | 6  |

Existen varias formas de implementar la RMQ. Un algoritmo trivial consiste en iterar el *array* desde el índice i hasta j, e informar del que tiene el valor mínimo, pero esto se ejecuta en tiempo  $O(n)$  por consulta. Si n es grande y hay muchas consultas, un algoritmo así resulta inviable.

En esta sección, respondemos al problema de la RMQ dinámica con un árbol de segmentos, que es otra forma de organizar datos en un árbol binario. Existen varias formas de implementar el árbol de segmentos. La nuestra utiliza el mismo concepto que el *array* compacto con índice a partir de 1 del montículo binario, donde utilizamos `vi` (nuestro atajo para `vector<int>`) `st` para representar el árbol binario. El índice 1 (no utilizamos el 0) es la raíz, y los hijos izquierdo y derecho del índice p son  $2 \times p$  y  $(2 \times p) + 1$ , respectivamente (consultar la sección 2.3, sobre el montículo binario). El valor de `st[p]` es el valor RMQ del segmento asociado al índice p.

<sup>25</sup>En los problemas dinámicos es habitual *actualizar* y consultar la información. Esto hace que las técnicas de procesamiento previo resulten inútiles.

La raiz del árbol de segmentos representa al segmento  $[0, n-1]$ . Para cada segmento  $[L, R]$  almacenado en el índice  $p$ , donde  $L \neq R$ , dicho segmento se divide en  $[L, (L+R)/2]$  y  $[(L+R)/2+1, R]$ , para sus vértices izquierdo y derecho. Los subsegmentos izquierdo y derecho se almacenan en los índices  $2 \times p$  y  $(2 \times p) + 1$ , respectivamente. Cuando  $L = R$ , es evidente que  $st[p] = L$  (o  $R$ ). En caso contrario, construiremos, de forma recursiva, el árbol de segmentos, comparando el valor mínimo de los subsegmentos izquierdo y derecho, y actualizando el valor  $st[p]$  del segmento. Este proceso queda implementado en la rutina `build`, que mostramos más adelante. Esta rutina, `build`, crea hasta  $O(1 + 2 + 4 + 8 + \dots + 2^{\log_2 n}) = O(2n)$  segmentos (más pequeños) y, en consecuencia, se ejecuta en  $O(n)$ . Sin embargo, como utilizamos la indexación de un *array* compacto sencilla contando desde 1, necesitamos que  $st$  tenga, al menos, un tamaño  $2 \times 2^{\lfloor \log_2(n) \rfloor + 1}$ . En nuestra implementación, nos basta con utilizar un límite superior flexible de complejidad de espacio  $O(4n) = O(n)$ . En las figuras 2.8 y 2.9, se muestra el árbol de segmentos correspondiente al array A anterior.

Con el árbol de segmentos ya preparado, se puede responder a una RMQ en  $O(\log n)$ . La respuesta para  $\text{RMQ}(i, i)$  es trivial, basta con devolver la propia  $i$ . Sin embargo, para el caso general  $\text{RMQ}(i, j)$ , son necesarias más comprobaciones. Digamos que  $p1 = \text{RMQ}(i, (i+j)/2)$  y  $p2 = \text{RMQ}((i+j)/2 + 1, j)$ . Aquí  $\text{RMQ}(i, j)$  es  $p1$  si  $A[p1] \leq A[p2]$  o  $p2$  en caso contrario. La rutina `rmq`, que mostramos más adelante, implementa este proceso.

Tomemos como ejemplo la consulta  $\text{RMQ}(1, 3)$ . El proceso mostrado en la figura 2.8 es el siguiente: comenzamos desde la raiz (índice 1), que representa el segmento  $[0, 6]$ . No podemos usar el mínimo almacenado del segmento  $[0, 6] = st[1] = 5$  como respuesta de  $\text{RMQ}(1, 3)$ , ya que es el valor mínimo de un segmento más grande<sup>26</sup> que el deseado  $[1, 3]$ . Partiendo de la raiz, solo tenemos que ir al subárbol izquierdo, ya que la raiz del derecho representa al segmento  $[4, 6]$ , que está fuera<sup>27</sup> del rango deseado en  $\text{RMQ}(1, 3)$ .

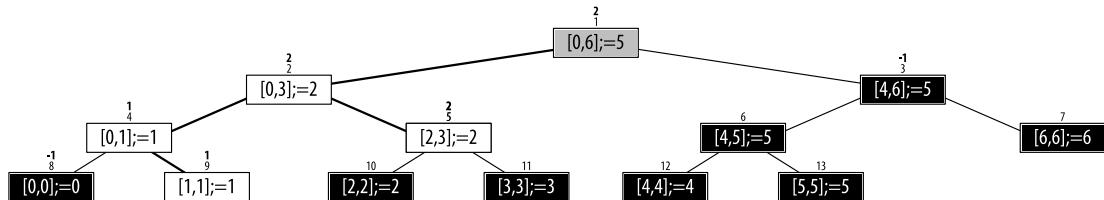


Figura 2.8: Árbol de segmentos del *array*  $A = \{18, 17, 13, 19, 15, 11, 20\}$  y  $\text{RMQ}(1, 3)$

Ahora, nos encontramos en la raiz del subárbol izquierdo (índice 2), que representa al segmento  $[0, 3]$ . Este sigue siendo más grande que la  $\text{RMQ}(1, 3)$  buscada. De hecho,  $\text{RMQ}(1, 3)$  está interseccionada *tanto* con el subsegmento izquierdo  $[0, 1]$  (índice 4) como el derecho  $[2, 3]$  (índice 5) del segmento  $[0, 3]$ , por lo que tenemos que explorar *ambos* subárboles.

El segmento izquierdo  $[0, 1]$  (índice 4) de  $[0, 3]$  (índice 2) no está todavía en  $\text{RMQ}(1, 3)$ , por lo que se hace necesaria otra división. Desde el segmento  $[0, 1]$  (índice 4), nos movemos a la derecha al segmento  $[1, 1]$  (índice 9), que ahora está dentro<sup>28</sup> de  $[1, 3]$ . En este punto, sabemos que  $\text{RMQ}(1, 1) = st[9] = 1$  y podemos devolver este valor como resultado. El segmento derecho  $[2, 3]$  (índice 5) de  $[0, 3]$  (índice 2) está dentro del  $[1, 3]$  solicitado. Gracias al

<sup>26</sup>Se dice que el segmento  $[L, R]$  es más grande que el rango de consulta  $[i, j]$  si  $[L, R]$  no está ni fuera ni dentro del rango de consulta (ver las otras notas al pie).

<sup>27</sup>Se dice que el segmento  $[L, R]$  está fuera del rango de consulta  $[i, j]$  si  $i > R \text{ || } j < L$ .

<sup>28</sup>Se dice que el segmento  $[L, R]$  está dentro del rango de consulta  $[i, j]$  si  $L \geq i \text{ & } R \leq j$ .

valor almacenado dentro de este vértice, sabemos que  $\text{RMQ}(2, 3) = \text{st}[5] = 2$ . No es necesario seguir recorriendo el árbol.

Volviendo a la llamada al segmento  $[0, 3]$  (índice 2), tenemos que  $p1 = \text{RMQ}(1, 1) = 1$  y  $p2 = \text{RMQ}(2, 3) = 2$ . Como  $A[p1] > A[p2]$ , ya que  $A[1] = 17$  y  $A[2] = 13$ , tenemos  $\text{RMQ}(1, 3) = p2 = 2$ . Ésa es la respuesta definitiva.

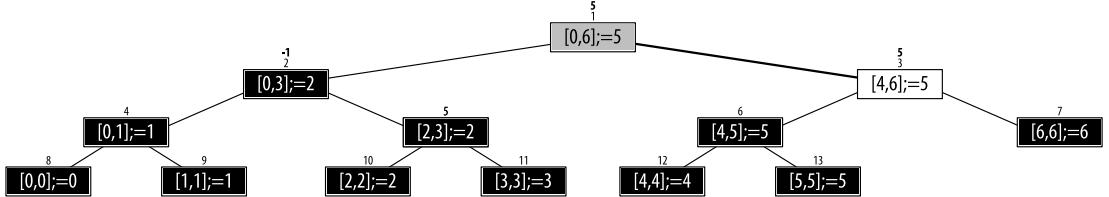


Figura 2.9: Árbol de segmentos del *array*  $A = \{18, 17, 13, 19, 15, 11, 20\}$  y  $\text{RMQ}(4, 6)$

Vamos a fijarnos en otro ejemplo:  $\text{RMQ}(4, 6)$ . La ejecución, como se muestra en la figura 2.9 es la siguiente: comenzamos desde el segmento raíz  $[0, 6]$  (índice 1). Como es mayor que  $\text{RMQ}(4, 6)$ , nos desplazamos a la derecha, al segmento  $[4, 6]$  (índice 3), ya que  $[0, 3]$  (índice 2) queda fuera. Como este segmento representa exactamente  $\text{RMQ}(4, 6)$ , basta con devolver el índice del elemento mínimo almacenado en este vértice, que es 5. Por lo tanto  $\text{RMQ}(4, 6) = \text{st}[3] = 5$ .

Esta estructura de datos nos ayuda a evitar recorrer las partes no necesarias de un árbol. En el peor caso, tendremos *dos* caminos de la raíz a una hoja, que supone solo  $O(2 \times \log(2n)) = O(\log n)$ . Por ejemplo: en  $\text{RMQ}(3, 4) = 4$ , hay un camino de la raíz a una hoja desde  $[0, 6]$  hasta  $[3, 3]$  (índice 1 → 2 → 5 → 11) y otro camino similar desde  $[0, 6]$  hasta  $[4, 4]$  (índice 1 → 3 → 6 → 12).

Si el array *A* es estático (es decir, no se ha modificado desde su creación), utilizar un árbol de segmentos para resolver el problema de la RMQ es *excesivo*, ya que existe una solución de programación dinámica que requiere un procesamiento previo en  $O(n \log n)$  y, después,  $O(1)$  por cada RMQ. Veremos esta solución más adelante, en la sección 9.33.

El árbol de segmentos es útil cuando el *array* subyacente se actualiza habitualmente (es dinámico). Por ejemplo, si  $A[5]$  cambia de 11 a 99, solo tenemos que actualizar los vértices a lo largo del camino de la hoja a la raíz en  $O(\log n)$ . La ruta es:  $[5, 5]$  (índice 13,  $\text{st}[13]$  no cambia) →  $[4, 5]$  (índice 6, ahora  $\text{st}[6] = 4$ ) →  $[4, 6]$  (índice 3, ahora  $\text{st}[3] = 4$ ) →  $[0, 6]$  (índice 1, ahora  $\text{st}[1] = 2$ ) en la figura 2.10. Para compararlo, la solución de DP presentada en la sección 9.33 requiere otro procesamiento previo, con coste  $O(n \log n)$ , para actualizar la estructura, por lo que no resulta eficiente en las actualizaciones dinámicas.

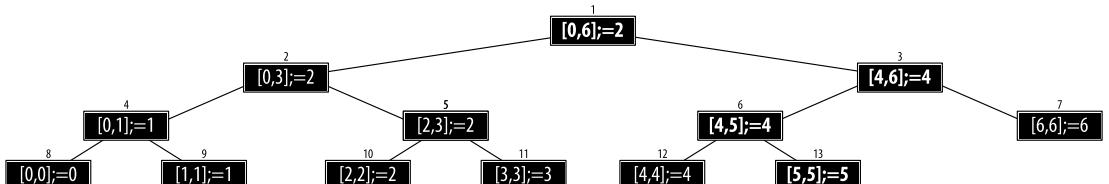


Figura 2.10: Actualización del *array*  $A$  a  $\{18, 17, 13, 19, 15, 99, 20\}$

A continuación, incluimos nuestra implementación del árbol de segmentos. Este código solo considera RMQ *estáticos* (dejamos las actualizaciones *dinámicas* como ejercicio para el lector).

```

1 class SegmentTree { // árbol de segmentos como array de montículo
2 private: vi st, A; // recuerda que vi es: typedef vector<int> vi;
3 int n;
4 int left (int p) { return p << 1; } // igual que montículos binarios
5 int right(int p) { return (p << 1) + 1; }
6
7 void build(int p, int L, int R) { // O(n)
8 if (L == R) // como L == R, cualquiera nos vale
9 st[p] = L; // guardar el índice
10 else { // calcular los valores de forma recursiva
11 build(left(p) , L , (L + R) / 2);
12 build(right(p), (L + R) / 2 + 1, R);
13 int p1 = st[left(p)], p2 = st[right(p)];
14 st[p] = (A[p1] <= A[p2]) ? p1 : p2;
15 }
16
17 int rmq(int p, int L, int R, int i, int j); // O(log n)
18 if (i > R || j < L) return -1; // segmento actual fuera del rango
19 if (L >= i && R <= j) return st[p]; // dentro del rango de la consulta
20
21 // calcular posición mínima en la izquierda y derecha del intervalo
22 int p1 = rmq(left(p) , L , (L+R) / 2, i, j);
23 int p2 = rmq(right(p), (L+R) / 2 + 1, R , i, j);
24
25 if (p1 == -1) return p2; // si vamos al segmento fuera de la consulta
26 if (p2 == -1) return p1; // igual que antes
27 return (A[p1] <= A[p2]) ? p1 : p2; // como en la rutina build
28 }
29
30 public:
31 SegmentTree(const vi &_A) {
32 A = _A; n = (int)A.size(); // copiar contenido para uso local
33 st.assign(4 * n, 0); // crear vector de ceros de tamaño suficiente
34 build(1, 0, n - 1); // build recursivo
35 }
36 int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); } // sobrecarga
37 };
38
39 int main() {
40 int arr[] = { 18, 17, 13, 19, 15, 11, 20 }; // el array original
41 vi A(arr, arr + 7);
42 SegmentTree st(A);
43 printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // respuesta = índice 2
44 printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // respuesta = índice 5
45 } // return 0;

```



### Ejercicio 2.4.3.1\*

Dibuja el árbol de segmentos correspondiente al *array A = {10, 2, 47, 3, 7, 9, 1, 98, 21}*. Responde a RMQ(1, 7) y RMQ(3, 8). Consejo: utiliza la herramienta de visualización de árboles de segmentos mostrada antes.

### Ejercicio 2.4.3.2\*

Hemos visto en esta sección cómo se pueden utilizar los árboles de segmentos para responder a consultas de rango mínimo (RMQ). Los árboles de segmentos también se pueden utilizar para responder a consultas de suma de rango (RSQ(*i, j*)), es decir, la suma de  $A[i] + A[i+1] + \dots + A[j]$ . Modifica el código del árbol de segmentos incluido, para que funcione con RSQ.

### Ejercicio 2.4.3.3

Usando un árbol de segmentos similar al del **ejercicio 2.4.3.1**, responde a las consultas RSQ(1, 7) y RSQ(3, 8). ¿Es este un buen método para resolver el problema si el *array A* no cambia nunca? (ver también las secciones 3.5.2 y 9.33).

### Ejercicio 2.4.3.4\*

El código del árbol de segmentos que hemos incluido, no cuenta con la operación de actualización (de punto), mencionada en el texto. Añade la función *update* en  $O(\log n)$ , para actualizar el valor de un índice determinado (punto) del *array A* y, al mismo tiempo, actualizar el árbol de segmentos correspondiente.

### Ejercicio 2.4.3.5\*

La operación de actualización (de punto) mostrada en el texto solo cambia el valor de un índice determinado en el *array A*. ¿Y si eliminamos los elementos existentes del *array A* o le insertamos elementos nuevos? ¿Puedes explicar qué ocurrirá con el código de árbol de segmentos dado y qué deberías hacer para solucionarlo?

### Ejercicio 2.4.3.6\*

Hay otra operación importante sobre árboles de segmentos que no hemos tratado, la operación de actualización de *rangos*. Supón que una subcadena determinada de A, se actualiza a un cierto valor común. ¿Podemos actualizar el árbol de segmentos con eficiencia? Estudia y resuelve el problema UVa 11402 - Ahoy Pirates.

#### 2.4.4 Árbol de Fenwick (binario indexado)

**El árbol de Fenwick**, también conocido como **árbol binario indexado** (BIT), fue inventado por *Peter M. Fenwick* en 1994 [18]. En este libro nos referiremos a él como árbol de Fenwick, en vez de como BIT, para diferenciarlo de las *manipulaciones de bits* normales. Un árbol de Fenwick es una estructura de datos muy útil para implementar *tablas de frecuencia acumulada dinámicas*. Supongamos que tenemos diferentes puntuaciones de exámenes<sup>29</sup> de  $m = 11$  estudiantes,  $f = \{2, 4, 5, 5, 6, 6, 6, 7, 7, 8, 9\}$ , donde éstas son *valores enteros* en el rango  $[1..10]$ . La tabla 2.1 muestra la frecuencia de cada puntuación de examen individual  $\in [1..10]$ , y la frecuencia acumulada de las puntuaciones en el rango  $[1..i]$ , indicadas por  $cf[i]$ , es decir, la suma de las frecuencias de las puntuaciones  $1, 2, \dots, i$ .

La tabla de frecuencias acumuladas se puede utilizar también como solución al problema de la consulta de suma de rangos (RSQ), mencionada en el **ejercicio 2.4.3.2\***. Almacena  $RSQ(1, i) \forall i \in [1..n]$  donde  $n$  es el índice/puntuación<sup>30</sup> entero más grande. En el ejemplo anterior, tenemos  $n = 10$ ,  $RSQ(1, 1) = 0$ ,  $RSQ(1, 2) = 1, \dots, RSQ(1, 6) = 7, \dots, RSQ(1, 8) = 10, \dots, y RSQ(1, 10) = 11$ . Podemos obtener la respuesta de la RSQ de un rango arbitrario  $RSQ(i, j)$  cuando  $i \neq 1$ , restando  $RSQ(1, j) - RSQ(1, i - 1)$ . Por ejemplo,  $RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6$ .

| Índice/<br>Puntuación | Frecuencia<br>$f$ | Frecuencia<br>acumulada $cf$ | Comentario                           |
|-----------------------|-------------------|------------------------------|--------------------------------------|
| 0                     | -                 | -                            | Índice 0 ignorado (valor centinela)  |
| 1                     | 0                 | 0                            | $cf[1] = f[1] = 0$                   |
| 2                     | 1                 | 1                            | $cf[2] = f[1]+f[2] = 0+1 = 1$        |
| 3                     | 0                 | 1                            | $cf[3] = f[1]+f[2]+f[3] = 0+1+0 = 1$ |
| 4                     | 1                 | 2                            | $cf[4] = cf[3]+f[4] = 1+1 = 2$       |
| 5                     | 2                 | 4                            | $cf[5] = cf[4]+f[5] = 2+2 = 4$       |
| 6                     | 3                 | 7                            | $cf[6] = cf[5]+f[6] = 4+3 = 7$       |
| 7                     | 2                 | 9                            | $cf[7] = cf[6]+f[7] = 7+2 = 9$       |
| 8                     | 1                 | 10                           | $cf[8] = cf[7]+f[8] = 9+1 = 10$      |
| 9                     | 1                 | 11                           | $cf[9] = cf[8]+f[9] = 10+1 = 11$     |
| 10                    | 0                 | 11                           | $cf[10] = cf[9]+f[10] = 11+0 = 11$   |

Tabla 2.1: Ejemplo de tabla de frecuencia acumulada

Si las frecuencias son *estáticas*, una tabla de frecuencia acumulada, como la mostrada en la tabla 2.1, se puede calcular de forma eficiente con un simple bucle  $O(n)$ . En primer lugar, establecemos

<sup>29</sup>Las puntuaciones aparecen ordenadas para facilitar la comprensión, pero no tienen por qué estarlo.

<sup>30</sup>Hay que diferenciar entre  $m =$  el número de datos de puntuación y  $n =$  el entero más grande entre los  $m$  datos. El significado de  $n$  en el árbol de Fenwick es un poco diferente en relación a otras estructuras de datos mostradas en este libro.

$cf[1] = f[1]$ . Después, para  $i \in [2..n]$ , calculamos  $cf[i] = cf[i - 1] + f[i]$ . Veremos más sobre esto en la sección 3.5.2. Sin embargo, cuando las frecuencias son actualizadas repetidas veces (incrementadas o decrementadas) y se realizan numerosas RSQ después, es mejor utilizar una estructura de datos *dinámica*.

En vez de utilizar un árbol de segmentos para implementar una tabla de frecuencia acumulada *dinámica*, podemos implementar, en su lugar, un árbol de Fenwick, que es *mucho más sencillo* (puedes comparar el código fuente de ambas implementaciones, el de esta sección y el de la anterior 2.4.3). Ésta es, quizás, una de las razones por las que el árbol de Fenwick se incluye en la actualidad en el temario de la IOI [20]. Las operaciones en un árbol de Fenwick son, además, extremadamente eficientes, ya que utilizan técnicas rápidas de manipulación de bits (ver la sección 2.2).

En esta sección, usaremos asiduamente la función `LSOne(i)` (que es, en realidad,  $(i \& (-i))$ ), dándole un nombre que coincide con el utilizado en el artículo original [18]. En la sección 2.2 hemos visto que la operación  $(i \& (-i))$  nos devuelve el bit menos significativo de  $i$ .

El árbol de Fenwick se implementa, típicamente, como un *array* (nosotros utilizamos un *vector*, por su flexibilidad en el tamaño). Este árbol está indexado por los *bits* de sus claves *enteras*. Estas claves enteras entran dentro del rango fijo  $[1..n]$ , no utilizando<sup>31</sup> el índice 0. En un concurso de programación,  $n$  se puede acercar a  $\approx 1M$ , de forma que el árbol de Fenwick deberá cubrir el rango  $[1..1M]$ , lo que es suficientemente grande para la mayoría de los problemas (de concursos) habituales. En la tabla 2.1 anterior, las puntuaciones  $[1..10]$  son las claves enteras en el *array* correspondiente, con tamaño  $n = 10$  y con  $m = 11$  datos de puntuación.

Pongamos que el *array* del árbol de Fenwick se llama `ft`. A partir de ahí, el elemento correspondiente al índice  $i$  es el responsable de los elementos del rango  $[(i-LSOne(i)+1)..i]$ , y `ft[i]` almacena la frecuencia acumulada de los elementos  $\{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, \dots, i\}$ . En la figura 2.11, el valor de `ft[i]` aparece en el círculo sobre el índice  $i$ , y el rango  $[i-LSOne(i)+1..i]$  lo hace como un círculo y una barra (si el rango ocupa más de un índice), sobre el mismo índice  $i$ . Podemos ver que `ft[4] = 2` es responsable del rango  $[(4-4+1)..4] = [1..4]$ , `ft[6] = 5` es responsable del rango  $[(6-2+1)..6] = [5..6]$ , `ft[7] = 2` lo es de  $[(7-1+1)..7] = [7..7]$ , `ft[8] = 10` igualmente de  $[(8-8+1)..8] = [1..8]$ , etc<sup>32</sup>.

Con una disposición así, si queremos obtener la frecuencia acumulada entre  $[1..b]$ , es decir `rsq(b)`, nos basta con sumar `ft[b]`, `ft[b']`, `ft[b'']`, ..., hasta que el índice  $b^i$  sea 0. Esta secuencia de índices se obtiene al restar el bit menos significativo, mediante la expresión de manipulación de bits  $b' = b-LSOne(b)$ . La iteración de esta manipulación de bits *elimina* el bit menos significativo de  $b$  en cada paso. Como el entero  $b$  solo tiene  $O(\log b)$  bits, `rsq(b)` se ejecuta en tiempo  $O(\log n)$ , cuando  $b = n$ . En la figura 2.11,  $rsq(6) = ft[6]+ft[4] = 5+2 = 7$ . Hay que darse cuenta de que los índices 4 y 6 son responsables de los rangos  $[1..4]$  y  $[5..6]$ , respectivamente. Al combinarlos, calculamos el rango completo de  $[1..6]$ . Los índices 6, 4 y 0 están relacionados en su forma binaria:  $b = 6_{10} = (110)_2$  se puede transformar en  $b' = 4_{10} = (100)_2$  y, posteriormente, en  $b'' = 0_{10} = (000)_2$ .

---

<sup>31</sup>Hemos decidido seguir la implementación original de [18], que ignora el índice 0, para facilitar una comprensión más sencilla de las operaciones de manipulación de bits del árbol de Fenwick. Esto se debe a que el índice 0 no tiene ningún bit activado. Por ello, la operación  $i +/- LSOne(i)$ , devuelve  $i$  cuando  $i = 0$ , y provocaría un bucle infinito si el programador no ha sido cuidadoso con este caso extremo clásico. El índice 0 se utiliza también como condición de terminación en la función `rsq`.

<sup>32</sup>No es objeto de este libro detallar por qué esta disposición funciona y, en su lugar, mostraremos que permite realizar operaciones de actualización y RSQ eficientes en  $O(\log n)$ . Si el lector está interesado, le recomendamos nuevamente la lectura de [18].

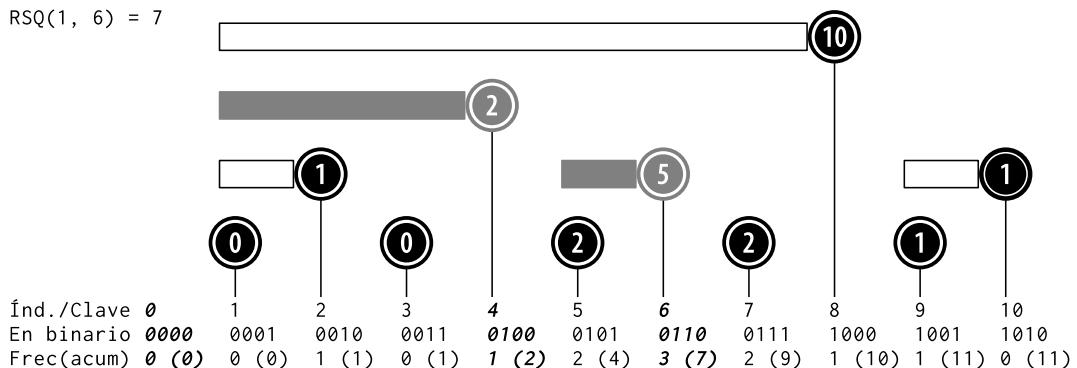


Figura 2.11: Ejemplo de rsq(6)

Teniendo  $\text{rsq}(b)$  disponible, obtener la frecuencia acumulada entre los índices  $[a..b]$ , donde  $a \neq 1$ , es sencillo, basta con evaluar  $\text{rsq}(a, b) = \text{rsq}(b) - \text{rsq}(a-1)$ . Por ejemplo, si queremos calcular  $\text{rsq}(4, 6)$ , es suficiente con devolver  $\text{rsq}(6) - \text{rsq}(3) = (5+2) - (0+1) = 7 - 1 = 6$ . Una vez más, esta operación se ejecuta en tiempo  $O(2 \times \log b) \approx O(\log n)$ , cuando  $b = n$ . La figura 2.12 muestra el valor de  $\text{rsq}(3)$ .

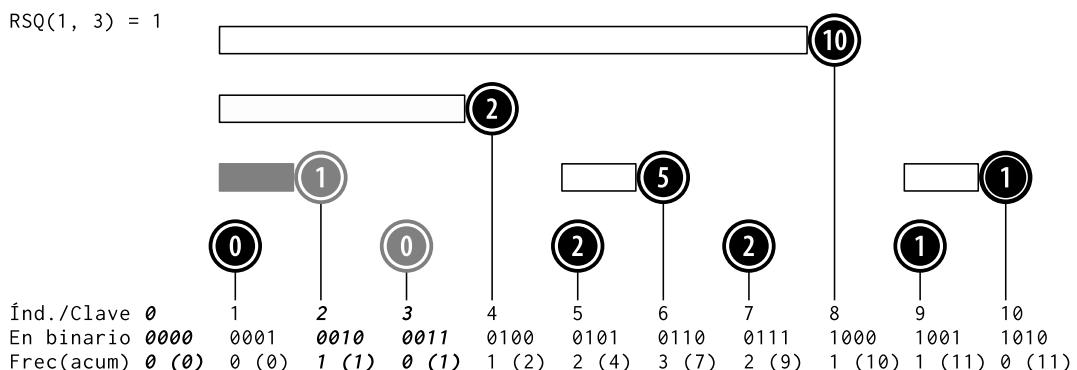


Figura 2.12: Ejemplo de rsq(3)

Cuando actualizamos el valor del elemento en el índice  $k$ , ajustando su valor por  $v$  (sabiendo que  $v$  puede ser positivo o negativo), es decir, cuando llamamos a  $\text{adjust}(k, v)$ , debemos actualizar  $\text{ft}[k]$ ,  $\text{ft}[k']$ ,  $\text{ft}[k'']$ , ..., hasta que el índice  $k^i$  supere a  $n$ , porque estos son los índices afectados. Esta secuencia de índices se obtiene con la siguiente expresión de manipulación de bits iterativa:  $k' = k + \text{LSOne}(k)$ . Desde cualquier  $k$  entero, la operación  $\text{adjust}(k, v)$  realizará un máximo de  $O(\log n)$  pasos hasta  $k > n$ , ya que estos son los índices afectados. En la figura 2.13,  $\text{adjust}(5, 1)$  afectará (sumará +1) a  $\text{ft}[k]$  en los índices  $k = 5_{10} = (101)_2$ ,  $k' = (101)_2 + (001)_2 = (110)_2 = 6_{10}$  y  $k'' = (110)_2 + (010)_2 = (1000)_2 = 8_{10}$ , a través de la expresión mostrada antes. Hay que fijarse en que, si proyectamos una línea hacia arriba desde el índice 5, en la figura 2.13, podremos ver cómo ésta *intersecciona* los rangos bajo la responsabilidad de los índices 5, 6 y 8.

En definitiva, el árbol de Fenwick soporta operaciones tanto RSQ como de actualización en espacio  $O(n)$  y tiempo  $O(\log n)$ , dado un conjunto de  $m$  claves enteras en el rango  $[1..n]$ .

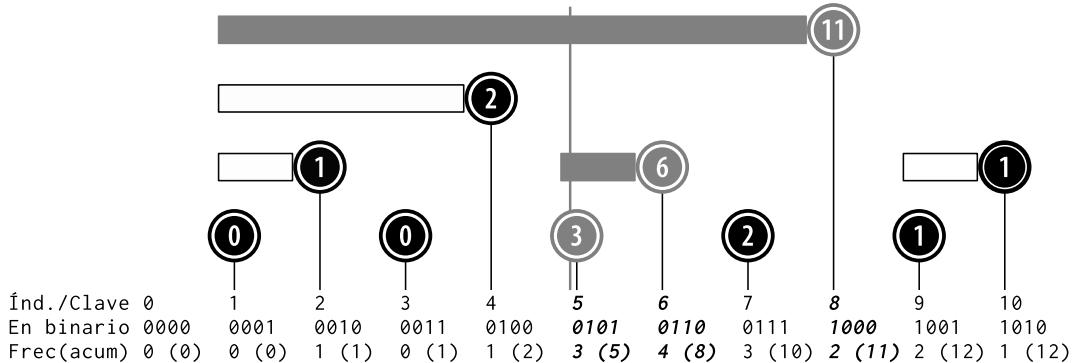


Figura 2.13: Ejemplo de `adjust(5, 1)`

Esto hace que el árbol de Fenwick sea una estructura de datos perfecta para resolver problemas de RSQ *dinámica* en *arrays* discretos (el problema de la RSQ *estática* se puede resolver con un procesamiento previo sencillo en  $O(n)$  y  $O(1)$  por consulta, como hemos visto antes). A continuación, se incluye nuestra *breve* implementación en C++ de un árbol de Fenwick básico:

```

1 class FenwickTree {
2 private: vi ft; // recuerda que vi es: typedef vector<int> vi;
3 public: FenwickTree(int n) { ft.assign(n+1, 0); } // inicializar n+1 ceros
4 int rsq(int b) { // devuelve RSQ(1, b)
5 int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
6 return sum; } // nota: LSOne(S) (S & (-S))
7 int rsq(int a, int b) { return rsq(b)-rsq(a-1); } // devuelve RSQ(a, b)
8 // ajusta el valor del elemento k-ésimo con v (v es +/inc o -/dec)
9 void adjust(int k, int v) { // nota: n = ft.size()-1
10 for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
11 };
12 int main() {
13 int f[] = { 2,4,5,5,6,6,6,7,7,8,9 }; // m = 11 puntuaciones
14 FenwickTree ft(10); // declarar árbol de Fenwick para el rango [1..10]
15 // insertar puntuaciones una por una a mano en un árbol de Fenwick vacío
16 for (int i = 0; i < 11; i++) ft.adjust(f[i], 1); // esto es O(m log n)
17 printf("%d\n", ft.rsq(1, 1)); // 0 => ft[1] = 0
18 printf("%d\n", ft.rsq(1, 2)); // 1 => ft[2] = 1
19 printf("%d\n", ft.rsq(1, 6)); // 7 => ft[6] + ft[4] = 5 + 2 = 7
20 printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11
21 printf("%d\n", ft.rsq(3, 6)); // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
22 ft.adjust(5, 2); // demostración de actualización
23 printf("%d\n", ft.rsq(1, 10)); // ahora 13
24 } // return 0;

```

[visualgo.net/fenwicktree](http://visualgo.net/fenwicktree)



ch2\_10\_fenwicktree\_ds.cpp



ch2\_10\_fenwicktree\_ds.java

### Ejercicio 2.4.4.1

Un ejercicio muy sencillo sobre las dos operaciones de bits básicas del árbol de Fenwick: ¿cuáles son los valores de  $90 - \text{LSOne}(90)$  y  $90 + \text{LSOne}(90)$ ?

### Ejercicio 2.4.4.2

¿Qué pasa si el problema que quieras resolver incluye un elemento en la clave entera 0? Recuerda que el rango de claves enteras estándar de nuestra biblioteca es  $[1..n]$ , y que esta implementación no puede utilizar el índice 0, ya que se usa como condición de terminación de `rsq`.

### Ejercicio 2.4.4.3

¿Y si el problema que quieras resolver utiliza claves no enteras? Por ejemplo, ¿cuál es la situación si las puntuaciones de exámenes de la tabla 2.1 fuesen  $f = \{5.5, 7.5, 8.0, 10.0\}$  (es decir, se permite un 0 o un 5 después del punto decimal)? ¿Y cuál es si las puntuaciones son  $f = \{5.53, 7.57, 8.10, 9.91\}$  (es decir, se permiten dos dígitos después del punto decimal)?

### Ejercicio 2.4.4.4

El árbol de Fenwick permite una operación adicional, que hemos decidido dejar como ejercicio para el lector: encontrar el índice más pequeño con una frecuencia acumulada dada. Por ejemplo, puede que necesitemos determinar la puntuación/índice mínima  $i$  en la tabla 2.1, de forma que haya, al menos, 7 estudiantes cubiertos por el rango  $[1..i]$  (puntuación/índice 6, en este caso). Implementa esta característica.

### Ejercicio 2.4.4.5\*

Resuelve el problema de RSQ dinámica UVa 12086 - Potentiometers, utilizando tanto un árbol de segmentos como un árbol de Fenwick. ¿Qué solución es más sencilla de implementar en este caso? Ver también la comparativa de la tabla 2.2.

### Ejercicio 2.4.4.6\*

Extiende el árbol de Fenwick unidimensional a uno bidimensional.

## Ejercicio 2.4.4.7\*

Los árboles de Fenwick se utilizan, normalmente, para actualizaciones de puntos y consultas (de suma) de rangos. Muestra cómo utilizar un árbol de Fenwick para realizar *actualizaciones de rangos* y consultas de puntos. Por ejemplo, dados muchos intervalos con rangos pequeños (desde 1 hasta un máximo de un millón), determina el número de intervalos que incluyen el índice  $i$ .

| Característica                        | Árbol de segmentos | Árbol de Fenwick |
|---------------------------------------|--------------------|------------------|
| Construir árbol desde <i>array</i>    | $O(n)$             | $O(m \log n)$    |
| RMin/MaxQ dinámicas                   | Sí                 | Muy limitado     |
| RSQ dinámica                          | Sí                 | Sí               |
| Complejidad de consulta               | $O(\log n)$        | $O(\log n)$      |
| Complejidad de actualización de punto | $O(\log n)$        | $O(\log n)$      |
| Longitud del código                   | Largo              | Corto            |

Tabla 2.2: Comparativa entre el árbol de segmentos y el árbol de Fenwick

### Perfiles de los inventores de estructuras de datos

**Peter M. Fenwick** es profesor asociado honorífico en la Universidad de Auckland. Inventó el árbol binario indexado en 1994 [18] como “tablas de frecuencia acumulada de compresión aritmética”. Desde entonces, el BIT ha sido incluido en el temario de la IOI [20] y utilizado en muchos concursos de programación, dada su estructura de datos eficiente y, al mismo tiempo, fácil de implementar.

## Ejercicios de programación

Ejercicios de programación que utilizan las estructuras de datos tratadas e implementadas:

### Problemas de estructuras de datos de grafos

1. [UVa 00599 - The Forrest for the Trees](#) \* ( $V - E$  = número de CC; usar un `bitset` de tamaño 26 para contar el número de vértices que tienen alguna arista) (trasponer lista de adyacencia)
2. [UVa 10895 - Matrix Transpose](#) \*
3. [UVa 10928 - My Dear Neighbours](#) (contar grados de salida)
4. [UVa 11550 - Demanding Dilemma](#) (DS de grafo; matriz de incidencia)
5. [UVa 11991 - Easy Problem from ...](#) \* (usar la idea de la lista de adyacencia)

Ver también más problemas de grafos en el capítulo 4.

### Conjuntos disjuntos para unión-buscar

1. [UVa 00793 - Network Connections](#) \* (trivial; aplicación de conjuntos disjuntos)
2. [UVa 01197 - The Suspects](#) (LA 2817 - Kaohsiung03; componentes conexos)
3. [UVa 10158 - War](#) (uso avanzado de conjuntos disjuntos con un giro interesante; memorizamos la lista de enemigos)
4. [UVa 10227 - Forests](#) (combinar dos conjuntos disjuntos si son compatibles)

- 5. **UVa 10507 - Waking up brain \***  
 (los conjuntos disjuntos simplifican este problema)
- 6. UVa 10583 - Ubiquitous Religions  
 (contar conjuntos disjuntos después de todas las uniones)
- 7. UVa 10608 - Friends  
 (encontrar el conjunto con el elemento más grande)
- 8. UVa 10685 - Nature  
 (encontrar el conjunto con el elemento más grande)
- 9. **UVa 11503 - Virtual Friends \***  
 (mantener el atributo del conjunto (tamaño) en el elemento rep)
- 10. UVa 11690 - Money Matters  
 (comprobar si el dinero total de cada miembro es 0)

### Estructuras de datos de árboles

- 1. UVa 00297 - Quadtrees  
 (problema sencillo de árbol *quad*)
- 2. UVa 01232 - SKYLINE  
 (LA 4108 - Singapore07; tenemos que usar un árbol de segmentos; este problema no es sobre RSQ/RMQ)
- 3. **UVa 11235 - Frequent Values \***  
 (consulta de rango máximo)  
 (árbol *quad* con actualizaciones o árbol de segmentos bidimensional)
- 4. UVa 11297 - Census  
 (pregunta sobre estructura de datos de árbol simple)
- 5. UVa 11350 - Stern-Brocot Tree  
 (árbol de segmentos con actualizaciones *perezosas*)
- 6. **UVa 11402 - Ahoy, Pirates \***  
 (LA 2191 - Dhaka06; problema de RSQ dinámica puro; se resuelve con árbol de Fenwick o árbol de segmentos)
- 7. UVa 12086 - Potentiometers  
 (uso inteligente de árbol de Fenwick/segmentos)
- 8. **UVa 12532 - Interval Product \***

Ver también: DS como parte de la solución de problemas difíciles en el capítulo 8.

## 2.5 Soluciones a los ejercicios no resaltados

**Ejercicio 2.2.1\***: subpregunta 1: ordenar primero  $S$  en  $O(n \log n)$  y, a continuación, realizar un barrido lineal en  $O(n)$ , comenzando desde el segundo elemento, para comprobar si un entero y el anterior son iguales (leer también la solución del **ejercicio 1.2.10**, tarea 4). Subpregunta 6: leer el primer párrafo del capítulo 3 y la discusión detallada de la sección 9.29. No incluimos las soluciones a las otras subpreguntas.

**Ejercicio 2.2.2**: las respuestas (salvo para las subpreguntas 7 y 8):

1.  $S \& (N - 1)$
2.  $(S \& (S - 1)) == 0$
3.  $S \& (S - 1)$
4.  $S \mid (S + 1)$
5.  $S \& (S + 1)$
6.  $S \mid (S - 1)$

**Ejercicio 2.3.1**: como la colección es dinámica, encontraremos consultas de inserción y eliminación frecuentes. Una inserción puede, potencialmente, cambiar el orden. Si almacenamos la información en un *array* estático, tendremos que utilizar una iteración  $O(n)$  de ordenación, después de cada inserción y eliminación (para eliminar el espacio vacío en el *array*). Ineficiente.

### Ejercicio 2.3.2:

1. `search(71)`: raiz (15) → 23 → 71 (encontrado)  
`search(7)`: raiz (15) → 6 → 7 (encontrado)  
`search(22)`: raiz (15) → 23 → subárbol izquierdo vacío (no encontrado).
2. En algún momento tendremos el mismo BST de la figura 2.2.
3. Para encontrar el elemento mínimo/máximo, podemos comenzar desde la raiz y desplazarnos a la izquierda/derecha, hasta que demos con un vértice que no tenga subárboles izquierdo/derecho, respectivamente. Ese vértice es la respuesta.
4. Obtendremos la salida ordenada: 4, 5, 6, 7, 15, 23, 50, 71. Consulta la sección 4.7.2 si no estás familiarizado con el algoritmo de recorrido de árboles inorden.
5. `successor(23)`: encontrar el elemento mínimo del subárbol con raiz a la derecha de 23, que es el subárbol con raiz en 71. La respuesta es 50.  
`successor(7)`: 7 no tiene subárbol derecho, así que debe ser el máximo de algún subárbol. Ese subárbol es el que tiene la raiz en 6. El padre de 6 es 15 y 6 es el subárbol izquierdo de 15. Según la propiedad del BST, 15 debe ser el sucesor de 7.  
`successor(71)`: 71 es el elemento más grande y no tiene sucesor.  
Nota: el algoritmo para encontrar el predecesor de un nodo es similar.
6. `delete(5)`: basta con eliminar 5, que es una hoja, del BST.  
`delete(71)`: como 71 es un vértice interno con un hijo, no podemos eliminarlo sin más, ya que provocaríamos la desconexión del BST en *dos* componentes. Podemos, en su lugar, reorganizar el subárbol con raiz en el padre de 71 (que es 23), causando que 23 tenga a 50 como su hijo derecho.
7. `delete(15)`: como 15 es un vértice con dos hijos, no podemos eliminarlo sin más, ya que provocaríamos la desconexión del BST en *tres* componentes. Para abordar el asunto, necesitamos encontrar al sucesor de 15 (que es 23) y utilizarlo para sustituir a 15. Entonces, podemos borrar el antiguo 23 del BST (ya no es un problema). Como apunte, también podemos utilizar `predecessor(clave)` en vez de `successor(clave)`, durante `delete(clave)`, en el caso de que la clave tenga dos hijos.

**Ejercicio 2.3.3\***: en la subtarea 1, ejecutamos un recorrido inorden en  $O(n)$  y vemos si los valores están ordenados. No incluimos las soluciones al resto de subtareas.

**Ejercicio 2.3.6**: las respuestas:

1. `Insert(26)`: insertar 26 como subárbol izquierdo de 3, intercambiar 26 con 3, después intercambiar 26 con 19 y parar. Después de la operación, el *array* de montículo máximo A contendrá  $\{-, 90, 26, 36, 17, 19, 25, 1, 2, 7, 3\}$ .
2. `ExtractMax()`: intercambiar 90 (elemento máximo que aparecerá después de ajustar la propiedad de montículo máximo) con 3 (la hoja actual más abajo y más a la derecha o, también, el último elemento del montículo máximo), intercambiar 3 con 36, intercambiar 3 con 25 y parar. El *array* de montículo máximo A contendrá entonces  $\{-, 36, 26, 25, 17, 19, 3, 1, 2, 7\}$ .

**Ejercicio 2.3.7**: sí, comprobar que los índices cumplen la propiedad de montículo máximo.

**Ejercicio 2.3.16:** usar `set` de la STL de C++ (o `TreeSet` de Java), ya que es un BST equilibrado con inserciones y eliminaciones dinámicas en  $O(\log n)$ . Podemos utilizar el recorrido inorden para mostrar la información del BST ordenada (`iterator` de C++ o `Iterator` de Java).

**Ejercicio 2.3.17:** usar `map` de la STL de C++ (`TreeMap` de Java) y una variable contador. También se puede usar una tabla de *hash*, pero no es necesario en un concurso de programación. Este truco se utiliza habitualmente en varios problemas de concursos. Ejemplo de uso:

```
1 char str[1000];
2 map<string, int> mapper;
3 int i, idx;
4 for (i = idx = 0; i < M; i++) { // el índice empieza en 0
5 scanf("%s", &str);
6 if (mapper.find(str) == mapper.end()) // si es la primera aparición
7 // también podemos comprobar si mapper.count(str) es mayor que 0
8 mapper[str] = idx++; // darle a str el índice actual e incrementar idx
9 }
```

**Ejercicio 2.4.1.3:** el grafo no es dirigido.

**Ejercicio 2.4.1.4\***: subtarea 1: para contar el número de vértices de un grafo: matriz de adyacencia/lista de adyacencia → mostrar el número de filas; lista de aristas → contar el número de vértices distintos en todas las aristas. Para contar el número de aristas de un grafo: matriz de adyacencia → suma del número de todas las entradas distintas de cero en cada fila; lista de adyacencia → suma de la longitud de todas las listas; lista de aristas → basta con mostar el número de filas. No incluimos las soluciones al resto de subtareas.

**Ejercicio 2.4.2.1:** para `int numDisjointSets()`, utilizar un contador entero adicional `numSets`. Inicialmente, durante `UnionFind(N)`, fijar `numSets = N`. Después, durante `unionSet(i, j)`, decrementar `numSets` en uno si `isSameSet(i, j)` devuelve falso. Ahora, basta con que `int numDisjointSets()` devuelva el valor de `numSets`.

Para `int sizeOfSet(int i)`, utilizamos otro vi `setSize(N)`, incializado a unos (cada conjunto tiene solo un elemento). Durante `unionSet(i, j)`, actualizamos el array `setSize`, ejecutando `setSize[find(j)] += setSize[find(i)]` (o al revés, según el rango), si `isSameSet(i, j)` devuelve falso. Basta con que `int sizeOfSet(int i)` nos dé el valor de `setSize[find(i)]`.

Estas dos variantes están implementadas en `ch2_08_unionfind_ds.cpp/java`.

**Ejercicio 2.4.3.3:**  $\text{RSQ}(1, 7) = 167$  y  $\text{RSQ}(3, 8) = 139$ .

**Ejercicio 2.4.4.1:**  $90 - \text{LSOne}(90) = (1011010)_2 - (10)_2 = (1011000)_2 = 88$  y  $90 + \text{LSOne}(90) = (1011010)_2 + (10)_2 = (1011100)_2 = 92$ .

**Ejercicio 2.4.4.2:** sencillo: incrementar todos los índices en uno. El índice  $i$  del árbol de Fenwick basado en 1 hará referencia ahora al índice  $i - 1$  en el problema real.

**Ejercicio 2.4.4.3:** sencillo: convertir los números de coma flotante en enteros. En la primera tarea podemos multiplicar cada número por 2. En la segunda, podemos multiplicarlos por 100.

**Ejercicio 2.4.4.4:** la frecuencia acumulada está ordenada, por lo que podemos utilizar una *búsqueda binaria*. Estudiar la técnica ‘búsqueda binaria de la respuesta’, tratada en la sección 3.3. La complejidad de tiempo resultante es  $O(\log^2 n)$ .

## 2.6 Notas del capítulo

Las estructuras de datos básicas de las secciones 2.2 y 2.3 aparecen en prácticamente cualquier libro de texto sobre estructuras de datos y algoritmos. Las referencias a las bibliotecas de C++/Java están en [www.cppreference.com](http://www.cppreference.com) y [java.sun.com/javase/7/docs/api](http://java.sun.com/javase/7/docs/api). Aunque, generalmente, se permite el acceso a estas páginas web durante los concursos de programación, sugerimos que trates de dominar la sintaxis de las operaciones más comunes, para minimizar el tiempo de programación.

Una excepción es el *conjunto ligero de booleanos* (la máscara de bits). Esta técnica *poco habitual* no se suele enseñar en los cursos de estructuras de datos y algoritmos, pero es fundamental para un programador competitivo, ya que logra mejoras de velocidad significativas, si se aplica a ciertos problemas. En este libro mencionamos esta estructura de datos en varias ocasiones, por ejemplo, en algunas rutinas iterativas de fuerza bruta y de *backtracking* optimizado (secciones 3.2.2 y 8.2.1), TSP con DP (sección 3.5.2) y DP con máscara de bits (sección 8.3.1). Todas utilizan máscaras de bits en vez de `vector<boolean>` o `bitset<size>`, debido a su eficiencia. Recomendamos la lectura del libro “Hacker’s Delight” [69], que trata la manipulación de bits con más detalle.

Indicamos algunas referencias adicionales para las estructuras de datos mencionadas en la sección 2.4. Para grafos, ver [58] y los capítulos 22 a 26 de [7]. Para conjuntos disjuntos para unión-buscar, ver el capítulo 21 de [7]. Para árboles de segmentos y otras estructuras de datos geométricas, ver [9]. Para el árbol de Fenwick, ver [30]. Recordamos que todas nuestras implementaciones de estructuras de datos, tratadas en la sección 2.4, evitan el uso de punteros y, en su lugar, utilizamos *arrays* o vectores.

Con más experiencia, y leyendo el código fuente que hemos incluido, podrás dominar más trucos en la aplicación de estas estructuras de datos. Dedica tiempo a explorar el código fuente proporcionado por este libro en [sites.google.com/site/stevenhalim/home/material](http://sites.google.com/site/stevenhalim/home/material).

En este libro se tratan más estructuras de datos, como las específicas para cadenas (**trie, array o árbol de sufijos**) en la sección 6.6. Aun así, existen muchas más que resulta imposible abordar. Si quieras mejorar en los concursos de programación, investiga otras técnicas de estructuras de datos, diferentes a las que hemos incluido. Por ejemplo, los **árboles AVL**, **árboles Red Black** o, incluso, los **árboles biselados**, son útiles para ciertos problemas que requieren implementar y aumentar (añadir más información a) BST equilibrados (ver la sección 9.29). Los **árboles de intervalos** (que son similares a los árboles de segmentos) y los **árboles quad** (para particionar el espacio bidimensional), son también útiles, ya que los conceptos que aportan pueden ayudar a resolver ciertos problemas de los concursos.

Muchas de las estructuras de datos mostradas en el libro se basan en la estrategia ‘divide y vencerás’ (tratada en la sección 3.3).



## Capítulo 3

---

# Paradigmas de resolución de problemas

*Si lo único que tienes es un martillo, todo te parece un clavo.*

— Abraham Maslow, 1962

### 3.1 Introducción y motivación

En este capítulo trataremos *cuatro* paradigmas de resolución de problemas, utilizados habitualmente para resolver problemas en los concursos de programación, concretamente, búsqueda completa (fuerza bruta), divide y vencerás, técnica voraz y programación dinámica. Todos los programadores competitivos, incluyendo los concursantes de la IOI y del ICPC, deberían dominar por completo estos métodos de resolución de problemas (además de otros), para ser capaces de abordar cada tipo concreto de problema usando la ‘herramienta’ apropiada. Triturar *cada* problema con soluciones de fuerza bruta, no es el camino para obtener un buen rendimiento en los concursos. Para ilustrarlo, utilizaremos cuatro tareas sencillas, que incluyen un *array* A que contiene  $n \leq 10K$  enteros pequeños  $\leq 100K$  (por ejemplo,  $A = \{10, 7, 3, 5, 8, 2, 9\}$ ,  $n = 7$ ) para obtener una idea general de lo que ocurre si tratamos de resolver todos los problemas utilizando la fuerza bruta como único recurso:

1. Encuentra los elementos mayor y menor de A (10 y 2 en el ejemplo).
2. Encuentra el elemento  $k$ -ésimo menor de A (si  $k = 2$ , la respuesta es 3 en el ejemplo).
3. Encuentra la distancia máxima  $g$  de forma que  $x, y \in A$  y  $g = |x - y|$  (8 en el ejemplo).
4. Encuentra la subsecuencia creciente más larga de A ( $\{3, 5, 8, 9\}$  en el ejemplo).

La solución a la primera tarea es sencilla: leer cada elemento de A y comprobar si es el elemento más grande (o más pequeño) visto hasta el momento. Esta es una solución de **búsqueda completa** de complejidad  $O(n)$ .

La segunda tarea ya es un poco más difícil. Podemos utilizar la solución anterior para encontrar el valor más pequeño y sustituirlo con un valor grande (por ejemplo,  $1M$ ) para ‘anularlo’. Despues, podemos proceder a buscar nuevamente el valor más pequeño (el segundo más pequeño en el *array* original) y sustituirlo por  $1M$ . Repitiendo este proceso  $k$  veces, encontraremos el valor  $k$ -ésimo más pequeño. Esta solución funciona, pero si  $k = \frac{n}{2}$  (la mediana), utilizando búsqueda completa se ejecutará en  $O(\frac{n}{2} \times n) = O(n^2)$ . En su lugar, podemos ordenar el *array* A en  $O(n \log n)$ , devolviendo la respuesta simplemente como  $A[k-1]$ . Sin embargo, una solución

mejor para un número pequeño de consultas es la de complejidad  $O(n)$  mostrada en la sección 9.29. Las dos soluciones, de complejidades  $O(n \log n)$  y  $O(n)$ , mencionadas, son del tipo **divide y vencerás**.

Igualmente, en la tercera tarea podemos considerar todas las parejas de enteros  $x$  e  $y$  posibles de  $A$ , comprobando si la distancia entre ellas es la mayor hasta el momento. Este abordaje de búsqueda completa se ejecuta con complejidad  $O(n^2)$ . Funciona, pero es lento y poco eficaz. Podemos demostrar que  $g$  se puede obtener buscando la diferencia entre los elementos menor y mayor de  $A$ . Estos dos enteros se pueden localizar con la solución de la primera tarea, con complejidad  $O(n)$ . Ninguna otra combinación de dos enteros en  $A$  puede tener una distancia mayor. Ésta es una solución **voraz**.

Para la cuarta tarea, intentar todas las posibles subsecuencias con complejidad  $O(2^n)$ , para encontrar la creciente más larga, no es practicable para todos los  $n \leq 10K$ . En la sección 3.5.2, veremos una solución sencilla de **programación dinámica**, con complejidad  $O(n^2)$  y otra voraz, más rápida, en el orden  $O(n \log k)$ .

## 3.2 Búsqueda completa

La técnica de búsqueda completa, conocida también como fuerza bruta o *backtracking* recursivo, es un método para resolver problemas recorriendo la totalidad (o parte) del espacio de búsqueda, y obtener así la solución requerida. Durante la búsqueda, podemos podar (es decir, decidir no explorar) áreas del espacio de búsqueda, si hemos determinado que no existe la posibilidad de encontrar la respuesta requerida en ellas.

Durante un concurso de programación, el concursante *debería* desarrollar una solución de búsqueda completa cuanto sea evidente que no hay otro algoritmo disponible (por ejemplo, es obvio que la tarea de enumerar *todas* las permutaciones de  $\{0, 1, 2, \dots, N - 1\}$  requiere  $O(N)$  operaciones) o cuando existen algoritmos mejores, pero son *excesivos* si la entrada resulta ser pequeña (el problema en el que hay que responder consultas de rango mínimo en la sección 2.4.3 se resuelve con un bucle  $O(N)$  para cada consulta, siempre que  $N \leq 100$ ).

En el ICPC, la búsqueda completa debería ser la primera solución a considerar, ya que será normalmente fácil de desarrollar y depurar. Hay que tener presente el principio de que conviene buscar soluciones cortas y sencillas. Una solución de búsqueda completa *libre de errores*, no debería recibir *nunca* un veredicto de respuesta incorrecta (WA) en un concurso de programación, ya que explora el espacio de búsqueda *completo*. Muchos problemas de programación tienen soluciones mejores, como se ilustra en la sección 3.1. Sin embargo, una solución de búsqueda completa puede terminar con un veredicto de tiempo límite superado (TLE). Con el análisis adecuado, se puede determinar el resultado más probable (TLE frente a AC), antes de iniciar la implementación (la tabla 1.4, en la sección 1.2.3, es un buen punto de partida). Si la solución de búsqueda completa tiene posibilidades de ejecutarse dentro del límite de tiempo, ésta debería de ser la opción prioritaria. De esta forma, tendremos tiempo para dedicar a problemas más complejos, en los que la búsqueda completa sea demasiado lenta.

En la IOI, necesitarás, normalmente, mejores técnicas de resolución de problemas, ya que las soluciones de búsqueda completa serán recompensadas solo con pequeñas fracciones de la puntuación total correspondiente a la subtarea. En cualquier caso, habrá que tener en cuenta la búsqueda completa, siempre que no se pueda encontrar una solución mejor ya que, al menos, se podrá incrementar la puntuación.

En ocasiones, ejecutar soluciones de búsqueda completa en *pequeños segmentos* de un problema complicado, puede ayudarnos a entender su estructura, a través de los patrones que podamos encontrar en su salida (en algunos problemas, es posible *visualizar* un patrón en la solución), lo que nos ayudará a encontrar un algoritmo más rápido. Algunos problemas de combinatoria, de la sección 5.4, se pueden resolver así. Además, la solución de búsqueda completa puede servir como forma de verificar otros algoritmos, lo que supone una comprobación adicional para otros algoritmos más rápidos, pero también más complejos, que podamos desarrollar.

Al finalizar la lectura de esta sección, puede que obtengas la impresión de que la búsqueda completa solo es útil para los ‘problemas fáciles’, y no debería tenerse en cuenta en los ‘difíciles’. Esto no es estrictamente cierto. Hay problemas difíciles que solo se pueden resolver mediante algoritmos de búsqueda completa. Hemos reservado esos problemas para la sección 8.2.

En las dos subsecciones siguientes, mostramos varios ejemplos (*fáciles*) de este sencillo, a la vez que desafiante, paradigma. En la sección 3.2.1 se pueden encontrar ejemplos que están implementados de forma *iterativa*. En la sección 3.2.2, los ejemplos son sobre soluciones implementadas *recursivamente* (con *backtracking*). Por último, en la sección 3.2.3, se pueden encontrar algunos consejos para que la solución, especialmente si es de búsqueda completa, tenga más posibilidades de superar el límite de tiempo requerido.

### 3.2.1 Búsqueda completa iterativa

#### Búsqueda completa iterativa (dos bucles anidados: UVa 725 - Division)

Enunciado resumido del problema: encontrar y mostrar todas las parejas de números de 5 dígitos que, colectivamente, utilicen los dígitos del 0 al 9 sin repetición, de forma que el primer número dividido por el segundo resulte en un entero  $N$ , donde  $2 \leq N \leq 79$ . Es decir,  $abcde/fghij = N$ , donde cada letra representa un dígito diferente. El primer dígito de uno de los números puede ser el cero, por ejemplo, con  $N = 62$ , tenemos  $79546/01283 = 62$  o  $94736/01528 = 62$ .

Un análisis rápido nos muestra que  $fghij$  solo puede encontrarse en el rango de 01234 a 98765, lo que supone, como mucho,  $\approx 100K$  posibilidades. Un límite mejor para  $fghij$  es el rango 01234 a  $98765/N$ , lo que serán, como mucho,  $\approx 50K$  posibilidades para  $N = 2$ , y se irá reduciendo a medida que  $N$  aumente. Para cada  $fghij$  que probemos, podemos obtener  $abcde$  a partir de  $fghij \times N$  y, después, comprobar si los 10 dígitos son diferentes. Esto se implementa a través de un doble bucle anidado con una complejidad temporal, como máximo, de  $\approx 50K \times 10 = 500K$  operaciones por cada caso de prueba. No es mucho, por lo tanto, es factible realizar una búsqueda completa iterativa. A continuación, se muestra la parte principal del código (con un truco de manipulación de bits mostrado en la sección 2.2, para determinar que cada dígito es único):

```
1 for (int fghij = 1234; fghij <= 98765/N; fghij++) {
2 int abcde = fghij*N; // así, abcde y fghij tienen un máximo de 5 dígitos
3 int tmp, used = (fghij < 10000); // si el dígito f=0, lo etiquetamos
4 tmp = abcde; while (tmp) { used |= 1 << (tmp%10); tmp /= 10; }
5 tmp = fghij; while (tmp) { used |= 1 << (tmp%10); tmp /= 10; }
6 if (used == (1<<10) - 1) // si se usan todos los dígitos, escribirlo
7 printf("%0.5d / %0.5d = %d\n", abcde, fghij, N);
8 }
```

## Búsqueda completa iterativa (muchos bucles anidados: UVa 441 - Lotto)

En los concursos, los problemas que se resuelven con un *único* bucle suelen estar en la categoría de *fáciles*. Los que requieren iteraciones con un anidado doble, como el UVa 725 - Division, mostrado antes, son algo más complejos, pero no necesariamente difíciles. Los programadores competitivos deben sentirse cómodos escribiendo código con *más de dos* bucles anidados.

Vamos a echarle un vistazo al problema UVa 441, que se resume así: dados  $6 < k < 13$  enteros, enumerar de forma ordenada todos los subconjuntos posibles de tamaño 6 de dichos enteros.

Como el tamaño del subconjunto requerido siempre será de 6, y la salida debe estar ordenada lexicográficamente (la entrada ya lo está), la solución más sencilla es utilizar *seis* bucles anidados, como vemos a continuación. Hay que tener en cuenta que, incluso en el caso de prueba más grande, donde  $k = 12$ , estos seis bucles solo producirán  ${}_{12}C_6 = 924$  líneas de salida. Es pequeño.

```
1 for (int i = 0; i < k; i++) scanf("%d", &S[i]); // k enteros ordenados
2 for (int a = 0 ; a < k-5; a++) // seis bucles anidados
3 for (int b = a+1; b < k-4; b++)
4 for (int c = b+1; c < k-3; c++)
5 for (int d = c+1; d < k-2; d++)
6 for (int e = d+1; e < k-1; e++)
7 for (int f = e+1; f < k ; f++)
8 printf("%d %d %d %d %d %d\n", S[a], S[b], S[c], S[d], S[e], S[f]);
```

## Búsqueda completa iterativa (bucle y poda: UVa 11565 - Simple Equations)

Enunciado resumido del problema: dados tres enteros  $A, B$  y  $C$  ( $1 \leq A, B, C \leq 10000$ ), hallar otros tres enteros distintos  $x, y$  y  $z$ , de forma que  $x+y+z = A$ ,  $x \times y \times z = B$  y  $x^2 + y^2 + z^2 = C$ .

La tercera ecuación,  $x^2 + y^2 + z^2 = C$ , es un buen punto de partida. Asumiendo que  $C$  será el valor más próximo a 10000, y que  $y$  y  $z$  son 1 y 2 ( $x, y$  y  $z$  tienen que ser distintos), el rango de valores posible para  $x$  es  $[-100 \dots 100]$ . Podemos utilizar el mismo razonamiento para obtener un rango similar para  $y$  y  $z$ . Después, escribiremos la siguiente solución iterativa, con un bucle anidado triple, que requiere  $201 \times 201 \times 201 \approx 8M$  operaciones por cada caso de prueba.

```
1 bool sol = false; int x, y, z;
2 for (x = -100; x <= 100; x++)
3 for (y = -100; y <= 100; y++)
4 for (z = -100; z <= 100; z++)
5 if (y != x && z != x && z != y && // los tres deben ser distintos
6 x+y+z == A && x*y*z == B && x*x + y*y + z*z == C) {
7 if (!sol) printf("%d %d %d\n", x, y, z);
8 sol = true; }
```

Cabe mencionar el uso de AND para acelerar la solución, al provocar una comprobación *ligera* de que  $x, y$  y  $z$  son diferentes entre sí, *antes* de ponernos con las fórmulas. El código anterior cumple con el tiempo límite requerido para este problema, pero podemos hacerlo mejor. También es posible utilizar la segunda ecuación,  $x \times y \times z = B$ , y asumir que  $x = y = z$  para obtener

$x \times x \times x < B$  o  $x < \sqrt[3]{B}$ . El nuevo rango de  $x$  será  $[-22 \dots 22]$ . Además, podemos seguir podando el espacio de búsqueda, utilizando sentencias condicionales `if`, para ejecutar solo algunos de los bucles (interiores), o utilizar `break/continue` para detener u omitir bucles. El código que mostramos a continuación, es mucho más rápido que el anterior (son necesarias más optimizaciones para resolver la versión más complicada de este problema, UVa 11571 - Simple Equations - Extreme!!):

```

1 bool sol = false; int x, y, z;
2 for (x = -22; x <= 22 && !sol; x++) if (x*x <= C)
3 for (y = -100; y <= 100 && !sol; y++) if (y != x && x*x + y*y <= C)
4 for (z = -100; z <= 100 && !sol; z++)
5 if (z != x && z != y &&
6 x+y+z == A && x*y*z == B && x*x + y*y + z*z == C) {
7 printf("%d %d %d\n", x, y, z);
8 sol = true; }
```

### Búsqueda completa iterativa (permutaciones: UVa 11742 - Social Constraints)

Enunciado resumido del problema: hay  $0 < n \leq 8$  espectadores en un cine. Se sientan en la primera fila, en  $n$  asientos libres consecutivos. Hay  $0 \leq m \leq 20$  restricciones de cómo pueden sentarse, por ejemplo, los espectadores **a** y **b** debe estar separados como mucho (o al menos) por **c** asientos. La pregunta es sencilla: ¿cuántas formas posibles de sentarlos existen?

La clave para resolver este problema es darnos cuenta de que debemos explorar **todas** las permutaciones (combinaciones de asientos) posibles. Una vez que hayamos llegado a esta conclusión, podemos deducir esta sencilla solución de ‘filtro’, de complejidad  $O(m \times n!)$ . Ponemos un contador a cero y probamos todas las  $n!$  permutaciones posibles. Vamos incrementando el contador cada vez que la permutación actual cumpla con todas las  $m$  restricciones. Cuando se hayan examinado las  $n!$  permutaciones, mostraremos el valor final del contador. Como el máximo de  $n$  es 8, y el de  $m$  es 20, el caso de prueba más grande solo requerirá  $20 \times 8! = 806400$  operaciones, lo que es una solución perfectamente viable.

Si nunca has escrito un algoritmo que genere todas las permutaciones de un conjunto de números (ver el **ejercicio 1.2.3**, tarea 7), puede que todavía no sepas cómo proceder. A continuación, incluimos un solución sencilla en C++.

```

1 #include <bits/stdc++.h> // next_permutation está en <algorithm> de la STL
2 // la rutina principal
3 int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7}; // primera permutación
4 do { // probar las O(n!) permutaciones posibles, entrada mayor 8! = 40320
5 ... // comprobar la restricción social basada en 'p' en O(m)
6 } // la complejidad de tiempo total es O(m * n!)
7 while (next_permutation(p, p+n));
```

### Búsqueda completa iterativa (subconjuntos: UVa 12455 - Bars)

Enunciado resumido del problema: dada una lista  $l$ , que contiene  $1 \leq n \leq 20$  enteros, ¿existe un subconjunto de la lista  $l$  cuya suma tenga como resultado el entero  $X$ ?

Podemos probar con todos los  $2^n$  subconjuntos posibles, sumar los enteros seleccionados de cada subconjunto con complejidad  $O(n)$ , y comprobar si el resultado es igual a  $X$ . Por lo tanto, la complejidad de tiempo del conjunto será de  $O(n \times 2^n)$ . En el caso de mayor tamaño, donde  $n = 20$ , esto significa  $20 \times 2^{20} \approx 21M$  de operaciones. Aunque es ‘grande’, resulta viable (por la razón explicada más adelante).

Si nunca has escrito un algoritmo que genere todos los subconjuntos<sup>1</sup> de un conjunto de números (ver el **ejercicio 1.2.3**, tarea 8), puede que no tengas claro cómo seguir. Una solución sencilla es utilizar la *representación binaria* de los enteros desde 0 hasta  $2^n - 1$ , para describir todos los subconjuntos posibles. Si no estás familiarizado con las técnicas de manipulación de bits, puedes consultar la sección 2.2. La solución se puede expresar de forma sencilla en C/C++, como incluimos a continuación (también funciona en Java). Como las operaciones de manipulación de bits son (muy) rápidas, los  $21M$  de operaciones necesarias para el caso más grande, se pueden realizar en menos de un segundo. Pero es posible una solución más rápida (sección 8.2.1).

```
1 // rutina principal, la variable 'i' (máscara de bits) ya se ha declarado
2 for (i = 0; i < (1<<n); i++) { // para cada subconjunto, O(2^n)
3 sum = 0;
4 for (int j = 0; j < n; j++) // comprobar pertenencia, O(n)
5 if (i & (1<<j)) // ¿está el bit 'j' activado en el subconjunto 'i'?
6 sum += 1[j]; // si lo está, procesar 'j'
7 if (sum == X) break; // respuesta encontrada: máscara de bits 'i'
8 }
```

### Ejercicio 3.2.1.1

En la solución de UVa 725, ¿por qué es mejor iterar sobre fghij que sobre abcde?

### Ejercicio 3.2.1.2

¿Funciona para UVa 725 un algoritmo 10! que permuta abcdefghij?

### Ejercicio 3.2.1.3\*

Java todavía *no tiene* una función `next_permutation` integrada. Si eres usuario de Java, escribe tu propia rutina de *backtracking* recursivo para generar todas las permutaciones de hasta 10 objetos ordenados. Este *backtracking* recursivo es similar al del problema 8-Queens de la sección 3.2.2.

<sup>1</sup>Este problema también es conocido como ‘suma de subconjuntos’, ver la sección 3.5.3.

### Ejercicio 3.2.1.4\*

¿Cómo se resuelve el problema UVa 12455 si  $1 \leq n \leq 30$  y cada entero puede tener un valor de hasta 1000000000? Consejo: ver la sección 8.2.4.

## 3.2.2 Búsqueda completa recursiva

### Backtracking sencillo: UVa 750 - 8 Queens Chess Problem

Enunciado resumido de problema: en el ajedrez (con un tablero de  $8 \times 8$  casillas), es posible colocar ocho reinas de forma que ninguna de ellas pueda atacar a otra. Determinar *todas* las posibilidades de que ocurra esto, con la posición de una de las reinas dada (las coordenadas  $(a, b)$  deben contener una reina). Expresar las posibilidades en orden lexicográfico (ordenadas).

La solución más ingenua es enumerar todas las combinaciones de 8 casillas diferentes de las  $8 \times 8 = 64$  posibles y comprobar si se pueden colocar en ellas las 8 reinas sin conflictos. Sin embargo, hay  $64C_8 \approx 4000M$  posibilidades, por lo que tal aproximación no merece ni un intento.

Una solución mejor, pero todavía ingenua, radica en el hecho de que en cada columna solo puede haber una reina, por lo que podemos proceder a partir de ahí. Esto reduce las posibilidades de los  $4000M$  a solo  $8^8 \approx 17M$ . Sigue siendo una solución que está al límite de lo aceptable. Si escribimos esta búsqueda completa, lo más probable es que acabemos con un veredicto de tiempo límite superado (TLE), especialmente si se nos plantean varios casos de prueba. Podemos utilizar más optimizaciones sencillas, como las descritas a continuación, para seguir reduciendo el espacio de búsqueda.

Sabemos que no puede haber dos reinas en la misma columna *o en la misma fila*. Con ello en mente, podemos seguir simplificando el problema original, hasta buscar las *permutaciones* válidas de 8! posiciones en filas. El valor de `row[i]` describe la posición de la reina en la columna *i*. Por ejemplo: `row = {1, 3, 5, 7, 2, 0, 6, 4}`, como muestra la figura 3.1, es una de las soluciones del problema; `row[0] = 1` implica que la reina de la columna 0 se encuentra en la fila 1, y así sucesivamente (el índice empieza en 0). Al modelarlo de esta forma, el espacio de búsqueda se *reduce* de  $8^8 \approx 17M$  a  $8! \approx 40K$ . Esta solución ya sería suficientemente rápida, pero podemos hacer más.

También sabemos que no puede haber dos reinas que compartan diagonales. Pongamos que la reina A está en  $(i, j)$  y la reina B en  $(k, l)$ . Se podrá atacar la una a la otra si  $\text{abs}(i-k) == \text{abs}(j-l)$ . Esta fórmula significa que las distancias vertical y horizontal entre las dos reinas son iguales, es decir, las reinas A y B se encuentran en una de las diagonales de la otra.

Una solución de *backtracking recursivo* sitúa a las reinas una por una en las columnas de 0 a 7, teniendo en cuenta las restricciones anteriores. Por último, si se encuentra una solución candidata, se comprueba si, al menos, una de las reinas cumple con los requisitos de la entrada, es decir `row[b] == a`. Esta solución de complejidad inferior a  $O(n!)$ , obtendrá un veredicto AC.

A continuación, mostramos nuestra implementación. Si nunca has escrito una solución de *backtracking* recursivo, analízala y, quizás, escríbelas en tu propio estilo de programación.

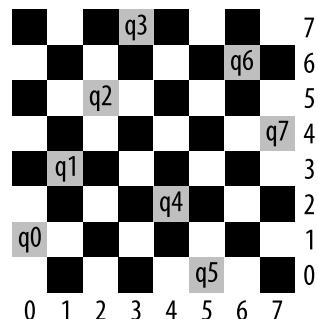


Figura 3.1: 8-Queens

```

1 #include <bits/stdc++.h> // utiliza la versión de enteros de 'abs'
2 using namespace std;
3
4 int row[8], TC, a, b, lineCounter; // es aceptable usar variables globales
5
6 bool place(int r, int c) {
7 for (int prev = 0; prev < c; prev++) // comprobar reinas ya colocadas
8 if (row[prev] == r || (abs(row[prev]-r) == abs(prev-c)))
9 return false; // comparten misma fila o diagonal -> inviable
10 return true; }
11 void backtrack(int c) {
12 if (c == 8 && row[b] == a) { // solución candidata, (a, b) tiene 1 reina
13 printf("%2d %d", ++lineCounter, row[0]+1);
14 for (int j = 1; j < 8; j++) printf(" %d", row[j]+1);
15 printf("\n");
16 return; }
17 for (int r = 0; r < 8; r++) // probar en todas las filas posibles
18 if (place(r, c)) // si se puede colocar en reina en esta fila y columna
19 row[c] = r, backtrack(c+1); // colocar esta reina y seguir recursión
20 }
21
22 int main() {
23 scanf("%d", &TC);
24 while (TC--) {
25 scanf("%d %d", &a, &b); a--; b--; // cambio a indexación desde 0
26 memset(row, 0, sizeof row); lineCounter = 0;
27 printf("SOLN COLUMN\n");
28 printf(" # 1 2 3 4 5 6 7 8\n\n");
29 backtrack(0); // generar las 8! soluciones candidatas posibles
30 if (TC) printf("\n");
31 } } // return 0;

```



ch3\_01\_UVa750.cpp



ch3\_01\_UVa750.java

### **Backtracking más complicado: UVa 11195 - Another n-Queen Problem**

Enunciado resumido del problema: dado un tablero de ajedrez de  $n \times n$  ( $3 < n < 15$ ), en el que hay casillas bloqueadas (no pueden colocarse reinas en ellas), ¿cuántas formas hay de colocar  $n$  reinas, de forma que no puedan atacarse entre ellas? Nota: las casillas bloqueadas *no pueden* utilizarse para proteger a las reinas.

El código de *backtracking* recursivo que hemos mostrado anteriormente, *no es* lo suficientemente rápido para  $n = 14$  sin casillas bloqueadas, que es el caso de prueba más complejo posible para este problema. La solución de complejidad inferior a  $O(n!)$  planteada sigue siendo válida para  $n = 8$ , pero no para  $n = 14$ . Tenemos que hacerlo mejor.

El principal problema con el código anterior, es que resulta demasiado lento comprobar si la posición de cada nueva reina es válida, ya que la comparamos con todas las  $c - 1$  anteriores (ver la función `bool place (int r, int c)`). Es mejor almacenar la misma información en tres *arrays* de booleanos (de momento, utilizaremos `bitset`):

```
bitset<30> rw, ld, rd; // para el n = 14 máximo, hay 27 diagonales
```

Inicialmente, todas las  $n$  filas (`rw`), las  $2 \times n - 1$  diagonales izquierdas (`ld`) y las  $2 \times n - 1$  diagonales derechas (`rd`), están sin utilizar (los tres `bitset` están establecidos como `false`). Cuando se coloca una reina en una casilla  $(r, c)$ , marcamos `rw[r] = true`, para que esta fila no pueda volver a utilizarse. Además, tampoco podrán volver a utilizarse todas las  $(a, b)$  donde  $\text{abs}(r-a) = \text{abs}(c-b)$ . Hay dos posibilidades si eliminamos la función `abs`:  $r-c = a-b$  y  $r+c = a+c$ . Debemos tener en cuenta que  $r+c$  y  $r-c$  representan los índices de los dos ejes diagonales. Como  $r-c$  puede ser negativo, añadimos un *desplazamiento* de  $n-1$  a ambos lados de la ecuación, de forma que  $r-c+n-1 = a-b+n-1$ . Si se coloca una reina en la casilla  $(r, c)$ , marcamos `ld[r-c+n-1] = true` y `rd[r+c] = true`, para desactivar las dos diagonales. Con estas estructuras de datos adicionales, y la restricción específica del problema UVa 11195 (`board[r][c]` no puede ser una casilla bloqueada), podemos modificar nuestro código para convertirlo en:

```
1 void backtrack(int c) {
2 if (c == n) { ans++; return; } // una solución
3 for (int r = 0; r < n; r++) // probar todas las filas posibles
4 if (board[r][c] != '*' && !rw[r] && !ld[r-c+n-1] && !rd[r+c]) {
5 rw[r] = ld[r-c+n-1] = rd[r+c] = true; // quitar etiqueta
6 backtrack(c+1);
7 rw[r] = ld[r-c+n-1] = rd[r+c] = false; // restaurar
8 }
}
```



[visualgo.net/recursion](http://visualgo.net/recursion)

### Ejercicio 3.2.2.1

El código incluido para el problema UVa 750 se puede optimizar más con una poda en la búsqueda, cuando '`row[b] != a`', en un momento *anterior* de la recursión (no solo cuando `c == 8`). Modificalo.

### Ejercicio 3.2.2.2\*

Por desgracia, la solución actualizada utilizando `bitset`: `rw`, `ld` y `rd`, seguirá obteniendo un veredicto TLE en el problema UVa 11195 - Another n-Queen Problem. Necesitamos acelerar más la solución, utilizando técnicas de máscaras de bits, y otra forma de utilizar las restricciones de las diagonales izquierda y derecha. Esta solución se tratará en la sección 8.2.1. De momento, utiliza la idea presentada (no AC) para el UVa 11195 y acelera el código del UVa 750 y otros dos problemas similares: UVa 167 y UVa 11085.

### 3.2.3 Consejos

La principal duda al escribir soluciones de búsqueda completa es si serán, o no, capaces de cumplir con el límite de tiempo. Si este fuese de 10 segundos (los jueces en línea no suelen utilizar tiempos tan grandes, por cuestiones de eficiencia), el programa se ejecuta en  $\approx 10$  segundos utilizando varios casos de prueba y el mayor tamaño de entrada especificado en el enunciado y, aun así, el veredicto es TLE, quizás sea una buena idea modificar el ‘código crítico’<sup>2</sup>, en vez buscar un nuevo algoritmo más rápido y que, quizás, sea difícil de diseñar.

Estos son algunos consejos a tener en cuenta a la hora de diseñar una solución de búsqueda completa para un problema dado, y que podrían ayudar a aumentar las posibilidades de que se ejecute dentro del tiempo permitido. Escribir una buena solución de búsqueda completa es un arte en sí mismo.

#### Consejo 1: filtrar frente a generar

Los programas que examinan muchas (cuando no todas) las soluciones candidatas y seleccionan aquellas que son correctas (o eliminan las incorrectas), se denominan ‘filtros’. Por ejemplo, las soluciones del problema de las ocho reinas, con complejidades de tiempo  $64C_8$  y  $8^8$ , la solución iterativa para los problemas UVa 725 y UVa 11742, etc. Normalmente, los programas ‘filtro’ están escritos de forma iterativa.

Los programas que van construyendo gradualmente las soluciones y podan inmediatamente las parciales que no son válidas, se llaman ‘generadores’. Por ejemplo, la solución mejorada del problema de las ocho reinas, con su complejidad inferior a  $O(n!)$ , junto a la comprobación de diagonales. Normalmente, los programas ‘generadores’ son más sencillos cuando se escriben de forma recursiva, lo que da una mayor flexibilidad a la hora de podar el espacio de búsqueda.

Generalmente, los filtros son más fáciles de programar, pero son lentos en su ejecución, ya que suele ser mucho más difícil podar el espacio de búsqueda iterativamente. Se debe hacer el cálculo (análisis de complejidad), para comprobar si un filtro resulta suficiente, o si es mejor implementar un generador.

#### Consejo 2: podar pronto el espacio de búsqueda impracticable/inferior

Al generar soluciones que utilicen *backtracking* recursivo (ver el consejo 1), podemos encontrar una solución parcial que nunca nos lleve a la solución completa. Es posible detener ahí el proceso y explorar otras partes del espacio de búsqueda. Por ejemplo: la comprobación diagonal en la solución al problema de las ocho reinas. Vamos a suponer que hemos colocado una reina en `row[0] = 2`. Colocar la siguiente en `row[1] = 1` o `row[1] = 3`, provocará un conflicto en la diagonal, y colocarla en `row[1] = 2`, lo provocará en la fila. Seguir buscando a partir de cualquiera de estas soluciones imposibles, nunca nos llevará a una solución válida. Por lo tanto, podemos podar las soluciones parciales en ese punto y concentrarnos solo en las que podrían ser viables: `row[1] = {0, 4, 5, 6, 7}`, lo que reduciría el tiempo de ejecución. Como norma general, cuanto antes se podes el espacio de búsqueda, mejor.

En otros problemas, podríamos calcular el ‘valor potencial’ de una solución parcial (y válida). Si el valor potencial es inferior al valor de la mejor solución actual, podemos abandonar la búsqueda ahí.

---

<sup>2</sup>Se dice que un programa pasa la mayor parte del tiempo ejecutando un 10 % de su código, el ‘código crítico’.

### Consejo 3: utilizar simetrías

Algunos problemas tienen simetrías y deberíamos tratar de aprovecharlas para reducir el tiempo de ejecución. En el problema de las ocho reinas, existen 92 soluciones posibles, pero solo 12 son únicas (o fundamentales), ya que hay simetrías rotacionales y lineales. Utilizando este hecho, es posible generar las 12 soluciones únicas y, si es necesario, el total de 92, rotándolas y reflejándolas. Ejemplo: `row = {7-1, 7-3, 7-5, 7-7, 7-2, 7-0, 7-6, 7-4} = {6, 4, 2, 0, 5, 7, 1, 3}`, es el reflejo horizontal de la disposición mostrada en la figura 3.1.

Sin embargo, tenemos que advertir que, en ocasiones, considerar las simetrías puede complicar el código. En la programación competitiva quizás no sea la mejor idea (nos interesa un código lo más corto posible, para minimizar la posibilidad de cometer errores). Si el beneficio que obtenemos aplicando la simetría no es significativo para resolver el problema, es mejor ignorarlo.

### Consejo 4: cálculo previo

A veces resulta útil generar tablas y otras estructuras de datos, que aceleren la búsqueda de un resultado, antes de la propia ejecución del programa. Esto se denomina cálculo previo, en el que se intercambia memoria/espacio por tiempo. Sin embargo, esta técnica no suele ser muy útil en los problemas de concursos de programación más recientes.

Por ejemplo, como sabemos que solo hay 92 soluciones para la versión estándar del problema de las ocho reinas, podemos crear un *array* bidimensional `int solution[92][8]` y llenarlo con las 92 posiciones válidas de las ocho reinas. Es decir, creamos un programa generador (que tardará un tiempo ejecutarse) que complete el *array* bidimensional `solution`. Después, podemos escribir otro programa que, simple y rápidamente, muestre las permutaciones correctas que satisfagan los requisitos del problema, a partir de las 92 configuraciones precalculadas.

### Consejo 5: intentar resolver el problema hacia atrás

Algunos problemas parecen mucho más sencillos cuando se resuelven ‘hacia atrás’ [53] (desde un ángulo *menos evidente*), en vez de utilizar un ataque frontal (desde el ángulo más obvio). Hay que estar preparados para las soluciones no convencionales.

Este consejo se entiende mejor con un ejemplo: el problema UVa 10360 - Rat Attack. Imagina un *array* bidimensional (de hasta  $1024 \times 1024$ ) que contiene ratas. Hay  $n \leq 20000$  ratas repartidas por las casillas. Determinar en qué casilla  $(x, y)$  habría que lanzar una bomba de gas para que se maximice el número de ratas muertas, en un cuadrado  $(x - d, y - d)$  a  $(x + d, y + d)$ . El valor de  $d$  es la potencia de la bomba de gas ( $d \leq 50$ ), ver la figura 3.2.

Una solución inmediata es abordar este problema desde su perspectiva más evidente: lanzar la bomba en cada una de las  $1024^2$  casillas posibles, y seleccionar la ubicación más efectiva. Por cada casilla bombardeada  $(x, y)$ , podemos realizar un barrido de complejidad  $O(d^2)$ , para contar el número de ratas muertas dentro del área afectada. En el peor de los casos, cuando el *array* tiene tamaño  $1024^2$  y  $d = 50$ , esto supondrá  $1024^2 \times 50^2 = 2621M$  operaciones. TLE<sup>3</sup>.

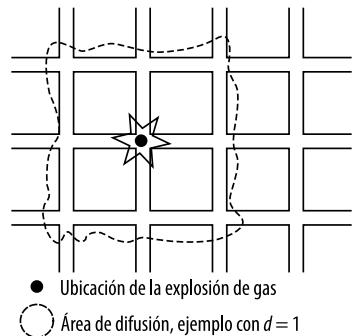


Figura 3.2: UVa 10360 [49]

<sup>3</sup>Aunque una CPU del año 2013 pueda calcular  $\approx 100M$  de operaciones en unos pocos segundos,  $2621M$

Otra opción es contemplar el problema al revés: crear un `array int killed[1024][1024]`. Para cada población de ratas en la coordenada  $(x, y)$ , añadirla a `killed[i][j]`, donde  $|i - x| \leq d$  y  $|j - y| \leq d$ . Esto es debido a que si se coloca una bomba en  $(i, j)$ , las ratas en la coordenada  $(x, y)$  morirán. Este proceso previo requiere  $O(n \times d^2)$  operaciones. Después, para determinar la posición óptima para el bombardeo, basta con encontrar las coordenadas de la entrada mayor en el array `killed`, lo que se puede hacer en  $1024^2$  operaciones. Esta solución solo requiere  $20000 \times 50^2 + 1024^2 = 51M$  de operaciones, en el peor caso posible ( $n = 20000, d = 50$ ),  $\approx 51$  veces más rápido que el ataque frontal. Será aceptada por el juez.

#### Consejo 6: optimizar el código fuente

Existen muchos trucos que se pueden utilizar para optimizar el código. Entender cómo funciona y está organizado un ordenador, especialmente el comportamiento de E/S, la memoria y la caché, puede ayudar a diseñar código mejor. A continuación, mostramos algunos ejemplos (aunque no se trata de una lista exhaustiva):

1. Una opinión interesada: utilizar C++ en vez de Java. Un algoritmo implementado utilizando C++ se ejecutará, normalmente, más rápido que en Java, en la mayoría de los jueces en línea, incluyendo UVa [49]. Algunos concursos de programación (pero no todos), le dan a los usuarios de Java tiempo adicional para compensar la diferencia de rendimiento.
2. Los usuarios de C/C++ deben utilizar las funciones `scanf/printf` de estilo C, que son más rápidas, en vez de `cin/cout`. Los usuarios de Java deben utilizar las funciones `BufferedReader/BufferedWriter`, también más veloces, de la siguiente manera:

```

1 BufferedReader br = new BufferedReader(// acelera
2 new InputStreamReader(System.in));
3 // Nota: después se necesita trocear cadenas y/o procesado de entrada
4
5 PrintWriter pw = new PrintWriter(new BufferedWriter(// acelera
6 new OutputStreamWriter(System.out)));
7 // PrintWriter nos permite usar la función pw.printf()
8 // no olvides llamar a pw.close() antes de salir de programa de Java

```

3. Utilizar la ordenación *quick* de la STL de C++ `algorithm::sort` (incluido en ‘introsort’), con una *expectativa* de complejidad de  $O(n \log n)$ , pero adecuada para la caché, en vez de la ordenación por montículos, de verdadera complejidad  $O(n \log n)$ , pero poco adecuada para la caché (sus operaciones raíz-hoja/hoja-raíz requieren muchos índices, lo que le dará pocas ventajas).
4. Acceder a los *arrays* bidimensionales fila por fila, en vez de por columnas, ya que los *arrays* multidimensionales se almacenan por filas en la memoria.
5. Realizar la manipulación de bits utilizando los tipos de datos de enteros integrados (hasta el entero de 64 bits), es más eficiente que la manipulación de índices de un *array* de booleanos (ver las máscaras de bits en las secciones 2.2, 8.2.1 y 8.3.1). Si necesitamos más de 64 bits, utilizar `bitset` de la STL de C++, en vez de `vector<bool>` (por ejemplo, en la criba de Eratóstenes, en la sección 5.5.1).

---

siguen siendo muchas en el ámbito de un concurso.

6. Utilizar estructuras de datos/tipos de bajo nivel, siempre que no se requiera la funcionalidad adicional de las de alto nivel (más amplia). Por ejemplo, utilizar un `array` con un tamaño algo superior al del tamaño de entrada máximo, en vez de un `vector` redimensionable. Además, utilizar `int` de 32 bits en vez de `long` de 64 bits, ya que los enteros de 32 bits son más rápidos en la mayoría de los jueces de 32 bits.
7. En Java, utilizar `ArrayList` (y `StringBuilder`), en vez de `Vector` (y `StringBuffer`), ya que son más rápidos. Los `Vector` y `StringBuffer` de Java son *seguros con hilos*, pero esta característica no es necesaria en los concursos de programación. Nota: en este libro utilizaremos `Vector` para evitar confusiones en la lectura de códigos de C++ y Java, para aquellos lectores que entiendan ambos lenguajes, y utilicen tanto `vector` de la STL de C++ como `Vector` de Java.
8. Declarar la mayoría de las estructuras de datos (especialmente las más pesadas, como los `arrays` grandes) una sola vez en el ámbito global. Reservar suficiente memoria como para hacer frente a la mayor entrada posible del problema. De esta forma evitaremos pasar estructuras de datos como argumentos de funciones. En los problemas con varios casos de prueba, basta con limpiar el contenido de la estructura de datos antes de abordar cada caso de prueba.
9. Cuando se pueda optar entre escribir el código de forma iterativa o recursiva, elegir la primera. Por ejemplo: la opción iterativa `next_permutation` de la STL de C++, y las técnicas de generación iterativa de subconjuntos utilizando máscaras de bits que se muestran en la sección 3.2.1, son (mucho) más rápidas que sus equivalentes recursivas (principalmente por el exceso de llamadas a funciones).
10. El acceso a `arrays` en bucles (anidados) puede ser lento. Si tienes un `array A` y accedes frecuentemente al valor de `A[i]` (sin cambiarlo) en bucles (anidados), puede ser ventajoso utilizar una variable local `temp = A[i]` y utilizarla en su lugar.
11. En C/C++, el uso *apropiado* de macros puede reducir el tiempo de implementación.
12. En C/C++, el uso *apropiado* de funciones *inline* puede reducir el tiempo de ejecución.
13. Para usuarios de C++: utilizar `arrays` de caracteres de estilo C llevará a una ejecución más rápida que `string` de la STL de C++. Para usuarios de Java: hay que tener cuidado con la manipulación de `String`, ya que ese tipo de objetos son inmutables. Por ello, las operaciones con `String` en Java pueden ser muy lentas. Es mejor utilizar `StringBuilder` en su lugar.

Busca en internet, o en libros de texto (por ejemplo, [69]), más información sobre cómo acelerar el código. Puedes practicar estas ‘habilidades de mejora del código’ seleccionando un problema difícil del juez UVa, donde el tiempo de ejecución de la mejor solución no sea 0,000s. Envía varias versiones de tu solución aceptada, y comprueba las diferencias en el tiempo de ejecución. Adopta aquellas prácticas que, de forma constante, supongan un código más rápido.

#### **Consejo 7: utilizar mejores estructuras de datos y algoritmos**

No es broma. Utilizar estructuras de datos y algoritmos mejores siempre superará cualquier optimización mencionada en los consejos 1 a 6. Si estás seguro de que has escrito el código de búsqueda completa más rápido posible, pero todavía obtienes un veredicto TLE, abandona la solución de búsqueda completa.

## Comentarios sobre búsqueda completa en concursos de programación

La principal fuente de información sobre ‘búsqueda completa’ de este capítulo, es la plataforma de entrenamiento de USACO [50]. Hemos adoptado el nombre ‘búsqueda completa’, frente a ‘fuerza bruta’ (con sus connotaciones negativas), ya que creemos que algunas soluciones de búsqueda completa pueden ser inteligentes y rápidas. Opinamos que el término ‘fuerza bruta inteligente’ puede resultar un tanto contradictorio.

Si es posible resolver un problema utilizando búsqueda completa, también estará claro cuándo utilizar los métodos de *backtracking* iterativos o recursivos. Los abordajes iterativos se utilizan cuando es posible derivar diferentes estados *con facilidad*, con alguna fórmula relativa a un cierto *contador*, y es necesario comprobar (casi) todos los estados, por ejemplo, explorando todos los índices de un *array*, enumerando (casi) todos los subconjuntos posibles de un pequeño conjunto, generando (casi) todas las permutaciones, etc. El *backtracking* recursivo se utiliza cuando es difícil derivar los diferentes estados con un índice sencillo y/o queremos podar (abundantemente) el espacio de búsqueda, por ejemplo, en el problema de las ocho reinas. Si el espacio de búsqueda de un problema que se puede resolver por búsqueda completa es grande, las soluciones de *backtracking* recursivo que permitan la poda temprana de secciones inútiles del espacio de búsqueda, son una opción mejor. La poda en búsquedas completas iterativas no es imposible, pero normalmente es complicada.

Una buena forma de mejorar tus habilidades con búsqueda completa, será resolver más problemas de este tipo, para afinar tu intuición sobre el hecho de si un problema es resoluble utilizando la. Hemos incluido, a continuación, una lista de esos problemas, separados en varias categorías. Intenta resolver todos los que puedas, especialmente los considerados **obligatorios** \*. En la sección 3.5, más adelante, encontrarás más ejemplos de *backtracking* recursivo, pero con el añadido de la técnica de *memoización*.

También trataremos técnicas de búsqueda más avanzadas en la sección 8.2, por ejemplo utilizando manipulación de bits en el *backtracking* recursivo, búsquedas de estado-espacio más complicadas o encuentro en el medio, y nos familiarizaremos con un tipo de problemas NP-complejo/completo sin ninguna propiedad especial y para los que, seguramente, no exista solución más rápida que la búsqueda completa.

En la sección 8.2.5, trataremos una clase poco utilizada de algoritmos heurísticos de búsqueda: búsqueda A\*, búsqueda de profundidad limitada (DLS), búsqueda de profundidad iterativa (IDS) y profundidad iterativa A\* (IDA\*).

## Ejercicios de programación

### Ejercicios de programación que se resuelven mediante búsqueda completa:

#### Iterativos (un bucle, exploración lineal)

- |                                                   |                                                          |
|---------------------------------------------------|----------------------------------------------------------|
| 1. UVa 00102 - Ecological Bin Packing             | (probar las 6 combinaciones)                             |
| 2. UVa 00256 - Quirksome Squares                  | (fuerza bruta; matemáticas; se puede precalcular)        |
| 3. <b>UVa 00927 - Integer Sequence from ...</b> * | (utilizar suma de series aritméticas)                    |
| 4. <b>UVa 01237 - Expert Enough</b> *             | (LA 4142 - Jakarta08; la entrada es pequeña)             |
| 5. <b>UVa 10976 - Fractions Again ?</b> *         | (pide el total de soluciones; fuerza bruta dos veces)    |
| 6. UVa 11001 - Necklace                           | (matemáticas por fuerza bruta; función <i>maximize</i> ) |
| 7. UVa 11078 - Open Credit System                 | (un barrido lineal)                                      |

### Iterativos (dos bucles anidados)

- |                                         |                                                                   |
|-----------------------------------------|-------------------------------------------------------------------|
| 1. UVa 00105 - The Skyline Problem      | (mapa de alturas; barrer a izquierda y derecha)                   |
| 2. UVa 00347 - Run, Run, Runaround ...  | (simular el proceso)                                              |
| 3. UVa 00471 - Magic Numbers            | (parecido a UVa 725)                                              |
| 4. UVa 00617 - Nonstop Travel           | (probar todas las velocidades enteras entre 30 y 60 mph)          |
| 5. UVa 00725 - Division                 | (probar todas)                                                    |
| 6. <b>UVa 01260 - Sales *</b>           | (LA 4843 - Daejeon10; comprobar todas)                            |
| 7. UVa 10041 - Vito's Family            | (probar todas las ubicaciones posibles de la casa de Vito)        |
| 8. <b>UVa 10487 - Closest Sums *</b>    | (ordenar y hacer $O(n^2)$ parejas)                                |
| 9. UVa 10730 - Antiarithmetic?          | (2 bucles anidados con poda lo resuelven; comparar con UVa 11129) |
| 10. <b>UVa 11242 - Tour de France *</b> | (con ordenación)                                                  |
| 11. UVa 12488 - Start Grid              | (2 bucles anidados; simular el proceso de adelantamiento)         |
| 12. UVa 12583 - Memory Overflow         | (2 bucles anidados; cuidado con contar de más)                    |

### Iterativos (tres o más bucles anidados, fáciles)

- |                                             |                                                                                                                      |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00154 - Recycling                    | (3 bucles anidados)                                                                                                  |
| 2. UVa 00188 - Perfect Hash                 | (3 bucles anidados; hasta que aparezca la respuesta)                                                                 |
| 3. <b>UVa 00441 - Lotto *</b>               | (6 bucles anidados; fácil)                                                                                           |
| 4. UVa 00626 - Ecosystem                    | (3 bucles anidados)                                                                                                  |
| 5. UVa 00703 - Triple Ties: The ...         | (3 bucles anidados)                                                                                                  |
| 6. <b>UVa 00735 - Dart-a-Mania *</b>        | (3 bucles anidados; después contar)                                                                                  |
| 7. <b>UVa 10102 - The Path in the ... *</b> | (4 bucles anidados sirven; no necesitamos BFS; obtener el máximo de la longitud Manhattan mínima de un '1' a un '3') |
| 8. UVa 10502 - Counting Rectangles          | (6 bucles anidados; rectángulos; no muy difícil)                                                                     |
| 9. UVa 10662 - The Wedding                  | (3 bucles anidados)                                                                                                  |
| 10. UVa 10908 - Largest Square              | (4 bucles anidados; cuadrado; no muy difícil)                                                                        |
| 11. UVa 11059 - Maximum Product             | (3 bucles anidados; la entrada es pequeña)                                                                           |
| 12. UVa 11975 - Tele-loto                   | (3 bucles anidados; simular el juego como se pide)                                                                   |
| 13. UVa 12498 - Ant's Shopping Mall         | (3 bucles anidados)                                                                                                  |
| 14. UVa 12515 - Movie Police                | (3 bucles anidados)                                                                                                  |

### Iterativos (tres o más bucles anidados, difíciles)

- |                                               |                                                                                                                                |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00253 - Cube painting                  | (probar todos; similar a UVa 11959)                                                                                            |
| 2. UVa 00296 - Safebreaker                    | (probar los 10000 códigos posibles; 4 bucles anidados; solución similar al juego 'Master Mind')                                |
| 3. UVa 00386 - Perfect Cubes                  | (4 bucles anidados con poda)                                                                                                   |
| 4. UVa 10125 - Sunsets                        | (ordenar; 4 bucles anidados; además búsqueda binaria)                                                                          |
| 5. UVa 10177 - (2/3/4)-D Sqr/Rects/...        | (2/3/4 bucles anidados; calcular previamente)                                                                                  |
| 6. UVa 10360 - Rat Attack                     | (también con suma máxima de DP de $1024^2$ )                                                                                   |
| 7. UVa 10365 - Blocks                         | (utiliza 3 bucles anidados con poda)                                                                                           |
| 8. UVa 10483 - The Sum Equals ...             | (2 bucles anidados para $a, b$ , deducir $c$ de $a, b$ ; hay 354 respuestas en el rango [0,01 .. 255,99]; similar a UVa 11236) |
| 9. <b>UVa 10660 - Citizen attention ... *</b> | (7 bucles anidados; longitud Manhattan)                                                                                        |
| 10. UVa 10973 - Triangle Counting             | (3 bucles anidados con poda)                                                                                                   |

- |                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 11. UVa 11108 - Tautology<br>12. <b><u>UVa 11236 - Grocery Store</u></b> *<br>13. UVa 11342 - Three-square<br>14. <i>UVa 11548 - Blackboard Bonanza</i><br>15. <b><u>UVa 11565 - Simple Equations</u></b> *<br>16. UVa 11804 - Argentina<br>17. UVa 11959 - Dice | (probar los $2^5 = 32$ valores con poda)<br>(3 bucles anidados para $a, b, c$ ; deducir $d$ de $a, b, c$ ; comprueba si la salida tiene 949)<br>(calcular previamente los valores cuadrados de $0^2$ a $224^2$ ; usar 3 bucles anidados para generar las respuestas; utiliza map para evitar duplicados)<br>(4 bucles anidados; cadenas; poda)<br>(3 bucles anidados con poda)<br>(5 bucles anidados)<br>(probar todas las posiciones de los dados; comparar con el segundo) |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Ver también simulación matemática en la sección 5.2.

### Iterativos (técnicas curiosas)

- |                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00140 - Bandwidth<br>2. <i>UVa 00234 - Switching Channels</i><br>3. UVa 00435 - Block Voting<br>4. UVa 00639 - Don't Get Rooked<br>5. <b><u>UVa 01047 - Zones</u></b> *                                                                | (el $n$ máximo es solo 8, usar next_permutation; el algoritmo dentro de next_permutation es iterativo)<br>(LA 5173 - WorldFinals Phoenix94; next_permutation; simulación)<br>(solo hay $2^{20}$ combinaciones de coaliciones posibles)<br>(generar $2^{4 \times 4} = 2^{16}$ combinaciones y podar)<br>(LA 3278 - WorldFinals Shanghai05; tener en cuenta que $n \leq 20$ , así que se pueden probar todos los subconjuntos de torres; después aplicar el principio de inclusión-exclusión para evitar contar de más) |
| 6. UVa 01064 - Network<br>7. UVa 11205 - The Broken Pedometer<br>8. UVa 11412 - Dig the Holes<br>9. <b><u>UVa 11553 - Grid Game</u></b> *                                                                                                     | (LA 3808 - WorldFinals Tokyo07; permutación de hasta 5 mensajes; cuidado con la palabra 'consecutivo')<br>(probar las $2^{15}$ máscaras de bits)<br>(next_permutation; encontrar una posibilidad entre 6)<br>(resolver intentando las $n!$ permutaciones; se puede usar DP y máscara de bits, ver sección 8.3.1, pero es excesivo)                                                                                                                                                                                    |
| 10. UVa 11742 - Social Constraints<br>11. UVa 12249 - Overlapping Scenes<br>12. <i>UVa 12346 - Water Gate Management</i><br>13. <i>UVa 12348 - Fun Coloring</i><br>14. <i>UVa 12406 - Help Dexter</i><br>15. <b><u>UVa 12455 - Bars</u></b> * | (probar todas las permutaciones)<br>(LA 4994 - KualaLumpur10; probar todas las permutaciones; un poco de coincidencia de cadenas)<br>(LA 5723 - Phuket11; probar las $2^n$ combinaciones; elegir la mejor)<br>(LA 5725 - Phuket11; probar las $2^n$ combinaciones)<br>(probar las $2^p$ máscaras de bits posibles; cambiar los '0' a '2')<br>(suma de subconjuntos; probar todos; UVa 12911 es más difícil)                                                                                                           |

### Backtracking recursivo (fáciles)

- |                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00167 - The Sultan Successor<br>2. UVa 00380 - Call Forwarding<br>3. UVa 00539 - The Settlers ...<br>4. <b><u>UVa 00624 - CD</u></b> *<br>5. UVa 00628 - Passwords<br>6. UVa 00677 - All Walks of length "n" ...<br>7. UVa 00729 - The Hamming Distance ...<br>8. UVa 00750 - 8 Queens Chess Problem<br>9. UVa 10276 - Hanoi Tower Troubles Again | (problema de las 8 reinas)<br>(backtracking sencillo; hay que trabajar con cadenas, sección 6.2)<br>(camino sencillo más largo en un grafo general pequeño)<br>(tamaño de entrada pequeño; el backtracking es suficiente)<br>(backtracking; seguir las reglas del enunciado)<br>(mostrar todas las soluciones con backtracking)<br>(generar todas las cadenas de bits posibles)<br>(problema clásico de backtracking)<br>(insertar un número, de uno en uno) |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

10. UVa 10344 - 23 Out of 5
  11. UVa 10452 - Marcus, help
  12. **UVa 10576 - Y2K Accounting Bug** \*
  13. **UVa 11085 - Back to the 8-Queens** \*
- (reordenar los 5 operandos y los 3 operadores)  
 (desde cada lugar, Indy puede ir a adelante/izda/dcha; probar todos)  
 (generar todos; poda; tomar el máximo)  
 (ver UVa 750; cálculo previo)

### Backtracking recursivo (medianos)

1. UVa 00222 - Budget Travel
  2. *UVa 00301 - Transportation*
  3. UVa 00331 - Mapping the Swaps
  4. UVa 00487 - Boggle Blitz
  5. **UVa 00524 - Prime Ring Problem** \*
  6. UVa 00571 - Jugs
  7. **UVa 00574 - Sum It Up** \*
  8. UVa 00598 - Bundling Newspaper
  9. UVa 00775 - Hamiltonian Cucle
  10. *UVa 10001 - Garden of Eden*
  11. *UVa 10063 - Knuth's Permutation*
  12. *UVa 10460 - Find the Permuted String*
  13. UVa 10475 - Help the Leaders
  14. **UVa 10503 - The dominoes solitaire** \*
  15. UVa 10506 - Ouroboros
  16. *UVa 10950 - Bad Code*
  17. UVa 11201 - The Problem with the ...
  18. UVa 11961 - DNA
- (parece un problema de DP, pero el estado no se puede *memoizar* ya que 'tank' es de coma flotante; por suerte, la entrada no es grande)  
 (es posible  $2^{22}$  con poda)  
 $(n \leq 5...)$   
 (usar map para almacenar las palabras generadas)  
 (ver también la sección 5.5.1)  
 (la solución puede ser subóptima; usar etiqueta para evitar ciclos)  
 (mostrar todas las soluciones con *backtracking*)  
 (mostrar todas las soluciones con *backtracking*)  
 (el *backtracking* es suficiente porque el espacio de búsqueda no puede ser tan grande; en un grafo denso es más probable tener un ciclo hamiltoniano, así que podemos podar antes; NO ES NECESARIO buscar el mejor, como en el problema del viajante)  
 (el límite superior de  $2^{32}$  puede asustar pero, con una poda eficiente, entramos en el límite de tiempo porque el caso de prueba no es límite)  
 (hacer lo que se pide)  
 (de naturaleza similar a UVa 10063)  
 (generar y podar; probar todos)  
 (solo 13 espacios como máximo)  
 (cualquier solución válida es AC; generar todos los dígitos siguientes posibles (hasta [0..9] en base 10); comprobar si sigue siendo una secuencia Ouroboros válida)  
 (ordenar la entrada; ejecutar *backtracking*; la salida debe estar ordenada; mostrar solo los 100 primeros en la salida)  
 (*backtracking* con cadenas)  
 (hay un máximo de  $4^{10}$  cadenas de ADN posibles; además, la potencia de mutación es de  $K \leq 5$  como mucho, así el espacio de búsqueda es mucho más pequeño; ordenar la entrada y eliminar duplicados)

### Backtracking recursivo (difíciles)

1. UVa 00129 - Krypton Factor
  2. UVa 00165 - Stamps
  3. **UVa 00193 - Graph Coloring** \*
  4. UVa 00208 - Firetruck
  5. **UVa 00416 - LED Test** \*
  6. UVa 00433 - Bank (Not Quite O.C.R.)
  7. UVa 00565 - Pizza Anyone?
- (*backtracking*; comprobación por procesamiento de cadenas; un poco de formato en la salida)  
 (también necesita un poco de DP; se puede calcular previamente)  
 (conjunto independiente máximo; la entrada es pequeña)  
 (LA 5147 - WorldFinals SanAntonio91; *backtracking* con algo de poda)  
 (*backtracking*; probar todos)  
 (similar a UVa 416)  
 (*backtracking* con mucha poda)

|                                  |                                                                                                                                                                                                                                                                                                                                |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8. UVa 00861 - Little Bishops    | ( <i>backtracking</i> con poda, como en la solución de <i>backtracking</i> recursivo para las 8 reinas; entonces calcular los resultados)                                                                                                                                                                                      |
| 9. <u>UVa 01262 - Password *</u> | (LA 4845 - Daejeon10; ordenar primero las columnas de las dos rejillas de $6 \times 5$ , entonces podemos procesar contraseñas comunes en orden lexicográfico; <i>backtracking</i> ; importante: ignorar dos contraseñas similares) (este problema es similar al de las N reinas; pero hay que encontrar y utilizar el patrón) |
| 10. UVa 10094 - Place the Guards | ( <i>backtracking</i> con poda; probar las $N!$ (13!) permutaciones que cumplen con el requisito; entonces calcular los resultados)                                                                                                                                                                                            |
| 11. UVa 10128 - Queue            | (empezar simplificando la entrada compleja; después <i>backtracking</i> ) (problema de ciclos de peso medio mínimo; se resuelve con <i>backtracking</i> con mucha poda cuando la media actual es mayor que el mejor coste de ciclo de peso medio encontrado)                                                                   |
| 12. UVa 10582 - ASCII Labyrinth  |                                                                                                                                                                                                                                                                                                                                |
| 13. UVa 11090 - Going in Cycle   |                                                                                                                                                                                                                                                                                                                                |

### 3.3 Divide y vencerás

Divide y vencerás (abreviado como D&C), es un paradigma de resolución en el que un problema se hace más *sencillo* al ‘dividirlo’ en trozos más pequeños y solucionando cada uno de ellos. Estos son los pasos:

1. Dividir el problema original, más o menos por la mitad, en *subproblemas*.
2. Encontrar soluciones a cada uno de esos subproblemas, que ahora serán más sencillos.
3. Si es necesario, combinarlas para obtener la solución completa del problema principal.

En las secciones anteriores del libro hemos visto algunos ejemplos de la técnica D&C: diferentes algoritmos de ordenación (por ejemplo, *quick*, por mezcla, por montículos) y búsqueda binaria en la sección 2.2, utilizan este paradigma. La forma en que se organizan los datos en un árbol de búsqueda binaria, montículo, árbol de segmentos y árbol Fenwick en las secciones 2.3, 2.4.3 y 2.4.4, también hacen uso de D&C.

#### 3.3.1 Usos destacados de la búsqueda binaria

En esta sección, trataremos el paradigma D&C junto al conocido algoritmo de búsqueda binaria. Clasificamos la búsqueda binaria como un algoritmo de ‘divide y vencerás’, aunque una referencia [40] sugiere que, en realidad, debería estar clasificado como ‘reduce (a la mitad) y vencerás’, pues lo cierto es que no combina el resultado. Ponemos de relieve este algoritmo porque casi todos los concursantes lo conocen, aunque no muchos son conscientes de que se puede utilizar de formas no evidentes.

##### Búsqueda binaria: el uso normal

Recordemos que el uso *tradicional* de la búsqueda binaria consiste en buscar un elemento en un *array ordenado y estático*. Comprobamos el elemento central del *array* ordenado para determinar

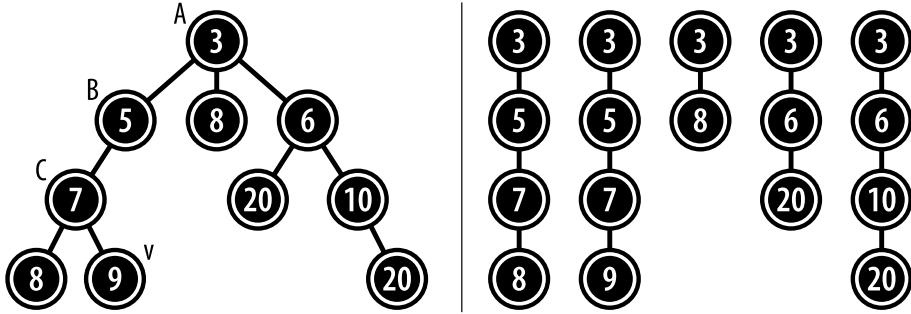


Figura 3.3: My Ancestor (los 5 caminos de la raíz a una hoja están ordenados)

si contiene lo que estamos buscando. Si es así, o si no hay más elementos a considerar, nos detenemos. En caso contrario, podemos decidir si la respuesta se encuentra a la izquierda o a la derecha del elemento central, y seguimos buscando. Como reducimos el espacio de búsqueda a la mitad (en términos binarios) después de cada comprobación, la complejidad de este algoritmo es  $O(\log n)$ . En la sección 2.2, hemos visto que existen rutinas integradas en las bibliotecas para ejecutar este algoritmo, como por ejemplo `algorithm::lower_bound` de la STL de C++ (y `Collections.binarySearch` en Java).

Ésta *no* es la única manera de utilizar la búsqueda binaria. El requisito previo para realizar una búsqueda binaria, *una secuencia ordenada estática (en forma de array o vector)*, aparece también en otras estructuras de datos menos comunes, como en un camino de la raíz a una hoja de un árbol (que no será binario o completo necesariamente) que satisfaga la propiedad de *montículo mínimo*. A continuación, tratamos esta variante.

### Búsqueda binaria en estructuras de datos no comunes

Este problema original se titula ‘My Ancestor’ y se utilizó en el concurso ICPC nacional de Tailandia en 2009. Enunciado resumido del problema: dado un árbol (genealógico) ponderado de hasta  $N \leq 80K$  vértices, con la característica especial de que *los valores de los vértices crecen de la raíz a las hojas*, encontrar el vértice del *antepasado* más cercano a la raíz, comenzando desde un vértice inicial  $v$ , que tenga, al menos, un peso  $P$ . Tendremos hasta  $Q \leq 20K$  de esas consultas *fueras de línea*. Examinemos la parte izquierda de la figura 3.3. Si  $P = 4$ , entonces la respuesta es el vértice etiquetado como ‘B’, con un valor de 5, ya que es el antepasado de  $v$  más cercano a la raíz ‘A’, y tiene un valor de  $\geq 4$ . Si  $P = 7$ , entonces la respuesta será ‘C’, con valor 7. Si  $P \geq 9$ , no hay respuesta válida.

La solución ingenua consiste en realizar un barrido lineal de complejidad  $O(N)$  por cada consulta: comenzando con el vértice dado  $v$ , vamos subiendo por el árbol (genealógico) hasta que encontremos el primer vértice cuyo progenitor directo tenga un valor  $< P$ , o hasta que lleguemos a la raíz. Si ese vértice tiene un valor  $\geq P$ , y no es el mismo  $v$ , habremos encontrado la solución. Como hay  $Q$  consultas, este abordaje se ejecuta en  $O(QN)$  (el árbol de entrada puede ser una lista enlazada ordenada, o cuerda, de longitud  $N$ ) y obtendremos un veredicto TLE ya que  $N \leq 80K$  y  $Q \leq 20K$ .

Una solución mejor consiste en almacenar las  $20K$  consultas (no tenemos que contestarlas inmediatamente). Recorrer el árbol *una sola vez*, comenzando desde la raíz, utilizando el algoritmo

de ordenación previa de recorrido del árbol, con complejidad  $O(N)$  (sección 4.7.2). Esta ordenación previa de recorrido del árbol, estará ligeramente modificada para recordar la secuencia ‘raíz a vértice actual’ parcial, a medida que se ejecuta. El *array* estará siempre ordenado, ya que los vértices junto a la ruta ‘raíz a vértice actual’ tienen pesos que se incrementan, como vemos en la figura 3.3 (a la derecha). La ordenación previa de recorrido del árbol que aparece a la izquierda de la figura 3.3, produce el siguiente *array* ordenado ‘raíz a vértice actual’ parcial:  $\{\{3\}, \{3, 5\}, \{3, 5, 7\}, \{3, 5, 7, 8\}$ , vuelta,  $\{3, 5, 7, 9\}$ , vuelta, vuelta, vuelta,  $\{3, 8\}$ , vuelta,  $\{3, 6\}$ ,  $\{3, 6, 20\}$ , vuelta,  $\{3, 6, 10\}$ , y, por último,  $\{3, 6, 10, 20\}$ , vuelta, vuelta, vuelta (hecho).

Durante el recorrido de ordenación previa de complejidad  $O(N)$ , y cuando lleguemos a un vértice consultado, podemos realizar una **búsqueda binaria** de coste  $O(\log N)$  (para ser precisos: `lower_bound`) en el *array* de pesos parcial ‘raíz a vértice actual’, para obtener el antepasado más cercano a la raíz con un valor de, al menos,  $P$ , guardando estas soluciones. Por último, podemos realizar una iteración  $O(Q)$  sencilla, para mostrar los resultados. La complejidad de tiempo global de este método es  $O(N + Q \log N)$ , que ahora estará en los límites de la entrada.

## Método de bisección

Hemos tratado las aplicaciones de la búsqueda binaria para encontrar elementos en secuencias ordenadas estáticas. Sin embargo, el **principio** de la búsqueda binaria<sup>4</sup> también se puede utilizar para encontrar la raíz de una función que podría ser difícil de calcular directamente.

Ejemplo: puedes comprar un coche con un préstamo que quieras pagar en cuotas mensuales de  $d$  dólares en  $m$  meses. Supongamos que el valor inicial del coche es  $v$  dólares, y que el interés establecido por el banco es del  $i\%$  por el remanente a pagar al final de cada mes. ¿Qué cantidad de dinero  $d$  debes pagar al mes (con dos dígitos de precisión decimal)?

Supongamos que  $d = 576,19$ ,  $m = 2$ ,  $v = 1000$  e  $i = 10\%$ . Transcurrido un mes, la deuda será de  $1000 \times (1,1) - 576,19 = 523,81$ . Después de dos meses, será de  $523,81 \times (1,1) - 576,19 \approx 0$ . Si solo tenemos los datos  $m = 2$ ,  $v = 1000$  e  $i = 10\%$ , ¿cómo podríamos determinar que  $d = 576,19$ ? En otras palabras, hay que encontrar la raíz  $d$  de forma que la función de pago de la deuda  $f(d, m, v, i) \approx 0$ .

Una forma *fácil* de resolver este problema de determinación de una raíz, es utilizar el método de bisección. Elegimos, como punto de salida, un rango razonable. Queremos fijar  $d$  dentro del rango  $[a..b]$ , donde  $a = 0,01$ , ya que tendremos que pagar al menos un céntimo y  $b = (1 + i\%) \times v$ , ya que el momento más temprano en el que podremos completar el pago es  $m = 1$ , si pagamos exactamente  $(1 + i\%) \times v$  dólares después del primer mes. En este ejemplo,  $b = (1 + 0,1) \times 1000 = 1100,00$  dólares. Para que funcione el método de bisección<sup>5</sup>, debemos asegurarnos de que los valores de la función en los dos extremos del rango real inicial  $[a..b]$ , es decir,  $f(a)$  y  $f(b)$ , tienen signos opuestos (como resulta cierto en los  $a$  y  $b$  calculados anteriormente).

Cabe poner de relieve que el método de bisección solo requiere  $O(\log_2((b - a)/\epsilon))$  iteraciones para obtener una respuesta lo suficientemente buena (el error es menor que el umbral de error  $\epsilon$  que nos podemos permitir). En este ejemplo, el método de bisección solo realiza  $\log_2 1099,99/\epsilon$  intentos. Utiliza un  $\epsilon = 1e-9$  pequeño, esto supone solo  $\approx 40$  iteraciones.

<sup>4</sup>Utilizamos el término ‘principio de la búsqueda binaria’ para referirnos al método de D&C de reducir a la mitad el rango de respuestas posibles. El ‘algoritmo de búsqueda binaria’ (encontrar el índice de un elemento en una cadena ordenada), el ‘método de bisección’ (encontrar la raíz de una función) y ‘buscar la respuesta de forma binaria’ (tratado en la siguiente subsección), son variantes de este principio.

<sup>5</sup>Los requisitos para el método de bisección (que utiliza el principio de la búsqueda binaria) son ligeramente diferentes a los del algoritmo de búsqueda binaria, que requiere un *array* ordenado.

| <b>a</b>   | <b>b</b>   | <b>d = <math>\frac{a+b}{2}</math></b> | <b>estado: <math>f(d, m, v, i)</math></b> | <b>acción</b>      |
|------------|------------|---------------------------------------|-------------------------------------------|--------------------|
| 0.01       | 1100.00    | 550.005                               | corto por 54.9895                         | incrementar $d$    |
| 550.005    | 1100.00    | 825.0025                              | largo por 522.50525                       | decrementar $d$    |
| 550.005    | 825.0025   | 687.50375                             | largo por 233.757875                      | decrementar $d$    |
| 550.005    | 687.50375  | 618.754375                            | largo por 89.384187                       | decrementar $d$    |
| 550.005    | 618.754375 | 584.379688                            | largo por 17.197344                       | decrementar $d$    |
| 550.005    | 584.379688 | 567.192344                            | corto por 18.896078                       | incrementar $d$    |
| 567.192344 | 584.379688 | 575.786016                            | corto por 0.849366                        | incrementar $d$    |
| ...        | ...        | ...                                   | unas pocas iteraciones después ...        | ...                |
| ...        | ...        | 576.190476                            | parar; el error es inferior a $\epsilon$  | respuesta = 576.19 |

Tabla 3.1: Ejecución del método de bisección en la función de ejemplo

Incluso utilizando un  $\epsilon = 1e-15$  más pequeño, solo necesitaríamos  $\approx 60$  intentos. Es evidente que el número de intentos es *pequeño*. El método de bisección es mucho más eficiente en comparación a la evaluación exhaustiva de todos los valores posibles de  $d = [0, 01..1100, 00]/\epsilon$  en el ejemplo propuesto. Nota: el método de bisección se puede escribir como un bucle que prueba los valores de  $d \approx 40$  a 60 veces (se puede consultar nuestra implementación en la siguiente sección).

### Buscar la respuesta de forma binaria

La versión resumida del problema UVa 11935 - Through the Desert dice: imagina que eres un explorador que intenta cruzar un desierto. Puedes utilizar un *jeep* con un depósito de combustible ‘suficientemente grande’, que inicialmente está lleno. A lo largo del viaje encontrarás diversos eventos como ‘conducir’ (que consume combustible), ‘fugas de gasolina’ (que aumenta más el consumo de combustible), ‘llegar a una estación de servicio’ (que te permite repostar el depósito a su capacidad original), ‘encontrar a un mecánico’ (que repara las fugas) o ‘llegar al destino’ (final). Tienes que determinar la capacidad *mínima posible* del depósito del vehículo, que te permita llegar el destino. La respuesta debe tener tres dígitos de precisión decimal.

Si conocemos la capacidad del depósito de combustible del *jeep*, estamos ante un problema de simulación. Partiendo del inicio, podemos simular cada evento sucesivo y determinar si el objetivo es posible sin quedarnos sin combustible. El problema está en que no conocemos dicha capacidad, sino que ese es, precisamente, el valor que estamos buscando.

Partiendo del enunciado del problema, podemos calcular que el rango de respuestas posibles está entre  $[0.000..10000.000]$ , con una precisión decimal de tres dígitos. Sin embargo, eso son  $10M$  de posibilidades. Si las probamos todas, obtendremos un veredicto de TLE.

Por suerte, este problema tiene una propiedad que podemos aprovechar. Supongamos que la respuesta correcta es  $X$ . Establecer la capacidad del depósito en cualquier valor entre  $[0.000..X-0.001]$  *no* hará que el *jeep* llegue a su destino. Por otro lado, fijar la capacidad en cualquier valor entre  $[X..10000.00]$  hará que el *jeep* llegue al destino, normalmente con algo de combustible sobrante. Esta particularidad nos permite buscar la respuesta  $X$  de forma binaria. Podemos utilizar el siguiente código para obtener la solución correcta:

```

1 #define EPS 1e-9 // este valor es ajustable; 1e-9 suele ser suficiente
2 bool can(double f) { // se omiten detalles de esta simulación
3 // devolver verdadero si el jeep puede llegar a destino con combustible f
4 // devolver falso en otro caso
5 }
6
7 // dentro de int main()
8 // búsqueda binaria de la respuesta, después simular
9 double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
10 while (fabs(hi-lo) > EPS) { // si no hemos encontrado la respuesta
11 mid = (lo + hi) / 2.0; // probar el valor central
12 if (can(mid)) { ans = mid; hi = mid; } // guardar el valor, continuar
13 else lo = mid;
14 }
15
16 % printf("%.3lf\n", ans); // al terminar el bucle, tenemos la respuesta

```

Algunos programadores prefieren utilizar un número constante de operaciones de refinado, en vez de permitir que el número de éstas varíe dinámicamente, para evitar errores de precisión al probar si `fabs(hi - lo) > EPS` y, con ello, verse en un bucle infinito. Los únicos cambios necesarios para implementarlo de esta segunda manera se muestran a continuación. El resto de código será el mismo.

```

1 double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
2 for (int i = 0; i < 50; i++) { // log_2 ((10000.0 - 0.0) / 1e-9) ~= 43
3 mid = (lo+hi) / 2.0; // iterar 50 veces debe dar suficiente precisión
4 if (can(mid)) { ans = mid; hi = mid; }
5 else lo = mid;
6 }

```

### Ejercicio 3.3.1.1

Hay una solución alternativa al problema UVa 11935, que no utiliza la técnica de ‘buscar la respuesta de forma binaria’. ¿Sabrías decir cuál es?

### Ejercicio 3.3.1.2\*

El ejemplo anterior implica buscar la respuesta de forma binaria, cuando la respuesta es un número de coma flotante. Modifica el código para resolver problemas de ‘búsqueda de la respuesta de forma binaria’, donde la respuesta esté en un *rango de enteros*.

## Comentarios sobre divide y vencerás en concursos de programación

El paradigma divide y vencerás se utiliza normalmente en algoritmos comunes que dependen de él: búsqueda binaria y sus variantes, ordenación *quick*, por mezcla o montículos y estructuras de datos: árbol de búsqueda binaria, montículo, árbol de segmentos, árbol de Fenwick, etc. Pero según nuestra experiencia, resulta que el uso más común de divide y vencerás en los concursos de programación se encuentra en el principio de búsqueda binaria. Si quieras tener éxito en los concursos de programación, deberías dedicar tiempo a practicar varias formas de aplicarlo.

Una vez que la técnica de búsqueda de la respuesta de forma binaria te resulte más familiar, te recomendamos visitar la sección 8.4.1, donde hay ejercicios de programación adicionales, que utilizan esta técnica junto con *otro algoritmo* que discutiremos posteriormente en el libro.

Hemos notado que no hay muchos problemas de D&C más allá de nuestra clasificación de búsqueda binaria. La mayoría de las soluciones D&C son relativas a la geometría o específicas de cada problema, por lo que no pueden tratarse en detalle. Sin embargo, encontraremos algunos en las secciones 8.4.1 (buscar la respuesta de forma binaria y fórmulas geométricas), 9.14 (índice de inversión), 9.21 (potencia de matrices) y 9.29 (problema de selección).

## Ejercicios de programación

### Ejercicios de programación que se resuelven utilizando divide y vencerás:

#### Búsqueda binaria

1. UVa 00679 - Dropping Balls (búsqueda binaria; existe solución con manipulación de bits)
2. UVa 00957 - Popes (búsqueda completa y búsqueda binaria: `upper_bound`)
3. UVa 10077 - The Stern-Brocot ... (búsqueda binaria)
4. UVa 10474 - Where is the Marble? (sencillo: usar `sort` y después `lower_bound`)
5. **UVa 10567 - Helping Fill Bates \*** (guardar índices `i`nc de cada `char` de `S` en 52 vectores; búsqueda binaria de la posición del `char` en el vector correcto)  
(búsqueda binaria)
6. UVa 10611 - Playboy Chimp (búsqueda binaria y algo de matemáticas)
7. UVa 10706 - Number Sequence (usar criba; búsqueda binaria)
8. UVa 10742 - New Rule in Euphonmia (ordenar; para un precio `p[i]`, comprobar si existe el precio (`M-p[i]`) con búsqueda binaria)  
(generar números con factor 2 y/o 3; `sort`; `upper_bound`)
9. **UVa 11057 - Exact Sum \*** (un tipo de búsqueda ternaria)  
(`[lower|upper]_bound` en la secuencia ordenada `N`)
10. UVa 11621 - Small Factors (el array de entrada tiene propiedades ordenadas especiales; usar `lower_bound` para acelerar la búsqueda)
11. *UVa 11701 - Cantor* (autor: Felix Halim)
12. UVa 11876 - N + NOD (N) (autor: Felix Halim)
13. **UVa 12192 - Grapevine \*** (método de bisección; soluciones alternativas en [http://www.algorithmist.com/index.php/UVa\\_10341](http://www.algorithmist.com/index.php/UVa_10341))
14. ICPC Nacional de Tailandia 2009 - My Ancestor (búsqueda binaria de la respuesta y simulación)

#### Método de bisección o búsqueda binaria de la respuesta

1. **UVa 10341 - Solve It \*** (método de bisección; soluciones alternativas en [http://www.algorithmist.com/index.php/UVa\\_10341](http://www.algorithmist.com/index.php/UVa_10341))
2. **UVa 11413 - Fill the ... \*** (búsqueda binaria de la respuesta y simulación)

3. UVa 11881 - Internal Rate of Return (método de bisección)
4. UVa 11935 - Through the Desert (búsqueda binaria de la respuesta y simulación)
5. **UVa 12032 - The Monkey ... \*** (búsqueda binaria de la respuesta y simulación)
6. **UVa 12190 - Electric Bill** (búsqueda binaria de la respuesta y álgebra)
7. IOI 2010 - Quality of Living (búsqueda binaria de la respuesta)

Ver también divide y vencerás para problemas de geometría (sección 8.4.1)

#### Otros problemas de divide y vencerás

1. **UVa 00183 - Bit Maps \*** (ejercicio sencillo de divide y vencerás)
2. IOI 2011 - Race (D&C; ver si el camino de la solución utiliza un vértice o no)

Ver también estructuras de datos con divide y vencerás (sección 2.3)

## 3.4 Voraz

Se dice de un algoritmo que es voraz si, de forma local, realiza la elección óptima en cada paso, con la esperanza de alcanzar así la solución óptima global. En algunos casos la voracidad funciona, la solución es corta y se ejecuta de forma eficiente. En *muchos* otros, sin embargo, no es así. Como se menciona en otros libros de texto habituales, por ejemplo [7, 38], un problema debe mostrar las dos siguientes propiedades para que un algoritmo voraz funcione:

1. Tiene subestructuras óptimas. La solución óptima al problema contiene soluciones óptimas a los subproblemas.
2. Tiene la propiedad voraz (la cuál es difícil o poco eficiente de demostrar en el entorno de un concurso). Si elegimos la que nos pueda parecer la mejor solución y procedemos a resolver el subproblema restante, llegaremos a la solución óptima. No será necesario reconsiderar nuestras elecciones anteriores.

### 3.4.1 Ejemplos

#### Cambio de monedas - la versión voraz

Enunciado del problema: dadas una cantidad objetivo de céntimos  $V$  y una lista de denominaciones de  $n$  monedas, es decir, tenemos  $\text{coinValue}[i]$  (en céntimos) para los tipos de monedas  $i \in [0..n-1]$ , ¿cuál es el número mínimo de monedas que debemos utilizar para representar la cantidad  $V$ ? Asumimos que tenemos una cantidad ilimitada de monedas de cualquier tipo. Por ejemplo: si  $n = 4$ ,  $\text{coinValue} = \{25, 10, 5, 1\}$  céntimos<sup>6</sup>, y queremos representar  $V = 42$  céntimos, podemos utilizar este algoritmo voraz: seleccionamos la moneda de mayor denominación que no supere la cantidad restante, es decir,  $42-\underline{25} = 17 \rightarrow 17-\underline{10} = 7 \rightarrow 7-\underline{5} = 2 \rightarrow 2-\underline{1} = 1 \rightarrow 1-\underline{1} = 0$ , 5 monedas en total. Esta aproximación es óptima.

<sup>6</sup>La presencia de la moneda de 1 céntimo nos asegura que podremos obtener cualquier valor.

El problema anterior tiene los dos elementos para que el algoritmo voraz tenga éxito:

1. Tiene subestructuras óptimas.

Hemos visto que, en nuestra búsqueda de la representación de 42 céntimos, hemos utilizado  $25+10+5+1+1$ . Ésta resulta ser una solución óptima de 5 monedas al problema original. Las soluciones óptimas a los subproblemas están contenidas dentro de la solución de 5 monedas, de forma que:

- a. Para representar 17 céntimos, podemos utilizar  $10+5+1+1$  (parte de la solución para 42 céntimos).
  - b. Para representar 7 céntimos, podemos utilizar  $5+1+1$  (también parte de la solución para 42 céntimos), etc.
2. Tiene la propiedad voraz: dada cada cantidad  $V$ , podemos restarle de forma voraz la denominación de moneda mayor que no supere en valor a la cantidad  $V$ . Se puede demostrar (no lo hacemos por brevedad) que utilizar cualquier otra estrategia no nos llevará a una solución óptima, al menos para este conjunto de denominaciones.

Sin embargo, este algoritmo voraz *no* funciona con *todos* los conjuntos de denominaciones de monedas. Tomemos el ejemplo de  $\{4, 3, 1\}$  céntimos. Para sumar 6 céntimos con este conjunto, el algoritmo voraz utilizará 3 monedas  $\{4, 1, 1\}$ , en vez de la solución óptima de 2 monedas  $\{3, 3\}$ . Volveremos sobre la versión general de este problema más tarde, en la sección 3.5.2 (programación dinámica).

#### UVa 410 - Station Balance (balanceo de carga)

Dadas  $1 \leq C \leq 5$  estancias, que pueden almacenar 0, 1 o 2 especímenes, si hay  $1 \leq S \leq 2C$  especímenes y una lista  $M$  de las masas de los  $S$  especímenes, determinar en qué estancia deberíamos almacenar cada especímen para minimizar el ‘desequilibrio’. En la figura 3.4 encontramos una explicación visual<sup>7</sup>.

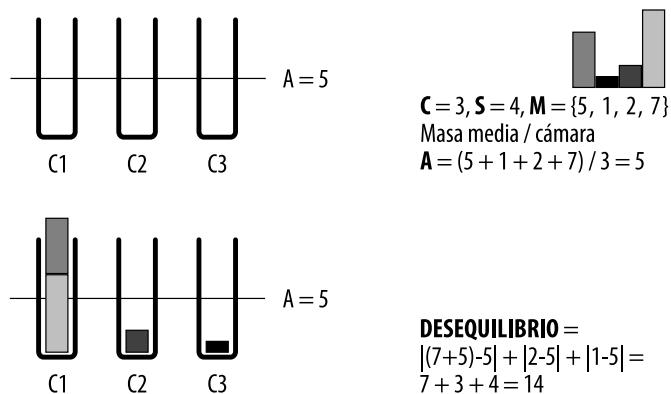


Figura 3.4: Visualización de UVa 410 - Station Balance

<sup>7</sup>Ya que  $C \leq 5$  y  $S \leq 10$ , este problema se podría resolver mediante búsqueda completa. Sin embargo, es más sencillo abordarlo utilizando un algoritmo voraz.

Determinamos que  $A = (\sum_{j=1}^S M_j)/C$ , es decir,  $A$  es la media de la masa total en cada una de las  $C$  estancias.

Determinamos que el desequilibrio  $= \sum_{i=1}^C |X_i - A|$ , es decir, la suma de las diferencias entre la masa total en cada cámara en relación a  $A$ , donde  $X_i$  es la masa total de los especímenes en la estancia  $i$ .

Este problema se puede resolver utilizando un algoritmo voraz pero, para llegar a esa solución, son necesarias algunas observaciones.

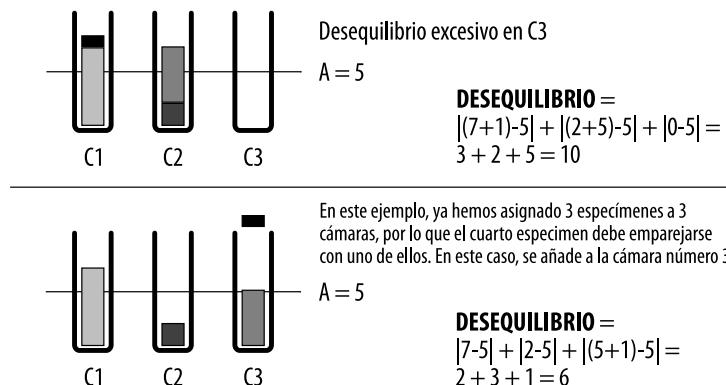


Figura 3.5: UVa 410 - Observaciones

Observación 1: si existe una estancia vacía, normalmente es beneficioso, y nunca peor, trasladar a un especimen desde una estancia con dos especímenes a la vacía. En caso contrario, la estancia vacía contribuye al desequilibrio, como se puede ver en la parte superior de la figura 3.5.

Observación 2: Si  $S > C$ , entonces  $S - C$  especímenes deben estar emparejados con una estancia que ya contenga otros especímenes, esto es el principio del palomar. Ver la parte inferior de la figura 3.5.

La clave es que la solución a este problema se puede simplificar mediante la ordenación: si  $S < 2C$ , sumamos  $2C - S$  especímenes vacíos de masa 0. Por ejemplo,  $C = 3$ ,  $S = 4$ ,  $M = \{5, 1, 2, 7\} \rightarrow C = 3, S = 6, M = \{5, 1, 2, 7, 0, 0\}$ . A continuación, ordenamos los especímenes según su masa, de forma que  $M_1 \leq M_2 \leq \dots \leq M_{2C-1} \leq M_{2C}$ . En el presente ejemplo,  $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$ . Al añadir especímenes vacíos y ordenarlos, la estrategia voraz resulta más evidente:

- Emparejamos los especímenes con masas  $M_1$  y  $M_{2C}$ , y van a la estancia 1, entonces
- Emparejamos los de masas  $M_2$  y  $M_{2C-1}$ , y los ponemos en la estancia 2, etc.

Este algoritmo voraz, conocido como *balanceo de carga*, funciona. Ver la figura 3.6.

Resulta complicado describir las técnicas utilizadas para llegar a esta solución voraz. Encontrar soluciones voraces es un arte, al igual que las soluciones de búsqueda completa requieren de un grado de creatividad. Un consejo: si no encontramos una estrategia voraz evidente, debemos *ordenar* los datos o introducir alguna variante, y volver a evaluar la situación.

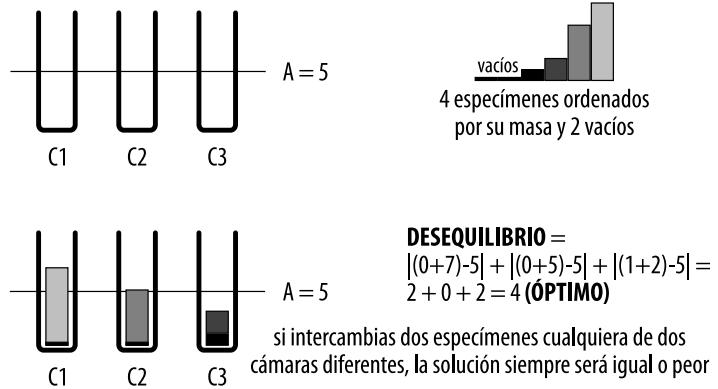


Figura 3.6: UVa 410 - Solución voraz

#### UVa 10382 - Watering Grass (cobertura de intervalos)

Enunciado del problema: en una franja horizontal de hierba de  $L$  metros de largo y  $W$  metros de ancho, hay  $n$  aspersores instalados. Cada aspersor está centrado verticalmente en la franja. Se nos proporciona la posición, en forma de distancia desde el extremo izquierdo de la línea central, y el radio de operación de cada uno de ellos. ¿Cuál es el número mínimo de aspersores que deben estar funcionando para regar la franja completa de hierba? Límite:  $n \leq 10000$ . Podemos ver una ilustración del problema a la izquierda de la figura 3.7. La respuesta, en este caso, es de 6 aspersores (etiquetados como {A, B, D, E, F, H}). Quedarán 2 aspersores sin usar: {C, G}.

No es posible resolver este problema mediante una estrategia de fuerza bruta que explore todos los subconjuntos posibles de aspersores, ya que su número podría ser de hasta 10000. Sin duda, resulta inabordable comprobar los  $2^{10000}$  subconjuntos existentes.

Este problema es, en realidad, una variante del conocido problema voraz llamado *cobertura de intervalos*. Sin embargo, incluye una simple variante geométrica. El problema original se ocupa de los intervalos, mientras que este trata de aspersores que tienen círculos de influencia en un área horizontal, en vez de intervalos sencillos. Lo primero que tendremos que hacer es transformar el problema, para que se parezca más a la cobertura de intervalos tradicional.

En la parte derecha de la figura 3.7, veremos que podemos convertir los círculos y franjas horizontales en intervalos. Podemos calcular  $dx = \sqrt{R^2 - (W/2)^2}$ . Supongamos que un círculo está centrado en  $(x, y)$ . El intervalo representado por este círculo es  $[x-dx..x+dx]$ . Para entender por qué esta idea funciona, hay que observar que el segmento de círculo adicional más allá de  $dx$ , alejado de  $x$ , no cubre completamente la franja en el área horizontal sobre la que se extiende. Si esta transformación geométrica te plantea dificultades, consulta la sección 7.2.4, en la que se tratan operaciones básicas con *triángulos rectángulos*.

Ahora que hemos transformado el problema original en el problema de cobertura de intervalos, podemos utilizar un algoritmo voraz. En primer lugar, nuestro algoritmo ordena los intervalos por su extremo izquierdo *creciente* y, en caso de que este sea igual, por su extremo derecho *decreciente*. Después, el algoritmo voraz procesa los intervalos de uno en uno. Toma el intervalo que cubra ‘lo más a la derecha posible’ y que también cubra de forma ininterrumpida desde la zona más a la izquierda hasta la zona más a la derecha de la franja de hierba. Ignora los intervalos que ya estén cubiertos por otros anteriores.

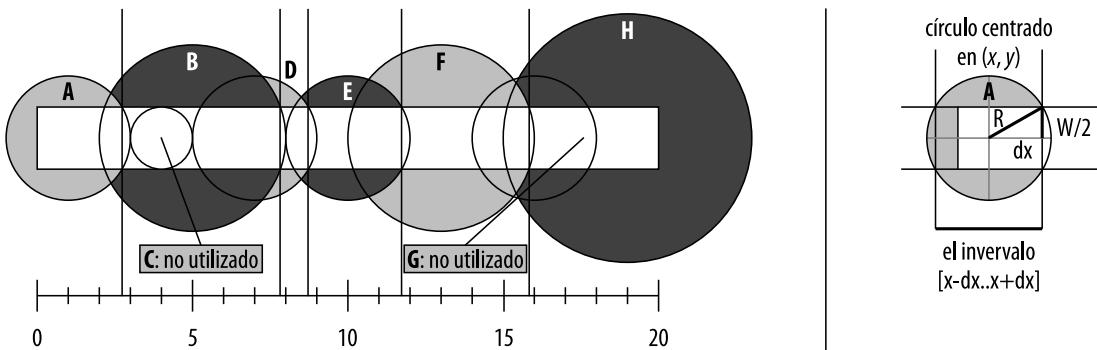


Figura 3.7: UVa 10382 - Watering Grass

En el caso mostrado en la parte izquierda de la figura 3.7, este algoritmo voraz comienza ordenando los intervalos para obtener la secuencia  $\{A, B, C, D, E, F, G, H\}$ . Despu s, los procesa de uno en uno. Comienza por 'A' (tiene que hacerlo), toma 'B' (conectado al intervalo 'A'), ignora 'C' (ya que est  integrado dentro de 'B'), toma 'D' (tiene que hacerlo, ya que 'B' y 'E' no est n conectados), toma 'E', toma 'F', ignora 'G' (ya que 'G' no est  'lo m s a la derecha posible' y tampoco llega al extremo derecho de la franja de hierba), toma 'H' (ya que est  conectado a 'F' y cubre m s zona derecha que 'G' llegando, de hecho, m s all  del final de la franja de hierba). En total, seleccionamos 6 aspersores:  $\{A, B, D, E, F, H\}$ , que ser  el m nimo posible para este caso de prueba.

#### UVa 11292 - Dragon of Loowater (ordenar primero la entrada)

Enunciado del problema: hay  $n$  cabezas de drag n y  $m$  caballeros ( $1 \leq n, m \leq 20000$ ). Cada cabeza de drag n tiene un *di metro* y cada caballero tiene una *estatura*. Un caballero de estatura  $H$  puede cortar la cabeza de un drag n  $D$  si  $D \leq H$ . Cada caballero solo le puede cortar la cabeza a un drag n. Dadas las listas de di metros de las cabezas de drag n y de las estaturas de los caballeros,  es posible cortar todas las cabezas de drag n? Si la respuesta es afirmativa,  cu l es la estatura m nima total de los caballeros utilizados para cortar las cabezas?

Existen varias formas de resolver este problema, pero vamos a ilustrar la que es, probablemente, m s sencilla. Este es un problema de emparejamiento bipartito (que veremos con m s detalle en la secci n 4.7.4), en el sentido de que debemos emparejar a ciertos caballeros con cabezas de drag n de forma maximalista. Sin embargo, tambi n se puede resolver de forma voraz: cada cabeza de drag n debe ser cortada por el caballero de menos estatura posible, siendo \'sta, al menos, igual a su di metro. Pero recibiremos la entrada en un orden arbitrario. Si ordenamos ambas listas en  $O(n \log n + m \log m)$ , podemos utilizar el siguiente barrido de complejidad  $O(\max(n, m))$  para obtener la respuesta. Aqu  tenemos otro ejemplo donde ordenar los datos de entrada pude llevarnos a la estrategia voraz requerida.

```

1 gold = d = k = 0; // arrays dragon/knight ordenados en orden no decreciente
2 while (d < n && k < m) { // todav a hay cabezas de dragones o caballeros
3 while (k < m && dragon[d] > knight[k]) k++; // buscar caballero requerido
4 if (k == m) break; // ning n caballero puede matar a esta cabeza de drag n

```

```

5 gold += knight[k]; // el rey paga esta cantidad de oro
6 d++; k++; // siguientes cabeza de dragón y caballero, por favor
7 }
8
9 if (d == n) printf("%d\n", gold); // todos los dragones decapitados
10 else printf("Loowater is doomed!\n");

```

### Ejercicio 3.4.1.1\*

¿Cuál de los siguientes conjuntos de monedas (todos en céntimos) se pueden resolver utilizando el algoritmo voraz de ‘cambio de monedas’, tratado en esta sección? Si el algoritmo voraz falla en un conjunto dado de denominaciones de monedas, determina el contraejemplo más pequeño de  $V$  céntimos en el que no es óptimo. Consulta [51] para más información sobre la búsqueda de estos contraejemplos.

1.  $S_1 = \{10, 7, 5, 4, 1\}$
2.  $S_2 = \{64, 32, 16, 8, 4, 2, 1\}$
3.  $S_3 = \{13, 11, 7, 5, 3, 2, 1\}$
4.  $S_4 = \{7, 6, 5, 4, 3, 2, 1\}$
5.  $S_5 = \{21, 17, 11, 10, 1\}$

### Comentarios sobre algoritmos voraces en concursos de programación

En esta sección hemos tratado tres problemas clásicos que se pueden resolver mediante algoritmos voraces: cambio de monedas (el caso especial), balanceo de carga y cobertura de intervalos. Conviene recordar las soluciones (excepción que confirma la regla, pues aquí podemos ignorar lo dicho anteriormente sobre no fiarse excesivamente de la memoria). También hemos visto una estrategia importante de solución de problemas que, normalmente, es aplicable a los problemas voraces: ordenar los datos de entrada para revelar soluciones ocultas.

En este libro encontraremos otros dos ejemplos clásicos de algoritmos voraces, como son el algoritmo de Kruskal (y Prim), para el problema del árbol recubridor mínimo (MST) (ver la sección 4.3) y el algoritmo de Dijkstra, para el problema del camino más corto desde un origen único (SSSP) (ver la sección 4.4.3). Existen muchos otros algoritmos voraces conocidos que hemos decidido no abordar, ya que son demasiado ‘específicos del problema’, y que raramente encontraremos en concursos de programación, como el código Huffman [7, 38], la mochila fraccional [7, 38], algunos problemas de organización de tareas, etc.

Sin embargo, en los concursos de programación actuales (tanto en el ICPC como en la IOI), raramente encontraremos las versiones canónicas de estos problemas clásicos. Utilizar algoritmos voraces para resolver problemas ‘no clásicos’ conlleva sus riesgos. Un algoritmo voraz normalmente no obtendrá un veredicto TLE, ya que suelen ser bastante rápidos, pero tienen tendencia a encontrarse con un WA. Demostrar que cierto problema ‘no clásico’ tiene subestructuras óptimas y la propiedad voraz, durante la celebración del concurso, puede resultar complicado o

consumir demasiado tiempo, así que el programador competitivo debería guiarse por la siguiente regla: si el tamaño de la entrada es ‘suficientemente pequeño’ para posibilitar una complejidad de tiempo que encaje bien en la búsqueda completa, o bien en la programación dinámica (ver la sección 3.5), será preferible utilizar una de las dos para asegurarnos una respuesta correcta. Solo utilizaremos un algoritmo voraz si el tamaño de la entrada que establece el enunciado del problema es demasiado grande para cualquiera de ellas, aun en el mejor de los casos.

Dicho esto, cada vez es más cierto que los autores de problemas intentan establecer los límites al tamaño de la entrada de los problemas en un punto en el que la decisión de utilizar, o no, un algoritmo voraz pueda resultar ambigua, de forma que los concursantes *no puedan* establecer rápidamente cuál es la mejor estrategia a seguir.

Debemos insistir en que resulta bastante complicado encontrar nuevos problemas voraces diferentes a los clásicos. Por lo tanto, el número de problemas voraces novedosos que encontraremos en la programación competitiva es menor que los que se resuelven mediante búsqueda completa o programación dinámica.

## Ejercicios de programación

**Ejercicios de programación que se resuelven con un algoritmo voraz (omitimos la mayoría de los comentarios para mantener los problemas interesantes):**

### Clásicos

- |                                                |                                                                                                                                             |
|------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00410 - Station Balance                 | (balanceo de carga)                                                                                                                         |
| 2. UVa 01193 - Radar Install...                | (LA 2519 - Beijing02; cobertura de intervalos)                                                                                              |
| 3. UVa 10020 - Minimal Coverage                | (cobertura de intervalos)                                                                                                                   |
| 4. UVa 10382 - Watering Grass                  | (cobertura de intervalos)                                                                                                                   |
| 5. <b>UVa 11264 - Coin Collector *</b>         | (variante del cambio de monedas)                                                                                                            |
| 6. <b>UVa 11389 - The Bus Driver Problem *</b> | (balanceo de carga)                                                                                                                         |
| 7. UVa 12321 - Gas Station                     | (cobertura de intervalos)                                                                                                                   |
| 8. <b>UVa 12405 - Scarecrow *</b>              | (problema de cobertura de intervalos más sencillo)                                                                                          |
| 9. IOI 2011 - Elephants                        | (se puede utilizar una solución voraz optimizada hasta la subtarea 3, pero las subtareas 4 y 5 necesitan una estructura de datos eficiente) |

**Implican ordenación (o la entrada ya está ordenada)**

1. UVa 10026 - Shoemaker's Problem
2. *UVa 10037 - Bridge*
3. UVa 10249 - The Grand Dinner
4. UVa 10670 - Work Reduction
5. UVa 10763 - Foreign Exchange
6. UVa 10785 - The Mad Numerologist
7. **UVa 11100 - The Trip, 2007 \***
8. UVa 11103 - WFF'N Proof
9. *UVa 11269 - Setting Problems*
10. **UVa 11292 - Dragon of Loowater \***
11. UVa 11369 - Shopaholic
12. UVa 11729 - Commando War

13. UVa 11900 - Boiled Eggs
14. **UVa 12210 - A Match Making Problem** \*
15. UVa 12485 - Perfect Choir

#### No clásicos, normalmente difíciles

1. UVa 00311 - Packets
2. *UVa 00668 - Parliament*
3. UVa 10152 - ShellSort
4. UVa 10340 - All in All
5. UVa 10440 - Ferry Loading II
6. UVa 10602 - Editor Nottobad
7. **UVa 10656 - Maximum Sum (II)** \*
8. UVa 10672 - Marbles on a tree
9. UVa 10700 - Camel Trading
10. UVa 10714 - Ants
11. **UVa 10718 - Bit Mask** \*
12. *UVa 10982 - Troublemakers*
13. UVa 11054 - Wine Trading in Gergovia
14. **UVa 11157 - Dynamic Frog** \*
15. *UVa 11230 - Annoying painting tool*
16. *UVa 11240 - Antimonotonicity*
17. *UVa 11335 - Discrete Pursuit*
18. UVa 11520 - Fill the Square
19. UVa 11532 - Simple Adjacency ...
20. UVa 11567 - Moliu Number Generator
21. *UVa 12482 - Short Story Competition*

## 3.5 Programación dinámica

La programación dinámica (en adelante DP) es, quizá, la técnica de solución de problemas más desafiante de los cuatro paradigmas que tratamos en el presente capítulo. Por ello, asegúrate de dominar los mecanismos tratados hasta ahora antes de leer esta sección. Prepárate también para encontrar infinidad de relaciones recursivas y recurrentes.

Las habilidades clave que debes desarrollar para dominar la DP son aquellas que te ayuden a determinar los *estados* del problema y las relaciones o *transiciones* entre los problemas principales y sus subproblemas. Ya hemos utilizado estas habilidades en el *backtracking* recursivo (ver la sección 3.2.2). De hecho, los problemas de DP con tamaños de entrada pequeños podrían ser resueltos mediante *backtracking* recursivo.

Si eres un recién llegado a la DP, puedes empezar asumiendo que la DP ('de arriba a abajo') es una forma 'inteligente', o 'más rápida', de *backtracking* recursivo. En esta sección, explicaremos las razones por las que la DP es, normalmente, más rápida que el *backtracking* recursivo, en aquellos problemas abordables por ella.

La DP se utiliza principalmente para resolver problemas de *optimización* y de *conteo*. Si encuentras un problema que diga “minimiza esto”, “maximiza aquello” o “de cuántas formas se puede hacer lo otro”, existe una (alta) probabilidad de que te encuentres ante un problema de DP. La mayoría de estos problemas, en los concursos de programación, solo piden el valor óptimo/total y no la solución óptima, lo que hace que el problema sea más fácil de resolver, al eliminar la necesidad de buscar esa solución. Sin embargo, otros problemas de DP más difíciles también pedirán, de alguna manera, la solución óptima. En esta sección, iremos afinando progresivamente nuestra comprensión de la programación dinámica.

### 3.5.1 Ilustración de programación dinámica

Ilustraremos el concepto de programación dinámica mediante un problema de ejemplo, UVa 11450 - Wedding Shopping. El enunciado resumido del problema dice: dadas diferentes opciones de cada prenda de ropa (por ejemplo, 3 modelos de camisa, 2 de cinturón, 4 de zapatos, ...) y un cierto presupuesto *limitado*, nuestra tarea consiste en *comprar un modelo de cada prenda*. No podemos gastar más dinero del que tenemos como presupuesto, pero sí queremos gastar la *máxima cantidad posible*.

La entrada consta de dos enteros  $1 \leq M \leq 200$  y  $1 \leq C \leq 20$ , donde  $M$  es el presupuesto y  $C$  el número de prendas que debemos comprar, seguido de la información relativa a las  $C$  prendas. Para cada prenda  $g \in [0..C-1]$ , recibiremos un entero  $1 \leq K \leq 20$ , que indica el número de modelos diferentes de  $g$ , seguido de  $K$  enteros, que indican el precio de cada modelo  $\in [1..K]$  de dicha prenda  $g$ .

La salida será un entero que indica la cantidad máxima de dinero que podemos gastar, comprando una prenda de cada, *sin superar el presupuesto*. Si no existe solución porque el presupuesto es demasiado pequeño, deberemos devolver el texto “*no solution*”.

Supongamos que tenemos el caso de prueba A, donde  $M = 20$  y  $C = 3$ :

- Precio de los 3 modelos de la prenda  $g = 0 \rightarrow \underline{6} \ 4 \ 8 // \text{no ordenados en la entrada}$
- Precio de los 2 modelos de la prenda  $g = 1 \rightarrow \underline{5} \ 10$
- Precio de los 4 modelos de la prenda  $g = 2 \rightarrow \underline{1} \ 5 \ 3 \ 5$

En este caso de prueba, la respuesta es 19, que *podría* resultar de la compra de los elementos subrayados ( $8+10+1$ ). Pero no es única, ya que ( $6+10+3$ ) y ( $4+10+5$ ) también son óptimas.

Sin embargo, supongamos que tenemos el caso de prueba B, donde  $M = 9$  (**presupuesto limitado**) y  $C = 3$ :

- Precio de los 3 modelos de la prenda  $g = 0 \rightarrow 6 \ 4 \ 8$
- Precio de los 2 modelos de la prenda  $g = 1 \rightarrow 5 \ 10$
- Precio de los 4 modelos de la prenda  $g = 2 \rightarrow 1 \ 5 \ 3 \ 5$

La respuesta será “*no solution*” ya que, aunque compramos el modelo más barato de cada prenda, el precio total ( $4+5+1 = 10$ ) superará nuestro presupuesto de  $M = 9$ .

Para apreciar mejor la utilidad de la programación dinámica para resolver el problema anterior, comprobemos hasta dónde llegaríamos en este problema en particular utilizando las *otras técnicas* vistas anteriormente.

### Técnica 1: voraz (respuesta incorrecta)

Como queremos maximizar el presupuesto que gastamos, una idea voraz (hay otras, que también resultarían en WA) consiste en tomar el modelo más caro de cada prenda  $g$  que podamos comprar, según nuestro presupuesto. Por ejemplo, en el caso A anterior, podemos elegir el modelo 3 de la prenda  $g = 0$ , con un precio de 8 (el dinero restante será de  $20 - 8 = 12$ ), después el modelo 2 de la prenda  $g = 1$  con un precio de 10 (dinero  $12 - 10 = 2$ ) y, por último, para la prenda  $g = 2$  solo podremos elegir el modelo 1, con un precio de 1, ya que el dinero restante no nos permite adquirir los modelos con precios 3 o 5. Esta estrategia voraz ‘funciona’ en los casos de prueba A y B que hemos presentado antes, y llega a las mismas soluciones óptimas de  $(8+10+1) = 19$  y “no solution”, respectivamente. Es, además, una solución muy rápida<sup>8</sup>:  $20+20+\dots+20$  hasta un total de 20 veces = 400 operaciones en el peor de los casos. Sin embargo, esta estrategia voraz no funcionará en muchos otros casos de prueba, como muestra el siguiente *contraejemplo* (caso de prueba C):

Caso de prueba C con  $M = 12$  y  $C = 3$ :

- 3 modelos de la prenda  $g = 0 \rightarrow 6 \underline{4} \ 8$
- 2 modelos de la prenda  $g = 1 \rightarrow \underline{5} \ 10$
- 4 modelos de la prenda  $g = 2 \rightarrow 1 \ 5 \underline{3} \ 5$

La estrategia voraz selecciona el modelo 3 de la prenda  $g = 0$  que tiene precio **8** (dinero  $12 - 8 = 4$ ), lo que provoca que no nos quede dinero suficiente para comprar ningún modelo de la prenda  $g = 1$ , lo que provocaría una respuesta incorrecta de “no solution”. Una solución óptima es  $\underline{4+5+3} = 12$ , lo que emplea todo el presupuesto disponible. Esta solución no es única, ya que  $6+5+1 = 12$  también consume todo el dinero.

### Técnica 2: divide y vencerás (respuesta incorrecta)

Este problema no se puede resolver utilizando el paradigma de divide y vencerás. Esto es debido a que los subproblemas (explicado en la siguiente subsección de búsqueda completa) no son independientes. Por lo tanto, no son resolubles por separado.

### Técnica 3: búsqueda completa (tiempo límite excedido)

Ahora veamos si la búsqueda completa (*backtracking* recursivo) puede resolver el problema. Una forma de utilizar *backtracking* recursivo consiste en escribir una función `comprar(dinero, g)` con dos parámetros: el *dinero* que nos queda y la prenda  $g$  que estamos intentando comprar. La pareja *(dinero, g)* constituye el *estado* del problema. Hay que fijarse en que el orden de los parámetros no influye, es decir, *(g, dinero)* sería un estado perfectamente válido. Más adelante, en la sección 3.5.3, veremos de forma más detallada cómo seleccionar los estados apropiados para un problema.

Comenzamos con *dinero* =  $M$  y prenda  $g = 0$ . A continuación, probamos con todos los modelos posibles de la prenda  $g = 0$  (un máximo de 20). Si seleccionamos el modelo  $i$ , restamos su

<sup>8</sup>No es necesario ordenar los precios para encontrar el más alto, ya que solo tenemos  $K \leq 20$  modelos. Una búsqueda de complejidad  $O(K)$  es suficiente.

precio del *dinero*, y repetimos el proceso de forma recursiva con la prenda  $g = 1$  (que también puede tener hasta 20 modelos), etc. Podemos detenernos cuando hayamos elegido el modelo de la última prenda  $g = C - 1$ . Si  $dinero < 0$  antes de llegar a elegir un modelo de la prenda  $g = C - 1$ , podemos podar esa solución, ya que no es válida. Finalmente, elegiremos la solución que resulte en la menor cantidad no negativa de *dinero*, de entre todas las que son válidas. Esto maximizará el dinero gastado, que será ( $M - \text{dinero}$ ).

Podemos definir formalmente estas recurrencias (transiciones) de búsqueda completa así:

1. Si  $\text{dinero} < 0$  (es decir, si el dinero es negativo),  $\text{comprar}(\text{dinero}, g) = -\infty$  (en la práctica, podemos devolver un valor negativo grande).
2. Si hemos comprado un modelo de la última prenda, es decir,  $g = C$ ,  $\text{comprar}(\text{dinero}, g) = M - \text{dinero}$  (que es el dinero total gastado).
3. El caso general,  $\forall \text{modelo} \in [1..K]$  de la prenda actual  $g$ ,  $\text{comprar}(\text{dinero}, g) = \max(\text{comprar}(\text{dinero}-\text{precio}[g][\text{modelo}], g+1))$ . Queremos maximizar este valor (recuerda que los valores no válidos son un negativo grande).

Esta solución es correcta, pero **demasiado lenta**. Vamos a analizar la complejidad de tiempo del peor caso. En el caso de prueba más grande, la prenda  $g = 0$  tiene hasta 20 modelos; la prenda  $g = 1$  *también* tiene hasta 20 modelos, y todas las prendas, incluyendo la última  $g = 19$ , *también* tienen hasta 20 modelos. Por lo tanto, esta búsqueda completa consumirá, en el peor de los casos,  $20 \times 20 \times \dots \times 20$  operaciones, lo que supone  $20^{20}$  = un número **muy grande**. Si solo tenemos como opción esta solución de búsqueda completa, no podremos resolver el problema.

#### Técnica 4: DP de arriba a abajo (aceptada)

Para resolver este problema, tenemos que utilizar el concepto de DP, ya que se cumplen dos requisitos previos para que ésta sea aplicable:

1. Este problema tiene subestructuras óptimas<sup>9</sup>. Podemos verlo ilustrado en la tercera recurrencia de búsqueda completa mencionada anteriormente: la solución del subproblema forma parte de la solución del problema completo. En otras palabras, si elegimos el modelo  $i$  de la prenda  $g = 0$ , para que nuestra selección final sea óptima, las posteriores elecciones para  $g = 1$  y siguientes, también deberán ser óptimas, con un presupuesto reducido de  $M - \text{precio}$ , donde *precio* es el precio del modelo  $i$ .
2. Este problema tiene subproblemas superpuestos. Ésta es la característica clave de la DP. El espacio de búsqueda de este problema *no es* tan grande como el cálculo aproximado de  $20^{20}$  que ya hemos mencionado, porque **muchos** de los subproblemas se *superponen*.

Vamos a comprobar si este problema tiene, de hecho, subproblemas superpuestos. Supongamos que tenemos 2 modelos de una determinada prenda  $g$  con el *mismo* precio  $p$ . En este caso, la búsqueda completa analizará el *mismo* subproblema  $\text{comprar}(\text{dinero}-p, g+1)$ , después de seleccionar *cualquiera* de los modelos. Esta situación también se producirá si alguna combinación de *dinero* y el precio del modelo elegido provoca que  $\text{dinero}_1 - p_1 = \text{dinero}_2 - p_2$  para la misma

---

<sup>9</sup>Los algoritmos voraces también requieren de subestructuras óptimas, pero en este caso el problema carece de la ‘propiedad voraz’, lo que lo hace irresoluble de esa manera.

prenda  $g$ . Esto causará, en la solución de búsqueda completa, que el mismo subproblema sea calculado *más de una vez*, lo que es, a todas luces, poco eficiente.

Así pues, ¿cuántos subproblemas *distintos* (o **estados**, en terminología de DP) existen en el problema? Solo  $201 \times 20 = 4020$ . No hay más que 201 valores de *dinero* posibles (de 0 a 200, ambos inclusive) y 20 valores de prenda  $g$  (de 0 a 19, ambos inclusive). Cada subproblema debe ser calculado solo *una vez*. Si podemos garantizar esto, la solución será *mucho más rápida*.

La implementación de esta solución de DP es sorprendentemente sencilla. Si ya tenemos la solución de *backtracking* recursivo (ver las recurrencias, o **transiciones** en términos de DP, que aparecen mencionadas en la técnica de búsqueda completa), podemos implementar la DP **de arriba a abajo** añadiendo estos dos pasos:

1. Inicializar<sup>10</sup> una tabla ‘recordatoria’ de DP con valores nulos que no se utilizarán en el problema, como por ejemplo ‘-1’. La tabla de DP deberá tener las dimensiones correspondientes a los estados del problema.
2. Comprobar si un estado ya ha sido calculado al principio de la función recursiva:
  - a) Si lo ha sido, basta con devolver el valor de la tabla recordatoria de DP, complejidad  $O(1)$ . Este es el origen del término *memoización*.
  - b) Si no lo ha sido, realizar los cálculos de forma normal (una sola vez) y almacenar el valor obtenido en la tabla recordatoria de DP, para que en las *llamadas siguientes* a este subproblema (estado) podamos dar una respuesta inmediata.

Analizar una solución básica<sup>11</sup> de DP es sencillo. Si existen  $M$  estados distintos, es necesario un espacio de memoria  $O(M)$ . Si el cálculo de un estado (la complejidad de la transición de DP) necesita  $O(k)$  pasos, la complejidad de tiempo global será de  $O(kM)$ . El problema UVa 11450 tiene  $M = 201 \times 20 = 4020$  y  $k = 20$  (ya que tendremos que iterar sobre 20 modelos de cada prenda  $g$  como mucho). Por lo tanto, la complejidad de tiempo será, como máximo, de  $4020 \times 20 = 80400$  operaciones por cada caso de prueba, lo que resulta muy manejable.

A continuación, mostramos nuestro código para ilustrarlo, especialmente para aquellos que nunca hayan programado un algoritmo de DP de arriba a abajo. Estudia el código y verifica que es, de hecho, muy similar al del *backtracking* recursivo que has visto en la sección 3.2.

```
1 // UVa 11450 - Wedding Shopping - de arriba a abajo
2 // este código es similar al del backtracking recursivo
3 // las partes específicas de la DP de arriba a abajo se indican con T->D
4 // utilizamos un array un poco más grande para evitar desbordamiento
5 #include <bits/stdc++.h>
6 using namespace std;
7
8 int i, j, TC, ans, M, C, price[30][30]; // price[g (<= 20)][model (<= 20)]
9 int memo[210][30]; // T->D: tabla recordatoria DP[money (<= 20)][g (<= 20)]
10
```

<sup>10</sup>En C/C++ la función `memset` de `<cstring>` es una buena forma de hacerlo.

<sup>11</sup>Por básica, nos referimos a “sin optimizaciones específicas que veremos más adelante, tanto en esta sección como en la sección 8.3”.

```

11 int shop(int money, int g) {
12 if (money < 0) return -1000000000; // falla, devuelve negativo grande
13 if (g == C) return M-money; // hemos comprado la última, hecho
14 // si comentamos la siguiente linea, la DP se convierte en backtracking
15 if (memo[money][g] != -1) return memo[money][g]; // T->D: memoización
16 int ans = -1; // empezar con un negativo porque los precios no lo son
17 for (int model = 1; model <= price[g][0]; model++) // probar cada modelo
18 ans = max(ans, shop(money-price[g][model], g+1));
19 return memo[money][g] = ans; } // T->D: memoizar ans y devolverlo
20
21 int main() { // fácil de programar si estás familiarizado
22 scanf("%d", &TC);
23 while (TC--) {
24 scanf("%d %d", &M, &C);
25 for (i = 0; i < C; i++) {
26 scanf("%d", &price[i][0]); // número de modelos en price[i][0]
27 for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
28 }
29 memset(memo, -1, sizeof memo); // T->D: inicializar tabla recordatoria
30 ans = shop(M, 0); // iniciar la DP de arriba a abajo
31 if (ans < 0) printf("no solution\n");
32 else printf("%d\n", ans);
33 } } // return 0;

```

Queremos aprovechar esta oportunidad para ilustrar otro estilo utilizado en la implementación de soluciones de DP (solo aplicable a los usuarios de C/C++). En vez de dirigirnos repetidamente a una determinada celda de la tabla recordatoria, podemos utilizar una variable de *referencia* local, para almacenar la dirección de memoria de la celda correspondiente en la tabla, como se puede ver a continuación. Ambos estilos de programación son similares, y la decisión de utilizar uno u otro depende de las preferencias del programador.

```

1 int shop(int money, int g) {
2 if (money < 0) return -1000000000; // orden de >1 casos base importante
3 if (g == C) return M-money; // money no puede ser <0 si llegamos aquí
4 int &ans = memo[money][g]; // recordar la dirección de memoria
5 if (ans != -1) return ans;
6 for (int model = 1; model <= price[g][0]; model++)
7 ans = max(ans, shop(money-price[g][model], g+1));
8 return ans; // ans (o memo[money][g]) se actualiza directamente
9 }

```



ch3\_02\_UVa11450\_td.cpp



ch3\_02\_UVa11450\_td.java

### Técnica 5: DP de abajo a arriba (aceptada)

Existe otra forma de implementar una solución de DP, llamada normalmente **DP de abajo a arriba**. Esta es, en realidad, la ‘versión auténtica’ de DP, ya que la DP era conocida originalmente como ‘el método tabular’ (técnica de cálculo que utiliza una tabla). Los pasos *básicos* para crear una solución de DP de abajo a arriba son los siguientes:

1. Determinar el conjunto de parámetros necesario que describan, de forma única, el problema (el estado). Este paso es similar al tratado en el *backtracking* recursivo en la DP de arriba a abajo.
2. Si hay  $N$  parámetros requeridos para representar el estado, preparar una tabla de DP de  $N$  dimensiones, con una única entrada por estado. Esto es equivalente a la tabla recordatoria de la DP de arriba a abajo. Sin embargo, hay diferencias. En la DP de abajo a arriba, solo será necesario inicializar algunas celdas de la tabla, con valores iniciales conocidos (los casos base). Recuerda que en la DP de arriba a abajo hemos inicializado la tabla recordatoria completamente, utilizando valores nulos (normalmente -1), para indicar que todavía no hemos calculado esos valores.
3. Por último, con los valores de casos base ya introducidos en la tabla, determinar las celdas/estados que pueden llenarse a continuación (las transiciones). Repetir este proceso hasta que la tabla de DP esté completa. En la DP de abajo a arriba esta parte se logra, normalmente, a través de iteraciones, utilizando bucles (otros detalles más adelante).

En el caso del UVa 11450, podemos escribir la DP de abajo a arriba de la siguiente manera: describimos el estado de un subproblema con dos parámetros, la prenda  $g$  y el *dinero* actuales. Esta formulación de los estados es, en esencia, equivalente al estado que hemos utilizado en la DP de arriba a abajo, salvo que hemos invertido el orden, de forma que  $g$  sea el primer parámetro (por lo que los valores de  $g$  son los índices de las filas de la tabla de DP y, así, podremos obtener ventaja del mejor aprovechamiento que hace de la caché el acceso a las filas en un *array* bidimensional, ver los trucos mencionados en la sección 3.2.3). A continuación, inicializamos una tabla bidimensional (matriz booleana) `accesible[g][dinero]`, de tamaño  $20 \times 201$ . Inicialmente, solo estableceremos como activas las celdas/estados accesibles con la compra de alguno de los modelos de la primera prenda  $g = 0$  (en la primera fila). Utilizaremos el caso de prueba A, visto anteriormente, como ejemplo. En la parte superior de la figura 3.8, solo las columnas ‘20-6 = 14’, ‘20-4 = 16’ y ‘20-8 = 12’ de la fila 0 están activadas.

A continuación, realizamos un bucle desde la segunda prenda  $g = 1$  (segunda fila) hasta la última  $g = C - 1 = 3 - 1 = 2$  (tercera y última fila), en orden creciente (fila por fila). Si `accesible[g-1][dinero]` es cierto, el siguiente estado `accessible[g][dinero-p]`, donde  $p$  es el precio de un modelo de la prenda actual  $g$ , también es accesible, siempre que el segundo parámetro (*dinero*) no sea negativo. En la parte central de la figura 3.8, vemos que `accesible[0][16]` se propaga a `accesible[1][16-5]` y `accesible[1][16-10]`, cuando se compran los modelos de precio 5 y 10 de la prenda  $g = 1$ , respectivamente. `accesible[0][12]` resulta en `accesible[1][12-10]`, cuando se compra el modelo de precio 10 de  $g = 1$ , y así sucesivamente. Repetiremos este proceso, fila por fila, hasta que hayamos terminado<sup>12</sup>.

<sup>12</sup>En la sección 4.7.1 veremos un modelo de DP como recorrido de un DAG (implícito). Para evitar un *backtracking* innecesario a lo largo del DAG, debemos visitar los vértices en su orden topológico (ver la sección 4.2.5). El orden en el que rellenamos la tabla de DP, es el topológico del DAG implícito que la soporta.

|   |   | dinero → |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
|---|---|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| g |   | 0        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0 | 0 | 0        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |    |
| 1 | 0 | 0        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |
| 2 | 0 | 0        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |

|   |   | dinero → |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
|---|---|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| g |   | 0        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0 | 0 | 0        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |    |
| 1 | 0 | 0        | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |
| 2 | 0 | 0        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |

|   |   | dinero → |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
|---|---|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| g |   | 0        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0 | 0 | 0        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |    |
| 1 | 0 | 0        | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |
| 2 | 0 | 1        | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |

Figura 3.8: DP de abajo a arriba (las columnas 21 a 200 no aparecen por razón de espacio)

Por último, podemos encontrar la respuesta en la última fila, donde  $g = C - 1$ . Buscaremos el estado en esa fila, cuyo índice sea alcanzable y lo más cercano a 0. En la parte inferior de la figura 3.8, la celda `accesible[2][1]` contiene la respuesta. Esto significa que podemos llegar al estado ( $dinero = 1$ ) comprando alguna combinación de las prendas. La respuesta final es, en realidad,  $M - dinero$  o, en este caso,  $20 - 1 = 19$ . La respuesta será “no solution” si ningún estado de la última fila es accesible (donde `accesible[C-1][dinero]` esté activado). Incluimos nuestra implementación del problema para su comparación con la versión ‘de arriba a abajo’:

```

1 // UVa 11450 - Wedding Shopping - de abajo a arriba
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 int g, money, k, TC, M, C;
6 int price[30][30]; // price[g (<= 20)][model (<= 20)]
7 bool reachable[30][210]; // tabla reachable[g (<= 20)][money (<= 200)]
8
9 int main() {
10 scanf("%d", &TC);
11 while (TC--) {
12 scanf("%d %d", &M, &C);
13 for (g = 0; g < C; g++) {
14 scanf("%d", &price[g][0]); // guardar número de modelos en price[g][0]
15 for (k = 1; k <= price[g][0]; k++)
16 scanf("%d", &price[g][k]);
17 }
18
19 memset(reachable, false, sizeof reachable); // limpiar todo
20 // valores iniciales (casos base), usando la primera prenda g = 0
21 for (k = 1; k <= price[0][0]; k++)
22 if (M - price[0][k] >= 0)
 reachable[0][M - price[0][k]] = true;

```

```

23
24 for (g = 1; g < C; g++) // para cada prenda restante
25 for (money = 0; money < M; money++) if (reachable[g-1][money])
26 for (k = 1; k <= price[g][0]; k++) if (money-price[g][k] >= 0)
27 reachable[g][money-price[g][k]] = true; // ahora alcanzable
28
29 for (money = 0; money <= M && !reachable[C-1][money]; money++);
30
31 if (money == M+1) printf("no solution\n"); // última fila sin bit activo
32 else printf("%d\n", M-money);
33 }
34 } // return 0;

```



ch3\_03\_UVa11450\_bu.cpp



ch3\_03\_UVa11450\_bu.java

Escribir soluciones de DP en el estilo de abajo a arriba tiene una ventaja en aquellos problemas en los que solo necesitamos la última fila de la tabla de DP (o, de forma general, la última actualización de todos los estados) para determinar la solución (como en este problema), ya que podemos optimizar el *uso de memoria* de nuestra solución, al sacrificar una dimensión en nuestra tabla de DP. En los problemas de DP con límites estrictos de memoria, este ‘truco para ahorrar espacio’ puede resultar útil, aunque la complejidad de tiempo global no se verá afectada.

Volvamos a la figura 3.8. Solo necesitamos almacenar dos filas, la que estemos procesando y la anterior ya procesada. Para calcular la fila 1, solo necesitamos saber qué columnas de la fila 0 están activadas como *accesible*. Para calcular la fila 2, igualmente solo necesitamos conocer las columnas de la fila 1 ya activadas. En general, para calcular cualquier fila  $g$ , solo será necesario conocer los valores de la fila  $g - 1$ . De esta forma, en vez de almacenar una matriz booleana *accesible*[ $g$ ][dinero] de tamaño  $20 \times 201$ , nos basta con almacenar *accesible*[2][dinero] de tamaño  $2 \times 201$ . Podemos aprovechar esta técnica de programación para referirnos a una fila como ‘anterior’ y a otra como ‘actual’ (por ejemplo, *ant* = 0, *act* = 1) y, después, intercambiarlas (ahora, *ant* = 1, *act* = 0) según vayamos realizando los cálculos. En este problema concreto, el ahorro de memoria no es significativo, pero en problemas de DP más complejos, por ejemplo donde hubiese miles de modelos de prendas en vez de 20, este ahorro de espacio podría ser importante.

### DP de arriba a abajo frente a de abajo a arriba

Aunque ambos estilos utilizan ‘tablas’, la forma en que se rellena la tabla de DP del método de abajo a arriba es diferente a la tabla *recordatoria* de arriba a abajo. En la DP de arriba a abajo, las entradas de la tabla recordatoria se van llenando ‘a demanda’, a través de la propia recursión. En la DP de abajo a arriba, hemos utilizando un ‘orden de llenado de la tabla de DP’ correcto para calcular los valores, de forma que ya hemos obtenido anteriormente los datos necesarios para procesar la celda actual. Esta forma de llenar la tabla, supone que se hace en el orden topológico del DAG implícito (lo veremos explicado con más detalle en la sección

4.7.1), en la estructura de recurrencia. En la mayoría de los problemas de DP, se puede lograr un orden topológico a través de la secuencia adecuada de algunos bucles (anidados).

La mayor parte de las veces, los dos estilos son igual de eficientes y la decisión de utilizar uno u otro es una cuestión de gustos. Sin embargo, en los problemas de DP más difíciles, un estilo podría ser mejor que el otro. Para ayudarte a entender qué estilo deberías utilizar al enfrentarte a un problema de DP, te recomendamos estudiar las diferencias entre las DP de arriba a abajo y de abajo a arriba que mostramos en la tabla 3.2.

| De arriba a abajo                                                                                                                                                                                                                                                                                                                                                                                                           | De abajo a arriba                                                                                                                                                                                                                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Pros:</b>                                                                                                                                                                                                                                                                                                                                                                                                                | <b>Pros:</b>                                                                                                                                                                                                                                                                                                                       |
| <ol style="list-style-type: none"> <li>1. Es una evolución natural de la búsqueda completa</li> <li>2. Calcula los subproblemas solo cuando es necesario (a veces esto es más rápido)</li> </ol>                                                                                                                                                                                                                            | <ol style="list-style-type: none"> <li>1. Más rápido si hay que volver a visitar muchos subproblemas, ya que no hay sobrecarga por las llamadas recursivas</li> <li>2. Puede ahorrar memoria con el ‘truco de ahorro de espacio’</li> </ol>                                                                                        |
| <b>Contras:</b>                                                                                                                                                                                                                                                                                                                                                                                                             | <b>Contras:</b>                                                                                                                                                                                                                                                                                                                    |
| <ol style="list-style-type: none"> <li>1. Más lento si hay que volver a visitar muchos subproblemas, debido a la sobrecarga de la llamada a la función (no suele perjudicar en concursos de programación)</li> <li>2. Si hay <math>M</math> estados, se necesita una tabla de tamaño <math>O(M)</math>, lo que puede llevar a un veredicto MLE en algunos problemas difíciles (sin el truco de la sección 8.3.4)</li> </ol> | <ol style="list-style-type: none"> <li>1. Para los programadores que prefieren la recursión, este estilo puede no resultar intuitivo</li> <li>2. Si hay <math>M</math> estados, la DP de abajo a arriba visita y rellena el valor de <i>todos</i> los <math>M</math> estados, aunque muchos de ellos no sean necesarios</li> </ol> |

Tabla 3.2: Tabla de decisión de programación dinámica

### Mostrar la solución óptima

Muchos problemas de DP solo piden el valor de la solución óptima (como en el caso del UVa 11450). Sin embargo, muchos concursantes pueden verse desconcertados cuando también se pide la solución óptima misma. Conocemos dos formas de hacerlo.

La primera se utiliza sobre todo en la DP de abajo a arriba (aunque también es aplicable en la DP de arriba a abajo), donde almacenamos la información precedente a cada estado. Si existe más de un precedente óptimo y debemos mostrar todas las soluciones óptimas, podemos almacenarlos en una lista. Una vez tenemos el estado final óptimo, podemos hacer el camino a la inversa, y seguir las transiciones óptimas de cada estado hasta que lleguemos a uno de los casos base. Si el problema pide todas las soluciones óptimas, este mecanismo las mostrará todas. Sin embargo, la mayoría de autores de problemas suele establecer criterios de salida adicionales, de forma que la solución óptima sea única (para facilitar la evaluación de la solución).

Ejemplo: veamos la parte inferior de la figura 3.8. El estado final óptimo es `accesible[2][1]`. El precedente de este es el estado `accesible[1][2]`. Nos desplazamos a `accesible[1][2]`. Ahora veamos la parte central de la figura 3.8. El precedente del estado `accesible[1][2]` es `accesible[0][12]`. A continuación, nos desplazamos a este `accesible[0][12]`. Como ya nos encontramos en uno de los estados base iniciales (en la primera fila), sabemos que la solución óptima es:  $(20 \rightarrow 12) = \text{precio } 8$ , después  $(12 \rightarrow 2) = \text{precio } 10$ , después  $(2 \rightarrow 1) = \text{precio } 1$ . Sin embargo, como se mencionaba antes, en el enunciado del problema, podría haber otras soluciones óptimas. Por ejemplo, también podríamos seguir la ruta `accesible[2][1] →`

`accesible[1][6] → accesible[0][16]`, que representa otra solución óptima:  $(20 \rightarrow 16) = \text{precio } 4$ , después  $(16 \rightarrow 6) = \text{precio } 10$ , después  $(6 \rightarrow 1) = \text{precio } 5$ .

La segunda forma se aplica mayoritariamente a la DP de arriba a abajo, donde utilizamos la potencia de la recursión y la *memoización*, para realizar el mismo trabajo. Utilizando el código de DP de arriba a abajo, mostrado anteriormente en la técnica 4, podemos añadir una función `void print_shop(int money, int g)`, que tenga la misma estructura que `int shop(int money, int g)`, pero que utilice los valores almacenados en la tabla recordatoria, para reconstruir la solución. A continuación, incluimos una implementación de ejemplo (que solo mostrará una solución óptima):

```
1 void print_shop(int money, int g) { // esta función devuelve void
2 if (money < 0 || g == C) return; // casos base similares
3 for (int k = 1; k <= price[g][0]; k++) // ¿qué modelo es óptimo?
4 if (shop(money - price[g][k], g+1) == memo[money][g]) {
5 printf("%d", price[g][k], g == C-1 ? '\n' : '-'); // este
6 print_shop(money - price[g][k], g+1); // recursivo a este estado
7 break; // NO VISITAR otros estados
8 }
}
```

### Ejercicio 3.5.1.1

Para verificar tu comprensión del problema UVa 11450, tratado en esta sección, determina cuál es la salida del siguiente caso de prueba D.

Caso de prueba D con  $M = 25$  y  $C = 3$ :

Precio de los 3 modelos de la prenda  $g = 0 \rightarrow 6\ 4\ 8$

Precio de los 2 modelos de la prenda  $g = 1 \rightarrow 10\ 6$

Precio de los 4 modelos de la prenda  $g = 2 \rightarrow 7\ 3\ 1\ 5$

### Ejercicio 3.5.1.2

¿Es la siguiente formulación de estado `shop(g, model)`, donde  $g$  representa la prenda actual y *model* el modelo actual, apropiada y exhaustiva para el problema UVa 11450?

### Ejercicio 3.5.1.3

Añadir el truco de ahorro de espacio al código de DP de abajo a arriba en la técnica 5.

### 3.5.2 Ejemplos clásicos

El mencionado UVa 11450, es un problema de DP no clásico (relativamente sencillo), donde *nosotros mismos* tenemos que determinar los estados de DP y transiciones correctas. Sin embargo, hay mucho otros problemas *clásicos* con soluciones de DP eficientes, es decir, sus estados y transiciones son *conocidos*. Por ello, todo concursante que desee obtener un buen resultado en el ICPC o la IOI, debe dominar esos problemas de DP clásicos y sus soluciones. En esta sección incluimos seis de ellos. Una vez que hayas entendido los principios básicos de estas soluciones, intenta resolver los ejercicios de programación que enumeran sus *variantes*.

#### 1. Suma de rangos unidimensional máxima

Enunciado resumido de UVa 507 - Jill Rides Again: dado un *array* de enteros A, que contiene  $n \leq 20K$  enteros distintos de cero, determinar la suma de rangos máxima (unidimensional) de A. En otras palabras, encontrar la consulta de suma de rangos (RSQ) entre dos índices  $i$  y  $j$  en  $[0..n-1]$ , esto es:  $A[i] + A[i+1] + A[i+2] + \dots + A[j]$  (ver las secciones 2.4.3 y 2.4.4).

Un algoritmo de búsqueda completa que pruebe todos los  $O(n^2)$  pares de  $i$  y  $j$ , calcula la RSQ( $i$ ,  $j$ ) solicitada en  $O(n)$  y, finalmente, elige la máxima, se ejecuta con una complejidad de tiempo de  $O(n^3)$ . Con el valor de  $n$  hasta  $20K$ , es una solución TLE.

En la sección 2.4.4 hemos tratado la siguiente estrategia de DP: procesar previamente el *array* A calculando  $A[i] += A[i-1] \forall i \in [1..n-1]$ , de forma que  $A[i]$  contenga la suma de los enteros del *subarray*  $A[0..i]$ . Ahora podremos calcular RSQ( $i$ ,  $j$ ) en  $O(1)$ :  $RSQ(0, j) = A[j]$  y  $RSQ(i, j) = A[j] - A[i-1] \forall i > 0$ . Así, el algoritmo de búsqueda completa anterior se ejecuta en  $O(n^2)$ . Con  $n$  hasta  $20K$ , sigue siendo una táctica TLE. Sin embargo, esta técnica resultará útil en otros casos (ver esta suma de rango unidimensional en la sección 8.4.2).

Hay un algoritmo incluso mejor para este problema. A continuación, incluimos la parte principal del algoritmo de Jay Kadane en  $O(n)$  (se puede interpretar como voraz o DP) que lo resuelve.

```
1 // dentro de int main()
2 int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5 }; // array de ejemplo A
3 int sum = 0, ans = 0; // importante, ans se debe inicializar a 0
4 for (int i = 0; i < n; i++) { // barrido lineal, O(n)
5 sum += A[i]; // extendemos vorazmente esta suma
6 ans = max(ans, sum); // mantenemos el RSQ máximo global
7 if (sum < 0) sum = 0; // pero reiniciamos la suma
8 // si llega a bajar de 0
9 }
printf("Max 1D Range Sum = %d\n", ans);
```



ch3\_04\_Max1DRangeSum.cpp



ch3\_04\_Max1DRangeSum.java

La idea clave del algoritmo de Kadane es mantener una suma constante de los enteros vistos hasta el momento y, de forma voraz, reiniciarla a 0, si la suma constante resulta en un valor inferior a 0. Esto se debe a que volver a comenzar desde 0 siempre es mejor que continuar desde una suma constante negativa. El algoritmo de Kadane es necesario para resolver este problema UVa 507, ya que  $n \leq 20K$ .

También podemos ver el algoritmo de Kadane como una solución de DP. En cada paso tenemos dos opciones: podemos apalancar la suma máxima acumulada anterior, o comenzar un nuevo rango. La variable de DP  $dp(i)$  representa, por tanto, la suma máxima de un rango de enteros que termina con el elemento  $A[i]$ . Por ello, la respuesta final es el máximo de todos los valores de  $dp(i)$ , donde  $i \in [0..n-1]$ . Si se permiten rangos de longitud cero, entonces 0 debe considerarse también una respuesta posible. La implementación anterior es, esencialmente, una versión eficiente que utiliza el truco de ahorro de espacio ya mencionado.

## 2. Suma de rangos bidimensional máxima

Enunciado resumido de UVa 108 - Maximum Sum: dada una matriz de enteros cuadrada  $A$   $n \times n$  ( $1 \leq n \leq 100$ ), donde cada entero tiene un valor en el rango  $[-127..127]$ , encontrar una submatriz de  $A$  con la suma máxima. Por ejemplo: la matriz  $4 \times 4$  ( $n = 4$ ) de la tabla 3.3.A, tiene una submatriz  $3 \times 2$  en la parte inferior izquierda, con una suma máxima de  $9 + 2 - 4 + 1 - 1 + 8 = 15$ .

| <b>A</b> | 0 | -2 | -7 | 0 | <b>B</b> | 0  | -2  | -9 | -9 | <b>C</b> | 0  | -2  | -9 | -9 |
|----------|---|----|----|---|----------|----|-----|----|----|----------|----|-----|----|----|
| 9        | 2 | -6 | 2  |   | 9        | 9  | -4  | 2  |    | 9        | 9  | -4  | 2  |    |
| -4       | 1 | -4 | 1  |   | 5        | 6  | -11 | -8 |    | 5        | 6  | -11 | -8 |    |
| -1       | 8 | 0  | -2 |   | 4        | 13 | -4  | -3 |    | 4        | 13 | -4  | -3 |    |

Tabla 3.3: UVa 108 - Maximum Sum

Abordar este problema de forma ingenua, mediante una búsqueda completa, como se muestra a continuación, no funciona, ya que se ejecuta en  $O(n^6)$ . Un algoritmo así es demasiado lento para el caso de prueba más grande, con  $n = 100$ .

```

1 maxSubRect = -127*100*100; // el menor valor posible para este problema
2 for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // coord. inicial
3 for (int k = i; k < n; k++) for (int l = j; l < n; l++) { // coord. final
4 subRect = 0; // sumar los elementos de este subrectángulo
5 for (int a = i; a <= k; a++) for (int b = j; b <= l; b++)
6 subRect += A[a][b];
7 maxSubRect = max(maxSubRect, subRect); } // aquí está la respuesta

```

La solución de la suma de rango unidimensional máxima de la subsección anterior, se puede extender a dos (o más) dimensiones, siempre que el principio de inclusión-exclusión se aplique correctamente. La única diferencia es que, mientras que en la versión unidimensional tratamos con subrangos superpuestos, en la bidimensional lo hacemos con submatrices superpuestas. Podemos convertir la matriz de entrada  $n \times n$  en una *matriz de suma acumulada*  $n \times n$ , donde  $A[i][j]$  ya no contiene su propio valor, sino la suma de todos los elementos dentro de la submatriz  $(0, 0)$  a  $(i, j)$ . Esto se puede hacer de forma simultánea a la lectura de la entrada, y se sigue ejecutando en  $O(n^2)$ . El siguiente código convierte la matriz cuadrada de entrada (ver la tabla 3.3.A) en una matriz de suma acumulada (ver la tabla 3.3.B).

```

1 scanf("%d", &n); // la dimensión de la matriz cuadrada de entrada
2 for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
3 scanf("%d", &A[i][j]);
4 if (i > 0) A[i][j] += A[i-1][j]; // si es posible, añadir desde arriba
5 if (j > 0) A[i][j] += A[i][j-1]; // si es posible, añadir desde la izda.
6 if (i > 0 && j > 0) A[i][j] -= A[i-1][j-1]; // evitar contar dos veces
7 } // principio de inclusión-exclusión

```

Con la matriz de suma, podemos devolver la suma de cualquier submatriz  $(i, j)$  a  $(k, l)$  en  $O(1)$ , utilizando el código que se reproduce a continuación. Por ejemplo, vamos a calcular la suma de  $(1, 2)$  a  $(3, 3)$ . Dividimos la suma en 4 partes y calculamos  $A[3][3] - A[0][3] - A[3][1] + A[0][1] = -3 - 13 - (-9) + (-2) = -9$ , como está resaltado en la tabla 3.3.C. Con esta formulación de DP en  $O(1)$ , el problema de la suma de rangos bidimensional máxima se puede resolver en  $O(n^4)$ . Para el caso de prueba más grande del problema UVa 108, con  $n = 100$ , es suficientemente rápido.

```

1 maxSubRect = -127*100*100; // el valor mínimo posible en este problema
2 for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) // coord. de inicio
3 for (int k = i; k < n; k++) for (int l = j; l < n; l++) { // coord. final
4 subRect = A[k][l]; // suma de los elementos de (0, 0) a (k, l): O(1)
5 if (i > 0) subRect -= A[i-1][l]; // O(1)
6 if (j > 0) subRect -= A[k][j-1]; // O(1)
7 if (i > 0 && j > 0) subRect += A[i-1][j-1]; // O(1)
8 maxSubRect = max(maxSubRect, subRect); } // aquí está la respuesta

```



ch3\_05\_UVa108.cpp



ch3\_05\_UVa108.java

A partir de estos dos ejemplos, los problemas de sumas de rangos unidimensionales y bidimensionales máximas, podemos ver que no todos los problemas de rangos requieren un árbol de segmentos o un árbol Fenwick, tratados en las secciones 2.4.3 y 2.4.4. Los problemas de rangos de entrada estática se pueden resolver muchas veces con técnicas de DP. También conviene mencionar que solucionar los problemas de rangos con técnicas de DP de abajo a arriba resulta totalmente natural, pues el operador ya es un *array* unidimensional o bidimensional. Podríamos escribir una solución recursiva de arriba a abajo, pero no sería tan evidente.

### 3. Subsecuencia creciente máxima (LIS)

Dada una secuencia  $\{A[0], A[1], \dots, A[n-1]\}$ , determinar su subsecuencia creciente máxima (LIS)<sup>13</sup>. Estas ‘subsecuencias’ no son, necesariamente, contiguas. Ejemplo:  $n = 8$ ,  $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$ . La LIS de longitud 4 es  $\{-7, 2, 3, 8\}$ .

Como se ha mencionado en la sección 3.1, una búsqueda completa ingenua que enumere todas las subsecuencias posibles, para encontrar la creciente máxima, es demasiado lento, ya que hay

<sup>13</sup>Hay otras variantes de este problema, incluyendo la subsecuencia *decreciente* máxima y la subsecuencia *no creciente/decreciente* máxima. Las subsecuencias crecientes se pueden modelar como grafos acíclicos dirigidos (DAG), y encontrar la LIS es equivalente a encontrar los caminos más largos del DAG (ver la sección 4.7.1).

| Índice | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 |
|--------|----|----|---|---|---|---|---|---|
| A      | -7 | 10 | 9 | 2 | 3 | 8 | 8 | 1 |
| LIS(i) | 1  | 2  | 2 | 2 | 3 | 4 | 4 | 2 |

Figura 3.9: Subsecuencia creciente máxima

$O(2^n)$  posibilidades. En vez de probar todas las subsecuencias posibles, vamos a considerar el problema desde otra perspectiva. Podemos escribir el estado de este problema con un único parámetro:  $i$ . Así,  $\text{LIS}(i)$  será el final de la LIS en el índice  $i$ . Sabemos que  $\text{LIS}(0) = 1$ , ya que el primer número en  $A$  es, en sí mismo, una subsecuencia. Para  $i \geq 1$ ,  $\text{LIS}(i)$  es un poco más complejo. Necesitamos encontrar el índice  $j$ , de forma que  $j < i$  y  $A[j] < A[i]$ , y  $\text{LIS}(j)$  es la máxima. Una vez que tenemos este índice  $j$ , sabemos que  $\text{LIS}(i) = \text{LIS}(j)+1$ . Podemos escribir formalmente esta recurrencia como:

1.  $\text{LIS}(0) = 1$  // el caso base
2.  $\text{LIS}(i) = \max(\text{LIS}(j)+1), \forall j \in [0..i-1] \text{ y } A[j] < A[i]$  // el caso recursivo, uno más que la mejor solución anterior que termina en  $j$  para todos  $j < i$ .

La respuesta es el valor mayor de  $\text{LIS}(k)$ ,  $\forall k \in [0..n-1]$ .

Vamos a ver ahora cómo funciona este algoritmo (ver también la figura 3.9):

- $\text{LIS}(0)$  es 1, el primer número en  $A = \{-7\}$ , el caso base.
- $\text{LIS}(1)$  es 2, porque podemos extender  $\text{LIS}(0) = \{-7\}$  con  $\{10\}$  para formar  $\{-7, 10\}$ , de longitud 2. La mejor  $j$  para  $i = 1$  es  $j = 0$ .
- $\text{LIS}(2)$  es 2, porque podemos extender  $\text{LIS}(0) = \{-7\}$  con  $\{9\}$  para formar  $\{-7, 9\}$  de longitud 2. No podemos extender  $\text{LIS}(1) = \{-7, 10\}$  con  $\{9\}$ , ya que no es creciente. La mejor  $j$  para  $i = 2$  es  $j = 0$ .
- $\text{LIS}(3)$  es 2, porque podemos extender  $\text{LIS}(0) = \{-7\}$  con  $\{2\}$  para formar  $\{-7, 2\}$  de longitud 2. No podemos extender  $\text{LIS}(1) = \{-7, 10\}$  con  $\{2\}$ , ya que no es creciente. Tampoco podemos extender  $\text{LIS}(2) = \{-7, 9\}$  con  $\{2\}$ , ya que tampoco es creciente. La mejor  $j$  para  $i = 3$  es  $j = 0$ .
- $\text{LIS}(4)$  es 3, porque podemos extender  $\text{LIS}(3) = \{-7, 2\}$  con  $\{3\}$  para formar  $\{-7, 2, 3\}$ . Ésta es la mejor de entre todas las posibilidades. La mejor  $j$  para  $i = 4$  es  $j = 3$ .
- $\text{LIS}(5)$  es 4, porque podemos extender  $\text{LIS}(4) = \{-7, 2, 3\}$  con  $\{8\}$  para formar  $\{-7, 2, 3, 8\}$ . Ésta es la mejor de entre todas las posibilidades. La mejor  $j$  para  $i = 5$  es  $j = 4$ .
- $\text{LIS}(6)$  es 4, porque podemos extender  $\text{LIS}(4) = \{-7, 2, 3\}$  con  $\{8\}$  para formar  $\{-7, 2, 3, 8\}$ . Ésta es la mejor de entre todas las posibilidades. La mejor  $j$  para  $i = 6$  es  $j = 4$ .
- $\text{LIS}(7)$  es 2, porque podemos extender  $\text{LIS}(0) = \{-7\}$  con  $\{1\}$  para formar  $\{-7, 1\}$ . Ésta es la mejor de entre todas las posibilidades. La mejor  $j$  para  $i = 7$  es  $j = 0$ .
- Las respuestas residen en  $\text{LIS}(5)$  o  $\text{LIS}(6)$ ; ambos valores (longitudes de la LIS) son 4. Hay que destacar que el índice  $k$ , donde  $\text{LIS}(k)$  es el más alto, puede estar en cualquier punto de  $[0..n-1]$ .

Es evidente que hay muchos subproblemas superpuestos en el problema de la LIS, porque para calcular  $LIS(i)$ , necesitamos calcular  $LIS(j) \forall j \in [0..i-1]$ . Sin embargo, solo hay  $n$  estados distintos, los índices de la LIS que terminan en el índice  $i$ ,  $\forall i \in [0..n-1]$ . Como necesitamos calcular cada estado con un bucle  $O(n)$ , este algoritmo de DP se ejecuta en  $O(n^2)$ .

Si es necesario, las soluciones de la LIS se pueden reconstruir almacenando la información del predecesor (las flechas en la figura 3.9) y siguiendo las flechas desde el índice  $k$  que contiene el valor más alto de  $LIS(k)$ . Por ejemplo,  $LIS(5)$  es el estado final óptimo. Comprueba la figura 3.9. Podemos seguir las flechas de la siguiente manera:  $LIS(5) \rightarrow LIS(4) \rightarrow LIS(3) \rightarrow LIS(0)$ , de forma que la solución óptima (leer hacia atrás) es el índice  $\{0, 3, 4, 5\}$  o  $\{-7, 2, 3, 8\}$ .

El problema de la LIS se puede resolver también utilizando el algoritmo *sensible a la salida voraz* y D&C, en  $O(n \log k)$  (donde  $k$  es la longitud de la LIS), en vez de  $O(n^2)$ , manteniendo un *array* que esté *siempre ordenado* y, por tanto, sea susceptible de una búsqueda binaria. Digamos que  $L$  es un *array* en el que  $L(i)$  representa el valor final más pequeño de todas las LIS de longitud  $i$  encontradas hasta el momento. Aunque esta definición es un poco complicada, es fácil ver que, si siempre está ordenado,  $L(i-1)$  será siempre más pequeño que  $L(i)$ , ya que, por definición, el penúltimo elemento de cualquier LIS (de longitud  $i$ ) es siempre más pequeño que el último. Así, podemos hacer una búsqueda binaria en el *array*  $L$  en  $O(\log k)$ , para determinar la subsecuencia más larga posible que podemos crear añadiendo el elemento actual  $A[i]$ , simplemente encontrando el índice del último elemento de  $L$  que es más pequeño que  $A[i]$ . Usando el mismo ejemplo, actualizaremos el *array*  $L$ , paso a paso, utilizando este algoritmo:

- Inicialmente, en  $A[0] = -7$ , tenemos  $L = \{-7\}$ .
- Podemos insertar  $A[1] = 10$  en  $L[1]$  para tener una LIS de longitud 2,  $L = \{-7, \underline{10}\}$ .
- Para  $A[2] = 9$ , sustituimos  $L[1]$ , de forma que tengamos una terminación de longitud 2 ‘mejor’:  $L = \{-7, \underline{9}\}$ . Ésta es una estrategia *voraz*. Al almacenar la LIS con un valor final más pequeño, maximizamos nuestra capacidad de extenderla con valores futuros.
- Para  $A[3] = 2$ , sustituimos  $L[1]$ , para tener una terminación de longitud 2 ‘todavía mejor’:  $L = \{-7, \underline{2}\}$ .
- Insertamos  $A[4] = 3$  en  $L[2]$  para tener una LIS más larga,  $L = \{-7, \underline{2}, \underline{3}\}$ .
- Insertamos  $A[5] = 8$  en  $L[3]$  para tener una LIS más larga,  $L = \{-7, \underline{2}, \underline{3}, \underline{8}\}$ .
- Para  $A[6] = 8$ , no cambia nada, ya que  $L[3] = 8$ .  $L = \{-7, \underline{2}, \underline{3}, \underline{8}\}$  sigue igual.
- Para  $A[7] = 1$ , mejoramos  $L[1]$ , para que  $L = \{-7, \underline{1}, \underline{3}, \underline{8}\}$ . Esto ilustra cómo el *array*  $L$  *no es* la LIS de  $A$ . Este paso es importante, ya que puede haber subsecuencias más largas *en el futuro*, que pueden extender la subsecuencia de longitud 2 en  $L[1] = 1$ . Por ejemplo, veamos este caso de prueba:  $A = \{-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4\}$ . La longitud de la LIS en este caso es 5.
- La respuesta es la longitud más larga del *array* ordenado  $L$  al final del proceso.



ch3\_06\_LIS.cpp



ch3\_06\_LIS.java

#### 4. Mochila 0-1 (suma de subconjuntos)

Problema<sup>14</sup>: dados  $n$  elementos, cada uno con su propio valor  $V_i$  y peso  $W_i$ ,  $\forall i \in [0..n-1]$ , y un tamaño máximo de la mochila  $S$ , calcular el valor máximo de los elementos que podemos cargar, si podemos ignorar o elegir<sup>15</sup> un objeto en particular (de ahí el término 0-1, para ignorar o elegir).

Ejemplo:  $n = 4$ ,  $V = \{100, 70, 50, 10\}$ ,  $W = \{10, 4, 6, 12\}$ ,  $S = 12$ .

- Eligiendo el elemento 0 de peso 10 y valor 100, no podemos llevar otro. No es óptimo.
- Eligiendo el elemento 3 de peso 12 y valor 10, no podemos llevar otro. No es óptimo.
- Eligiendo los elementos 1 y 2 tenemos totales de peso 10 y valor 120. Es el máximo.

Solución: utilizar estas recurrencias de búsqueda completa  $\text{val}(\text{id}, \text{remW})$ , donde  $\text{id}$  es el índice del elemento actual a considerar y  $\text{remW}$  es el peso restante que podemos cargar en la mochila:

1.  $\text{val}(\text{id}, 0) = 0$  // si  $\text{remW} = 0$ , no podemos llevar nada más
2.  $\text{val}(n, \text{remW}) = 0$  // si  $\text{id} = n$ , hemos considerado todos los elementos
3. si  $W[\text{id}] > \text{remW}$ , habrá que ignorar este elemento  $\text{val}(\text{id}, \text{remW}) = \text{val}(\text{id}+1, \text{remW})$
4. si  $W[\text{id}] \leq \text{remW}$ , tenemos dos opciones: ignorar o llevar este objeto; tomamos el máximo  
 $\text{val}(\text{id}, \text{remW}) = \max(\text{val}(\text{id}+1, \text{remW}), V[\text{id}] + \text{val}(\text{id}+1, \text{remW}-W[\text{id}]))$

La respuesta se encuentra llamando a  $\text{value}(0, S)$ . Hay que tener en cuenta los subproblemas superpuestos en este problema de la mochila 0-1. Por ejemplo: después de tomar el elemento 0, e ignorar los elementos 1-2, llegamos al estado  $(3, 2)$ , el tercer objeto ( $\text{id} = 3$ ) con dos unidades de peso restantes ( $\text{remW} = 2$ ). Después de ignorar el elemento 0 y tomar los elementos 1-2, también llegamos al mismo estado  $(3, 2)$ . Aunque hay subproblemas superpuestos, solo tenemos  $O(nS)$  estados distintos posibles (porque  $\text{id}$  puede variar entre  $[0..n-1]$  y  $\text{remW}$  puede hacerlo entre  $[0..S]$ ). Podemos calcular cada uno de esos estados en  $O(1)$ , de forma que la complejidad de tiempo total<sup>16</sup> de esta solución de DP es  $O(nS)$ .

Como comentario, indicar que la versión de arriba a abajo de esta solución de DP es, normalmente, más rápida que la versión de abajo a arriba. Esto se debe a que, en realidad, no se visitan todos los estados y, por ello, los estados de DP requeridos son solo un subconjunto (muy pequeño) del espacio completo. Recuerda que la DP de arriba a abajo solo visita *los estados requeridos*, mientras que la de abajo a arriba visita *todos los estados distintos*. En nuestra biblioteca de código fuente incluimos ambas versiones.



ch3\_07\_UVa10130.cpp



ch3\_07\_UVa10130.java

<sup>14</sup>Este problema también es conocido como el de suma de subconjuntos. Tienen un enunciado similar: dado un conjunto de enteros y un entero  $S$ , ¿hay un subconjunto (no vacío) cuya suma sea igual a  $S$ ?

<sup>15</sup>Hay otras variantes de este problema, como el problema de la mochila continua, que tiene una solución voraz.

<sup>16</sup>Si  $S$  es tan grande que  $nS >> 1M$ , esta solución de DP no es viable, ni con el truco de ahorro de espacio.

## 5. Cambio de monedas (CC) - la versión general

Problema: dada una cantidad de céntimos objetivo  $V$  y una lista de denominaciones de  $n$  monedas, es decir, tenemos  $\text{coinValue}[i]$  (en céntimos) para los tipos de monedas  $i \in [0..n-1]$ , ¿cuál es la cantidad mínima de monedas que debemos usar para representar  $V$ ? Asumimos que tenemos un suministro ilimitado de monedas de cualquier tipo (ver también la sección 3.4.1).

Ejemplo 1:  $V = 10$ ,  $n = 2$ ,  $\text{coinValue} = \{1, 5\}$ ; podemos usar:

- A. Diez monedas de 1 céntimo  $= 10 \times 1 = 10$ ; monedas usadas = 10
- B. Una moneda de 5 céntimos + cinco de 1 céntimo  $= 1 \times 5 + 5 \times 1 = 10$ ; monedas = 6
- C. Dos monedas de 5 céntimos  $= 2 \times 5 = 10$ ; monedas usadas = 2  $\rightarrow$  óptimo

Podemos usar un algoritmo voraz si las denominaciones de las monedas son adecuadas (ver la sección 3.4.1). De echo, el ejemplo anterior se puede resolver mediante un algoritmo voraz. Sin embargo, para el caso general, tenemos que utilizar DP. Ver el siguiente ejemplo:

Ejemplo 2:  $V = 7$ ,  $n = 4$ ,  $\text{coinValue} = \{1, 3, 4, 5\}$ . La solución voraz dará como resultado 3 monedas combinadas de forma  $5+1+1 = 7$ , pero la óptima es 2 monedas (en concreto 4+3).

Solución: utilizar estas relaciones de recurrencia de búsqueda completa para `change(value)`, donde `value` es la cantidad restante de céntimos que debemos representar en monedas:

1. `change(0) = 0 // necesitamos 0 monedas para 0 céntimos`
2. `change(< 0) = infinity // en la práctica nos sirve un valor positivo grande`
3. `change(value) = min + (1 + change(value-coinValue[i]))`  $\forall i \in [0..n-1]$

La respuesta está en el valor devuelto por `change(V)`.

|          |   |   |   |   |   |   |   |   |   |   |    |
|----------|---|---|---|---|---|---|---|---|---|---|----|
| <0       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $\infty$ | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2  |

$V=10, n=2, \text{coinValue}=\{1, 5\}$

Figura 3.10: Cambio de monedas

La figura 3.10 muestra que:  $\text{change}(0) = 0$  y  $\text{change}(< 0) = \infty$ : estos son los casos base.  $\text{change}(1) = 1$ , de  $1 + \text{change}(1-1)$ , ya que  $1 + \text{change}(1-5)$  es inviable (devuelve  $\infty$ ).  $\text{change}(2) = 2$ , de  $1 + \text{change}(2-1)$ , ya que  $1 + \text{change}(2-5)$  también es inviable (devuelve  $\infty$ ). Lo mismo para  $\text{change}(3)$  y  $\text{change}(4)$ .  $\text{change}(5) = 1$ , de  $1 + \text{change}(5-5) = 1$  moneda, menor que  $1 + \text{change}(5-1) = 5$  monedas. Y así hasta  $\text{change}(10)$ . La respuesta está en `change(V)`, que es `change(10) = 2` en este ejemplo.

Podemos ver que hay muchos subproblemas superpuestos en el problema del cambio de monedas (por ejemplo, tanto `change(10)` como `change(6)` requieren el valor de `change(5)`). Sin embargo, solo hay  $O(V)$  estados distintos posibles (ya que `value` puede variar entre  $[0..V]$ ). Como necesitamos comprobar  $n$  tipos de monedas por estado, la complejidad de tiempo total de esta solución de DP es  $O(nV)$ .

Una variante de este problema consiste en contar el número de formas (canónicas) posibles de obtener el valor  $V$  céntimos, utilizando una lista de denominaciones de  $n$  monedas. En el ejemplo 1 anterior, la respuesta es 3:  $\{1+1+1+1+1 + 1+1+1+1+1, 5 + 1+1+1+1+1, 5 + 5\}$ .

Solución: utiliza la relación de recurrencias de búsqueda completa `ways(type, value)`, donde `value` es igual que en el caso anterior, pero ahora tenemos un parámetro `type` adicional, para el índice del tipo de moneda que estamos considerando en cada momento. Este segundo parámetro `type` es importante, ya que esta solución considera los tipos de monedas de forma secuencial. Una vez que hayamos decidido ignorar cierta denominación de moneda, no debemos volver a considerarla, para evitar contarla dos veces:

1. `ways(type, 0) = 1 // una manera, no usar nada`
2. `ways(type, <0) = 0 // ninguna manera, no podemos llegar al valor negativo`
3. `ways(n, value) = 0 // ninguna manera, tras considerar los tipos de monedas ∈ [0..n-1]`
4. `ways(type, value) = ways(type+1, value) + // si ignoramos este tipo de moneda,  
ways(type, value-coinValue[type]) // y además usamos este tipo`

Solo hay  $O(nV)$  estados distintos posibles. Como cada estado se puede calcular en  $O(1)$ , la complejidad de tiempo total<sup>17</sup> de esta solución de DP es  $O(nV)$ . La respuesta se encuentra llamando a `ways(0, V)`. Nota: si las denominaciones de las monedas no cambian y hay muchas consultas con diferentes  $V$ , podemos *no* reiniciar la tabla de memoria. Por lo tanto, el algoritmo se ejecutaría en  $O(nV)$  la primera vez y bastaría con consultas  $O(1)$  las siguientes.



`ch3_08_UVa674.cpp`



`ch3_08_UVa674.java`

## 6. El problema del viajante (TSP)

Problema: dadas  $n$  ciudades y las distancias que separan a cada pareja de ellas, en forma de una matriz simétrica `dist` de tamaño  $n \times n$ , calcular el coste mínimo de realizar una ruta<sup>18</sup> que comience en cualquier ciudad  $s$ , visite las otras  $n - 1$  ciudades *exactamente una vez* y, finalmente, vuelva a la ciudad inicial  $s$ .

Ejemplo: el grafo mostrado en la figura 3.11 tiene  $n = 4$  ciudades. Por lo tanto, tenemos  $4! = 24$  posibles rutas (permutaciones de 4 ciudades). Una de las rutas mínimas es A-B-C-D-A, con un coste de  $20 + 30 + 12 + 35 = 97$  (pero puede haber más de una solución óptima).

Una solución de ‘fuerza bruta’ al TSP (ya sea iterativa o recursiva) que pruebe todas las  $O((n - 1)!)$  rutas posibles (fijando la ciudad inicial en el vértice A, para aprovechar la simetría) solo resulta efectiva cuando  $n$  es un máximo de 12, ya que  $11! \approx 40M$ . Cuando  $n > 12$ , esas soluciones de fuerza bruta obtendrían un veredicto TLE en los concursos de programación. Sin embargo, si existen casos de prueba múltiples, el límite para la mencionada solución será, probablemente, de solo  $n = 11$ .

<sup>17</sup>Si  $V$  es tan grande que  $nV >> 1M$ , esta solución de DP no es viable ni con el truco de ahorro de espacio.

<sup>18</sup>Esa ruta se llama camino hamiltoniano y consiste en un ciclo en un grafo no dirigido, que visita cada vértice exactamente una vez y vuelve al vértice inicial.

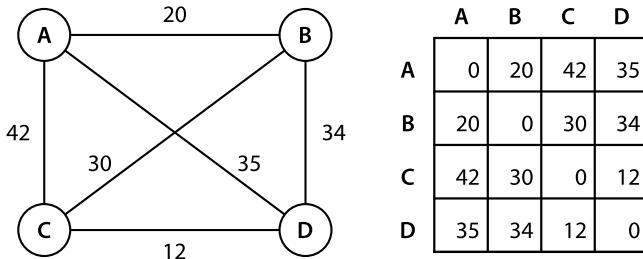


Figura 3.11: Un grafo completo

Podemos utilizar DP para el TSP, ya que es obvio que el cálculo de las subrutas se superpone, por ejemplo, la ruta  $A - B - C - (n - 3)$  ciudades distintas y vuelve finalmente a  $A$ , está evidentemente superpuesta a la ruta  $A - C - B -$  igualmente  $(n - 3)$  ciudades distintas volviendo a  $A$ . Si podemos evitar repetir los cálculos de las longitudes de estas subrutas, podemos ahorrar mucho tiempo. Sin embargo, cada estado distinto en el TSP depende de dos parámetros: la última ciudad/vértice visitada  $\text{pos}$  y algo que no habíamos visto antes, un *subconjunto* de ciudades visitadas.

Hay muchas formas de representar un conjunto. Sin embargo, ya que vamos pasar esta información del conjunto como parámetro de una función recursiva (si utilizamos DP de arriba a abajo), la representación que utilicemos debe ser ligera y eficiente. En la sección 2.2, hemos visto una opción viable para ello: la *máscara de bits*. Si tenemos  $n$  ciudades, usamos un entero binario de longitud  $n$ . Si el bit  $i$  es ‘1’ (activado), decimos que este elemento (ciudad)  $i$  está dentro del conjunto (ha sido visitado) y el elemento  $i$  no está dentro del conjunto (no ha sido visitado) si, en su lugar, el bit es ‘0’ (desactivado). Por ejemplo,  $\text{mask} = 18_{10} = 10010_2$  implica que los elementos (ciudades) {1, 4} están en el conjunto<sup>19</sup> (y han sido visitados). No olvides que, para comprobar si el bit  $i$  de un entero  $\text{mask}$  está activado o desactivado, podemos utilizar  $\text{mask} \& (1 << i)$ . Para establecer el bit  $i$  de un entero  $\text{mask}$ , podemos usar  $\text{mask} |= (1 << i)$ .

Solución: utilizar estas relaciones de recurrencia de búsqueda completa para `tsp(pos, mask)`:

1. `tsp(pos, 2n - 1) = dist[pos][0] // todas las ciudades visitadas, se vuelve a la inicial.`  
Nota:  $\text{mask} = (1 << n) - 1$  o  $2^n - 1$  implica que los  $n$  bits de  $\text{mask}$  están activados.
2. `tsp(pos, mask) = min(dist[pos][nxt] + tsp(nxt, mask | (1 << nxt))) // ∀ nxt ∈ [0..n-1], nxt != pos y (mask & (1 << nxt)) es '0' (desactivado).` Básicamente intentamos, a cada paso, todas las siguientes ciudades posibles que no hayan sido visitadas.

Solo hay  $O(n \times 2^n)$  estados distintos, porque hay  $n$  ciudades y tenemos que recordar hasta  $2^n$  ciudades más que han sido visitadas en cada ruta. Cada estado se puede calcular en  $O(n)$ , por lo que la complejidad de tiempo total de esta solución de DP es  $O(2^n \times n^2)$ . Esto nos permite resolver hasta<sup>20</sup>  $n \approx 16$  ya que  $16^2 \times 2^{16} \approx 17M$ . No suponen una gran mejora sobre la solución de fuerza bruta, pero si el problema TSP que estamos resolviendo tiene un tamaño de entrada  $11 \leq n \leq 16$ , la opción evidente es la DP. La respuesta se encuentra llamando a `tsp(0, 1)`:

<sup>19</sup>Recuerda que en  $\text{mask}$  los índices empiezan en 0 y se cuentan desde la derecha.

<sup>20</sup>Como los problemas de concursos de programación requieren, normalmente, soluciones exactas, la solución al TSP con DP que hemos incluido es ya una de las mejores. En la vida real, el TSP se debe resolver, a menudo, con miles de ciudades. Para resolver problemas grandes como este, tenemos métodos no exactos como los que aparecen en [26].

empezamos desde la ciudad 0 (podemos hacerlo desde cualquier vértice, pero la elección más simple es el 0) y establecemos `mask = 1`, para asegurarnos de no volver a visitar esa ciudad 0.

Normalmente, en los concursos de programación, para resolver los problemas TSP con DP es necesario realizar algún tipo de procesamiento previo del grafo, que genere la matriz de distancias `dist`, antes de ejecutar la solución de DP. Vemos estas variantes en la sección 8.4.3.

Las soluciones de DP que implican un conjunto (pequeño) de booleanos, como uno de los parámetros, son generalmente conocidas como técnicas de DP con máscaras de bits. En las secciones 8.3 y 9.3, encontramos problemas de DP más complejos, que requieren de esta técnica.



[visualgo.net/rectree](http://visualgo.net/rectree)



ch3\_09\_UVa10496.cpp



ch3\_09\_UVa10496.java

### Ejercicio 3.5.2.1

La solución al problema de suma de rangos bidimensionales máxima se ejecuta en  $O(n^4)$ . En realidad, existe una solución  $O(n^3)$  que combina la solución de DP para el problema de la suma de rangos unidimensional máxima, para una dimensión, y la idea propuesta por Kadane para la otra. Resuelve el problema UVa 108 con una solución  $O(n^3)$ .

### Ejercicio 3.5.2.2

La solución para la consulta del rango mínimo( $i, j$ ) en arrays unidimensionales de la sección 2.4.3, utiliza un árbol de segmentos. Resulta excesivo si el array dado es estático y no cambia durante las consultas. Usa una técnica de DP para responder a  $\text{RMQ}(i, j)$  en  $O(n \log n)$ , con procesamiento previo y una complejidad de  $O(1)$  por consulta.

### Ejercicio 3.5.2.4

¿Es posible utilizar una técnica de búsqueda completa iterativa, que intente todos los subconjuntos posibles de  $n$  elementos, como la que hemos tratado en la sección 3.2.1, para resolver el problema de la mochila 0-1? ¿Dónde están las limitaciones, si las hay?

### Ejercicio 3.5.2.5\*

Vamos a suponer que añadimos un parámetro más al problema clásico de la mochila 0-1. Hagamos que  $K_i$  indique el número de copias del elemento  $i$ , para utilizarlo en el problema. Por ejemplo,  $n = 2$ ,  $V = \{100, 70\}$ ,  $W = \{5, 4\}$ ,  $K = \{2, 3\}$ ,  $S = 17$ , significa que hay dos copias del elemento 0, con peso 5 y valor 100, y hay tres copias del elemento 1, con peso 4 y valor 70. La solución óptima de este ejemplo es utilizar uno de los elementos 0 y tres elementos 1, con un peso total de 17 y un valor total de 310. Resuelve esta variante del problema de la mochila 0-1, asumiendo que  $1 \leq n \leq 500$ ,  $1 \leq S \leq 2000$ ,  $n \leq \sum_{i=0}^{n-1} K_i \leq 40000$ . Consejo: todo entero se puede escribir como la suma de potencias de 2.

### Ejercicio 3.5.2.6\*

La solución al TSP con DP, que aparece en esta sección, se puede mejorar *un poco*, para que sea capaz de resolver el caso de prueba con  $n = 17$  en un concurso. Describe el pequeño cambio necesario para hacerlo posible. Consejo: ten en cuenta la simetría.

### Ejercicio 3.5.2.7\*

Además del cambio menor que se pide en el **ejercicio 3.5.2.6\***, ¿qué otros cambios son necesarios para tener una solución de DP al TSP que sea capaz de resolver  $n = 18$  (o, incluso,  $n = 19$ , pero con muchos menos casos de prueba)?

## 3.5.3 Ejemplos no clásicos

Aunque los problemas de DP son un tipo muy común, y con una frecuencia de aparición alta en concursos de programación recientes, los problemas de DP clásicos, en sus *formas puras*, ya no aparecen en los ICPC o IOI modernos. Los hemos estudiado para entender la DP, pero debemos aprender a resolver muchos otros problemas de DP no clásicos (que, quizás, pasen a ser clásicos en un futuro cercano) y, así, desarrollar nuestras ‘capacidades de DP’ durante el proceso. En esta subsección, tratamos otros dos ejemplos no clásicos, añadidos al problema UVa 11450 - Wedding Shopping que hemos visto antes en detalle. También hemos seleccionado algunos problemas de DP no clásicos fáciles, como ejercicios de programación. Una vez que hayas resuelto la mayoría de estos problemas, quedas invitado a conocer los más complicados en otras partes de este libro, como los de las secciones 4.7.1, 5.4, 5.6, 6.5, 8.3, 9.2, 9.21, etc.

### 1. UVa 10943 - How do you add?

Enunciado resumido del problema: dado un entero  $n$ , ¿de cuántas formas se pueden sumar  $K$  enteros no negativos, menores o iguales a  $n$ , para llegar a  $n$ ? Límites:  $1 \leq n, K \leq 100$ . Por ejemplo: para  $n = 20$  y  $K = 2$ , hay 21 formas:  $0 + 20$ ,  $1 + 19$ ,  $2 + 18$ ,  $3 + 17$ , ...,  $20 + 0$ .

En términos matemáticos, el número de formas se puede expresar como  $\binom{n+k-1}{k-1}$  (ver la sección 5.4.2, sobre coeficientes binomiales, que también necesitan de DP). Utilizaremos este sencillo problema para volver a ilustrar los principios de la programación dinámica que hemos visto en esta sección, especialmente el proceso de deducir los estados oportunos de un problema y las correspondientes transiciones de un estado a otro, dados los casos base.

En primer lugar, debemos determinar los parámetros de este problema, que representarán los distintos estados. En este caso, solo hay dos parámetros,  $n$  y  $K$ . Por lo tanto, solo hay 4 combinaciones posibles:

1. Si no elegimos ninguno de ellos, no podemos representar un estado. Esta opción se ignora.
2. Si solo elegimos  $n$ , no podremos saber cuántos números  $\leq n$  se han usado.
3. Si solo elegimos  $K$ , no podremos saber el  $n$  objetivo de la suma.
4. Por lo tanto, el estado de este problema se representa por una pareja (o 2-tupla)  $(n, K)$ . El orden de los parámetros es irrelevante, es decir, la pareja  $(K, n)$  también es válida.

A continuación, debemos determinar los casos base. Resulta que este problema es muy sencillo cuando  $K = 1$ . Independientemente de  $n$ , sola hay *una forma* de sumar exactamente un número menor o igual a  $n$  para obtener  $n$ : el propio  $n$ . No hay ningún otro caso base para este problema.

Para el caso general, tenemos esta formulación recursiva que no es muy complicada de deducir: en el estado  $(n, K)$ , donde  $K > 1$ , podemos dividir  $n$  en un número  $X \in [0..n]$  y  $n - X$ , así,  $n = X + (n - X)$ . Al hacerlo, llegamos al subproblema  $(n - X, K - 1)$ , es decir, dado un número  $n - X$ , ¿de cuántas formas se pueden sumar  $K - 1$  números menores o iguales a  $n - X$ , para llegar a  $n - X$ ? Después, podemos sumar todas esas formas.

Estas ideas se pueden expresar como una recurrencia de búsqueda completa `ways(n, K)`:

1. `ways(n, 1) = 1 // solo podemos usar 1 número para llegar a n, el propio n`
2. `ways(n, K) =  $\sum_{X=0}^n$  ways(n-X, K-1) // sumar todas las formas posibles, recursivamente`

Este problema tiene subproblemas superpuestos. Por ejemplo, en el caso de prueba  $n = 1, K = 3$ , tiene los siguientes: se llega al estado  $(n = 0, K = 1)$  dos veces (ver la figura 4.39 en la sección 4.7.1). Sin embargo, solo hay  $n \times K$  estados posibles de  $(n, K)$ . El coste de calcular cada estado es de  $O(n)$ . Por lo tanto, la complejidad de tiempo total es  $O(n^2 \times K)$ . Como  $1 \leq n, K \leq 100$ , es una técnica viable. La respuesta se encuentra llamando a `ways(n, K)`.

Hay que tener en cuenta que este problema, en realidad, solo necesita el resultado módulo  $1M$  (es decir, los 6 últimos dígitos de la respuesta). En la sección 5.5.8 se trata el cálculo con aritmética modular.



`ch3_10_UVa10943.cpp`



`ch3_10_UVa10943.java`

## 2. UVa 10003 - Cutting Sticks

Enunciado resumido del problema: tenemos una vara de longitud  $1 \leq l \leq 1000$  a la que se le realizan hasta  $1 \leq n \leq 50$  cortes (se dan las coordenadas de los cortes, en el rango  $[0..l]$ ). El

coste de un corte viene determinado por la longitud de la vara. La tarea consiste en encontrar una secuencia de cortes de forma que el coste total sea mínimo.

Ejemplo:  $l = 100$ ,  $n = 3$ , y las coordenadas de los cortes:  $A = \{25, 50, 75\}$  (ya ordenadas)

Si cortamos de izquierda a derecha, el coste será de 225:

1. El primer corte es en la coordenada 25, coste total hasta el momento = 100.
2. El segundo corte es en la coordenada 50, coste total hasta el momento =  $100 + 75 = 175$ .
3. El tercer corte es en la coordenada 75, coste total final =  $175 + 50 = 225$ .

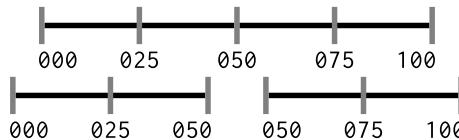


Figura 3.12: Ilustración de Cutting Sticks

Sin embargo, la respuesta óptima es 200:

1. El primer corte es en la coordenada 50, coste total hasta el momento = 100; (este corte aparece en la figura 3.12).
2. El segundo corte es en la coordenada 25, coste total hasta el momento =  $100 + 50 = 150$ .
3. El tercer corte es en la coordenada 75, coste total final =  $150 + 50 = 200$ .

¿Cómo abordamos este problema? Una aproximación inicial puede consistir en el siguiente algoritmo de búsqueda completa: probar todos los posibles puntos de corte. Antes de eso, debemos seleccionar una definición de estados adecuada para el problema: los trozos (intermedios). Podemos describir un trozo de la vara a partir de sus dos extremos: **izquierdo** y **derecho**. Sin embargo, estos dos valores pueden ser enormes y complicará las soluciones más tarde, cuando queramos *memoizar* sus valores. Podemos obtener ventaja del hecho de que solo hay  $n + 1$  varas más pequeñas después de cortar la vara original  $n$  veces. Los extremos de cada uno de los trozos se pueden describir como 0, las coordenadas del corte y  $l$ . Por lo tanto, añadimos dos coordenadas más, de forma que  $A = \{0, \text{el } A \text{ original y } l\}$ , para que podamos describir una vara por los índices de sus extremos en  $A$ .

Después, podremos utilizar estas recurrencias para `cut(izquierdo, derecho)`, donde **izquierdo** y **derecho** son los índices izquierdo y derecho de la vara con respecto a  $A$ . Inicialmente, la vara está descrita por **izquierdo** = 0 y **derecho** =  $n+1$ , es decir, una vara de longitud  $[0..l]$ :

1. `cut(i-1, i) = 0,  $\forall i \in [1..n+1]$`  // si  $izquierdo+1 = derecho$  donde **izquierdo** y **derecho** son los índices en  $A$ , tendremos un trozo de vara que ya no hay que dividir más.
2. `cut(izquierdo, derecho) = min(cut(izquierdo, i) + cut(i, derecho) + (A[derecho] - A[izquierdo]))  $\forall i \in [izquierdo+1..derecho-1]$`  // probar todos los puntos de corte posibles y elegir el mejor. El coste de un corte es la longitud del trozo actual, determinado por  $(A[derecho] - A[izquierdo])$ . La respuesta se encuentra en `cut(0, n+1)`.

Ahora, vamos a analizar la complejidad de tiempo. Inicialmente, tenemos  $n$  elecciones para los puntos de corte. Una vez que cortamos en un punto determinado, nos quedan  $n - 1$  elecciones para el segundo punto de corte. Esto se repite hasta que no nos queda ningún punto de corte. Probar todos los puntos de corte posibles de esta forma nos lleva a un algoritmo  $O(n!)$ , que es imposible para  $1 \leq n \leq 50$ .

Sin embargo, este problema tiene subproblemas superpuestos. Por ejemplo, en la figura 3.12, el corte en el índice 2 (punto de corte = 50) resulta en dos estados: (0, 2) y (2, 4). También se puede llegar al mismo estado (2, 4) al cortar en el índice 1 (punto de corte = 25) y, después, cortando en el índice 2 (punto de corte = 50). Por lo tanto, el espacio de búsqueda no es tan grande. Solo hay  $(n+2) \times (n+2)$  índices izquierdo/derecho posibles, o  $O(n^2)$  estados distintos, y se pueden almacenar en memoria. El tiempo requerido para calcular un estado es  $O(n)$ . Así, la complejidad de tiempo total (de DP de arriba a abajo) es  $O(n^3)$ . Como  $n \leq 50$ , esta solución es viable.



ch3\_11\_UVa10003.cpp



ch3\_11\_UVa10003.java

### Ejercicio 3.5.3.1\*

Casi todo el código fuente de esta sección que está disponible en la página web del libro (<http://cpbook.net>) (LIS, Coin Change, TSP y UVa 10003 - Cutting Sticks) está escrito como DP de arriba a abajo, debido a los gustos de los autores del libro. Reescríbelo utilizando la técnica de DP de abajo a arriba.

### Ejercicio 3.5.3.2\*

Resuelve el problema de Cutting Sticks en  $O(n^2)$ . Consejo: usa la aceleración de DP de Knuth-Yao, al verificar que la recurrencia satisface la desigualdad del cuadrángulo ([2]).

## Comentarios sobre programación dinámica en concursos de programación

Las técnicas *básicas* de DP (y voraces) siempre aparecen en los libros de texto de algoritmia más habituales, por ejemplo ‘Introduction to Algorithms’ [7], ‘Algorithm Design’ [38], ‘Algorithm’ [8], etc. En esta sección, hemos visto seis problemas de DP clásicos, y sus soluciones. En la tabla 3.4 aparece un breve resumen. Estos problemas de DP clásicos, en caso de que aparezcan en concursos de programación actuales, lo harán seguramente como parte de problemas más grandes y complejos.

Para ayudar a manterse al día con la dificultad creciente y creatividad requeridas para el uso de estas técnicas (especialmente la DP no clásica), recomendamos la lectura de los tutoriales de algoritmos de TopCoder [30], y tratar de resolver los problemas de concursos de programación más recientes.

|            | RSQ 1D     | RSQ 2D       | LIS          | Mochila       | CC              | TSP              |
|------------|------------|--------------|--------------|---------------|-----------------|------------------|
| Estado     | (i)        | (i, j)       | (i)          | (id, remW)    | (v)             | (pos, mask)      |
| Espacio    | $O(n)$     | $O(n^2)$     | $O(n)$       | $O(nS)$       | $O(V)$          | $O(n2^n)$        |
| Transición | subarray   | submatriz    | todo $j < i$ | tomar/ignorar | las $n$ monedas | las $n$ ciudades |
| Tiempo     | $O(n + 1)$ | $O(n^2 + 1)$ | $O(n^2)$     | $O(nS)$       | $O(nV)$         | $O(2^n n^2)$     |

Tabla 3.4: Resumen de los problemas de DP clásicos de esta sección

A lo largo de este libro, volveremos a encontrar la DP en varias ocasiones: el algoritmo de DP de Floyd Warshall (sección 4.5), DP en DAG (implícitos) (sección 4.7.1), alineación de cadenas (distancia de edición), subsecuencia común más larga (LCS), otra DP en algoritmos para cadenas (sección 6.5), más DP avanzada (sección 8.3) y muchos otros en el capítulo 9.

En el pasado (década de 1990), un concursante con buenas habilidades en DP se convertía en el ‘rey de los concursos de programación’, pues los ‘problemas decisivos’ solían ser de programación dinámica. En la actualidad, dominar la DP es un requisito *básico*. Es imposible obtener un buen resultado en un concurso de programación sin este conocimiento. Sin embargo, debemos seguir recordando al lector de este libro que no se debe afirmar que se conoce la DP solo por memorizar las soluciones de los problemas clásicos. Hay que ser un maestro en el arte de resolver problemas de DP: aprender a determinar los estados (la tabla de DP) que pueden representar de forma única y eficiente los subproblemas y, también, saber llenar esa tabla de DP, a través de iteraciones de arriba a abajo o de abajo a arriba.

No hay mejor manera de dominar estos paradigmas de resolución de problemas que resolviendo auténticos problemas de concursos de programación. A continuación, incluimos una lista de varios de ellos. Una vez que estés familiarizado con estos, puedes empezar a estudiar los nuevos problemas de DP que han comenzado a aparecer en los concursos de programación recientes.

## Ejercicios de programación

### Ejercicios de programación que se resuelven mediante programación dinámica:

#### Suma de rangos unidimensional máxima

1. UVa 00507 - Jill Rides Again (problema estándar)
2. [UVa 00787 - Maximum Sub ... \\*](#) (producto de rango máximo unidimensional; cuidado con el 0; utilizar BigInteger de Java)
3. [UVa 10684 - The Jackpot \\*](#) (estándar; algoritmo de Kadane)
4. [UVa 10755 - Garbage Heap \\*](#) (suma de rango máxima bidimensional en 2 de las 3 dimensiones; suma de rango máximo unidimensional con el algoritmo de Kadane en la otra dimensión)

Ver más ejemplos en la sección 8.4.

#### Suma de rangos bidimensional máxima

1. [UVa 00108 - Maximum Sum \\*](#) (suma de rango máxima bidimensional)
2. UVa 00836 - Largest Submatrix (convertir '0' a -INF)
3. UVa 00983 - Localized Summing for ... (suma de rango máxima bidimensional; obtener la submatriz)

- |                                                                                                                                                              |                                                                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4. UVa 10074 - Take the Land<br>5. UVa 10667 - Largest Block<br>6. <b><u>UVa 10827 - Maximum Sum on ... *</u></b><br><br>7. <b><u>UVa 11951 - Area *</u></b> | (problema estándar)<br>(problema estándar)<br>(copiar la matriz $n \times n$ en otra $n \times 2n$ ; entonces el problema vuelve a ser estándar)<br>(usar <code>long long</code> ; suma de rango máxima bidimensional; podar el espacio de búsqueda si es posible) |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Subsecuencia creciente máxima (LIS)

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00111 - History Grading<br>2. UVa 00231 - Testing the Catcher<br>3. UVa 00437 - The Tower of Babylon<br>4. <b><u>UVa 00481 - What Goes Up? *</u></b><br>5. UVa 00497 - Strategic Defense Initiative<br>6. UVa 01196 - Tiling Up Blocks<br><br>7. UVa 10131 - Is Bigger Smarter?<br>8. UVa 10534 - Wavio Sequence<br>9. <i>UVa 11368 - Nested Dolls</i><br>10. <b><u>UVa 11456 - Trainsorting *</u></b><br>11. <b><u>UVa 11790 - Murcia's Skyline *</u></b> | (cuidado con el sistema de clasificación)<br>(directo)<br>(se puede modelar como LIS)<br>(LIS en $O(n \log k)$ ; mostrar solución)<br>(se debe mostrar la solución)<br>(LA 2815 - Kaohsiung03; ordenar todos los bloques en L[i] creciente, después es el problema LIS clásico)<br>(ordenar los elefantes por IQ decreciente; LIS en peso creciente)<br>(hay que usar LIS en $O(n \log k)$ dos veces)<br>(ordenar en una dimensión; LIS en la otra)<br>( $\max(\text{LIS}(i) \text{ y } \text{LDS}(i) - 1), \forall i \in [0 \dots n-1]$ )<br>(combinación de LIS y LDS; ponderado) |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Mochila 0-1 (suma de subconjuntos)

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00562 - Dividing Coins<br>2. UVa 00990 - Diving For Gold<br>3. UVa 01213 - Sum of Different Primes<br><br>4. UVa 10130 - SuperSale<br>5. UVa 10261 - Ferry Loading<br>6. <b><u>UVa 10616 - Divisible Group Sum *</u></b><br>7. UVa 10664 - Luggage<br>8. <b><u>UVa 10819 - Trouble of 13-Dots *</u></b><br>9. <i>UVa 11003 - Boxes</i><br><br>10. UVa 11341 - Term Strategy<br><br>11. <b><u>UVa 11566 - Let's Yum Cha *</u></b><br><br>12. UVa 11658 - Best Coalition | (usar una tabla unidimensional)<br>(mostrar la solución)<br>(LA 3619 - Yokohama06; extensión de la mochila 0-1; s: (id, remN, remK) en vez de s: (id, remN))<br>(problema de la mochila 0-1 muy básico)<br>(s: (coche_actual, izquierda, derecha))<br>(la entrada puede ser negativa; usar <code>long long</code> )<br>(suma de subconjuntos)<br>(mochila 0-1 con la variante de 'tarjeta de crédito')<br>(probar todos los pesos máximos de 0 a<br>$\max(\text{weight}[i] + \text{capacity}[i]), \forall i \in [0..n-1]$ ; si conocemos un peso máximo, ¿cuántas cajas podemos apilar?)<br>(s: (id, h_aprendido, h_izquierda); t: aprendido módulo 'id' por 1 hora o ignorar)<br>(problema de lectura en inglés; en realidad una variante de la mochila: doblar cada <i>dim sum</i> y añadir otro parámetro para comprobar si hemos comprado muchos platos)<br>(s: (id, compartir); t: formar/ignorar coalición con id) |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Cambio de monedas (CC)

- |                                                        |                                                                                  |
|--------------------------------------------------------|----------------------------------------------------------------------------------|
| 1. UVa 00147 - Dollars<br>2. UVa 00166 - Making Change | (similar a UVa 357 y 674)<br>(dos variantes de cambio de monedas en un problema) |
|--------------------------------------------------------|----------------------------------------------------------------------------------|

- 3. **UVa 00357 - Let Me Count The Ways \*** (similar a UVa 147 y 674)
- 4. UVa 00674 - Coin Change (problema de cambio de monedas básico)
- 5. **UVa 10306 - e-Coins \*** (variante: cada moneda tiene dos componentes)
- 6. UVa 10313 - Pay the Price (cambio de monedas modificado y suma de rangos 1D con DP)
- 7. UVa 11137 - Ingenuous Cubrency (usar long long)
- 8. **UVa 11517 - Exact Change \*** (una variación sobre el problema de cambio de monedas)

### El problema del viajante (TSP)

- 1. **UVa 00216 - Getting in Line \*** (TSP, se puede resolver con *backtracking*)
- 2. **UVa 10496 - Collecting Beepers \*** (en realidad, como  $n \leq 11$ , este problema se puede resolver con *backtracking* recursivo y suficiente poda)
- 3. **UVa 11284 - Shopping Trip \*** (requiere procesado previo de caminos más cortos; variante del TSP donde podemos irnos antes a casa; solo hay que modificar un poco la recurrencia de DP del TSP: en cada estado tenemos otra opción, ir antes a casa)

Ver más ejemplos en las secciones 8.4.3 y 9.3.

### No clásicos (fáciles)

- 1. UVa 00116 - Unidirectional TSP (similar a UVa 10337)
- 2. *UVa 00196 - Spreadsheet* (las dependencias de las celdas son acíclicas; podemos *memoizar* el valor directo (o indirecto) de cada celda)
- 3. *UVa 01261 - String Popping* (LA 4844 - Daejeon10; problema de *backtracking* sencillo; podemos usar un `set<string>` para evitar comprobar el mismo estado (una subcadena) dos veces)  
( $s: (l, r)$ )  
(utilizar técnica de desplazamiento porque el valor puede ser negativo)  
( $s: (\text{idx}, \text{rem1}, \text{rem2})$ ; en qué sitio estamos ahora, hasta 30; las varillas restantes se comprueban en NCPC; y las restantes se comprobarán en BCEW; para cada sitio, dividimos las varillas,  $x$  se comprueban en NCPC y  $m[i] - x$  en BCEW; mostrar la solución)
- 4. UVa 10003 - Cutting Sticks
- 5. UVa 10036 - Divisibility
- 6. *UVa 10086 - Test the Rods*
- 7. **UVa 10337 - Flight Planner \*** (DP; caminos más cortos en un DAG)
- 8. UVa 10400 - Game Show Math
- 9. *UVa 10446 - The Marriage Interview* (sirve *backtracking* con una poda inteligente)
- 10. UVa 10465 - Homer Simpson
- 11. *UVa 10520 - Determine it* (editar un poco la función recursiva dada; añadir *memoización*)
- 12. *UVa 10688 - The Poor Giant* (tabla de DP unidimensional)
- 13. **UVa 10721 - Bar Codes \*** (escribir la fórmula dada como DP de arriba a abajo con *memoización*)
- 14. UVa 10910 - Mark's Distribution
- 15. UVa 10912 - Simple Minded ...
- 16. **UVa 10943 - How do you add? \*** (el ejemplo en el enunciado es incorrecto, debería ser:  
 $1+(1+3)+(1+3)+(1+3) = 1+4+4+4 = 13$ , ganando a 14; por lo demás, DP sencilla)  
( $s: (n, k)$ ; probar todos desde 1 hasta m)  
(tabla de DP bidimensional)  
( $s: (\text{long}, \text{último}, \text{suma})$ ; probar el siguiente carácter)  
( $s: (n, k)$ ; probar todos los puntos de separación posibles; una solución alternativa consiste en usar la fórmula matemática cerrada  
 $C(n + k - 1, k - 1)$ , que también necesita DP)
- 17. *UVa 10980 - Lowest Price in Town* (DP sencilla)

|                                    |                                                               |
|------------------------------------|---------------------------------------------------------------|
| 18. UVa 11026 - A Grouping Problem | (DP; idea similar al teorema de los binomios)                 |
| 19. UVa 11407 - Squares            | (se puede <i>memoizar</i> )                                   |
| 20. UVa 11420 - Chest of ...       | (s: (anterior, id, numlck); bloquear/desbloquear esta cómoda) |
| 21. UVa 11450 - Wedding Shopping   | (DP estándar)                                                 |
| 22. UVa 11703 - sqrt log sin       | (se puede <i>memoizar</i> )                                   |

### Otros problemas de DP clásica de este libro

|                                                                 |                          |
|-----------------------------------------------------------------|--------------------------|
| 1. Floyd Warshall para caminos más cortos entre todos los pares | (sección 4.5)            |
| 2. Alineamiento de cadenas (distancia de edición)               | (sección 6.5)            |
| 3. Subsecuencia común máxima                                    | (sección 6.5)            |
| 4. Multiplicación de cadenas de matrices                        | (sección 9.20)           |
| 5. Conjunto independiente máximo (ponderado)                    | (en árbol, sección 9.22) |

Ver también las secciones 4.7.1, 5.4, 5.6, 6.5, 8.3, 8.4 y partes del capítulo 9, para encontrar *más* ejercicios relativos a la programación dinámica.

## 3.6 Soluciones a los ejercicios no resaltados

**Ejercicio 3.2.1.1:** se hace para evitar el operador de división y trabajar solo con enteros. Si, en su lugar, iteramos por  $abcde$ , podemos encontrar un resultado no entero al calcular  $fghij = abcde/N$ .

**Ejercicio 3.2.1.2:** también será AC, ya que  $10! \approx 3$  millones, más o menos igual que el ejercicio de la sección 3.2.1.

**Ejercicio 3.2.2.1:** modificar la función de *backtracking* para que se parezca a este código:

```

1 void backtrack(int c) {
2 if (c == 8 && row[b] == a) { // la solución candidata (a, b) tiene 1 reina
3 printf("%2d %d", ++lineCounter, row[0]+1);
4 for (int j = 1; j < 8; j++) printf(" %d", row[j]+1);
5 printf("\n");
6 for (int r = 0; r < 8; r++) // probar todas las filas posibles
7 if (col == b && r != a) continue; // AÑADIR ESTA LÍNEA
8 if (place(r, c)) { // si se puede poner una reina en esta fila y columna
9 row[c] = r; backtrack(c+1); // poner aquí esta reina y recursividad
10 }
11 }
12 }

```

**Ejercicio 3.3.1.1:** este problema se puede resolver sin ‘encontrar la respuesta de forma binaria’. Simular el viaje una vez. Solo necesitamos encontrar el mayor requisito de combustible en todo el viaje y hacer que el despósito sea suficiente para ello.

**Ejercicio 3.5.1.1:** prenda  $g = 0$ , tomar el tercer modelo (cuesta 8); prenda  $g = 1$ , tomar el primer modelo (cuesta 10); prenda  $g = 2$ , tomar el primer modelo (cuesta 7); dinero utilizado = 25. No queda nada. El caso de prueba C también se puede resolver con un algoritmo voraz.

**Ejercicio 3.5.1.2:** no, esta formulación de estado no funciona. Necesitamos saber cuánto dinero nos queda en cada subproblema, para que podamos determinar si todavía hay suficiente para comprar un determinado modelo de la prenda actual.

**Ejercicio 3.5.1.3:** el código de DP de abajo a arriba modificado es el siguiente:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int g, money, k, TC, M, C, cur;
5 int price[30][30];
6 bool reachable[2][210]; // tabla reachable[SÓLO 2 FILAS][dinero (<= 200)]
7
8 int main() {
9 scanf("%d", &TC);
10 while (TC--) {
11 scanf("%d %d", &M, &C);
12 for (g = 0; g < C; g++) {
13 scanf("%d", &price[g][0]);
14 for (k = 1; k <= price[g][0]; k++)
15 scanf("%d", &price[g][k]);
16 }
17
18 memset(reachable, false, sizeof reachable);
19 for (k = 1; k <= price[0][0]; k++) if (M - price[0][k] >= 0)
20 reachable[0][M - price[0][k]] = true;
21
22 cur = 1; // empezamos con esta fila
23 for (g = 1; g < C; g++) {
24 memset(reachable[cur], false, sizeof reachable[cur]); // reinicia fila
25 for (money = 0; money < M; money++) if (reachable[!cur][money])
26 for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
27 reachable[cur][money - price[g][k]] = true;
28 cur = !cur; // TRUCO IMPORTANTE: voltear las dos filas
29 }
30
31 for (money = 0; money <= M && !reachable[!cur][money]; money++);
32
33 if (money == M+1) printf("no solution\n"); // última fila sin bit activo
34 else printf("%d\n", M - money);
35 } } // return 0;
```

**Ejercicio 3.5.2.1:** la solución en  $O(n^3)$  para suma de rango bidimensional máxima es:

```
1 scanf("%d", &n); // la dimensión de la matriz cuadrada de entrada
2 for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
3 scanf("%d", &A[i][j]);
4 if (j > 0) A[i][j] += A[i][j-1]; // añadir solo columnas de esta fila i
```

```

5 }
6
7 maxSubRect = -127*100*100; // el valor mínimo posible en este problema
8 for (int l = 0; l < n; l++) for (int r = l; r < n; r++) {
9 subRect = 0;
10 for (int row = 0; row < n; row++) {
11 // suma de rango unidimensional máxima de esta fila i
12 if (l > 0) subRect += A[row][r] - A[row][l-1];
13 else subRect += A[row][r];
14
15 // algoritmo de Kadane en las filas
16 if (subRect < 0) subRect = 0; // voraz, reiniciar si la suma < 0
17 maxSubRect = max(maxSubRect, subRect);
18 }
}

```

**Ejercicio 3.5.2.2:** la solución está en la sección 9.33.

**Ejercicio 3.5.2.3:** la solución ya está en ch3\_06\_LIS.cpp/java.

**Ejercicio 3.5.2.4:** la solución de búsqueda completa iterativa para generar y comprobar todos los subconjuntos posibles de tamaño  $n$  se ejecuta en  $O(n \times 2^n)$ . Esto es válido para  $n \leq 20$ , pero muy lento cuando  $n > 20$ . La solución de DP de la sección 3.5.2 se ejecuta en  $O(n \times S)$ . Si  $S$  no es muy grande, podemos trabajar con un  $n$  muy superior a 20 elementos.

### 3.7 Notas del capítulo

Un consejo sobre este capítulo: no te limites a memorizar las soluciones descritas, sino que debes recordar y asimilar el proceso mental y las estrategias de resolución de problemas que hemos visto. Las buenas habilidades de resolución de problemas son más importantes que las soluciones memorizadas de problemas informáticos bien conocidos, cuando nos encontramos con los problemas (normalmente novedosos) de los concursos de programación.

Muchos problemas del ICPC o la IOI requieren una combinación (ver sección 8.4) de diferentes estrategias. Si tuviésemos que elegir un único capítulo de todo este libro que los concursantes deberían dominar completamente, sin duda sería este, especialmente para el caso de la IOI.

En la tabla 3.5, comparamos las cuatro técnicas de resolución de problemas en relación a su posible resultado, según el tipo de problema que se nos plantea. Tanto en esa tabla como en la lista de ejercicios de programación de esta sección, podrás ver que hay *muchos más* problemas de búsqueda completa y de DP que problemas de D&C y voraces. Por lo tanto, recomendamos al lector que se concentre en mejorar sus habilidades sobre la búsqueda completa y la DP antes de ocuparse del D&C o su ‘sentimiento voraz’.

|                | Problema de CS | Problema D&C | Problema voraz | Problema de DP |
|----------------|----------------|--------------|----------------|----------------|
| Solución de CS | AC             | TLE/AC       | TLE/AC         | TLE/AC         |
| Solución D&C   | WA             | AC           | WA             | WA             |
| Solución voraz | WA             | WA           | AC             | WA             |
| Solución de DP | MLE/TLE/AC     | MLE/TLE/AC   | MLE/TLE/AC     | AC             |
| Frecuencia     | Alta           | Muy baja     | Baja           | Alta           |

Tabla 3.5: Comparativa de técnicas de resolución de problemas (orientativa)

Finalizamos este capítulo incidiendo en que, para algunos problemas de la vida real, especialmente aquellos clasificados como NP-complejos [7], muchas de las técnicas tratadas en esta sección no serán válidas. Por ejemplo, el problema de la mochila 0-1 (suma de subconjuntos), que tiene una complejidad de DP de  $O(2^n \times n^2)$ , es demasiado lento si  $n$  es mayor de 19 (ver el **ejercicio 3.5.2.7\***). Para esos problemas, podemos recurrir a heurísticas o técnicas de búsqueda local, como la búsqueda tabú [26, 25], algoritmos genéticos, optimizaciones de colonia de hormigas, templado simulado, búsqueda de haz, etc. Sin embargo, todas estas búsquedas de tipo heurístico no se incluyen en el temario de la IOI [20] y tampoco suelen usarse en el ICPC.

# Capítulo 4

---

## Grafos

*Todo el mundo está separado por una media de  
≈ seis pasos de cualquier otra persona de la Tierra*

— Stanley Milgram

experimento de los seis grados de separación de 1969, [64]

### 4.1 Introducción y motivación

Muchos problemas de la vida real se pueden ver como problemas de grafos. Algunos tienen soluciones eficientes, otros todavía no. En este capítulo, relativamente extenso y con muchas figuras, trataremos problemas de grafos que aparecen normalmente en concursos de programación, los algoritmos que los resuelven y las implementaciones *prácticas* de los mismos. Cubriremos temas como el recorrido básico de grafos, árboles recubridores mínimos, rutas más cortas de origen único o de todos los pares, flujos de red y, también, grafos con propiedades especiales.

Al escribir este capítulo, asumimos que el lector *ya está* familiarizado con la terminología de grafos de la tabla 4.1. Si encuentras algún término desconocido, te recomendamos la lectura de otros libros de referencia como [7, 58] (o consulta en internet), para aclarar su significado.

| Vértices/Nodos | Aristas           | Conjunto $V$ ; tamaño $ V $ | Conjunto $E$ ; tamaño $ E $ | Grafo $G(V, E)$         |
|----------------|-------------------|-----------------------------|-----------------------------|-------------------------|
| No/Ponderado   | No/Dirigido       | Disperso                    | Denso                       | Grado de entrada/salida |
| Camino         | Ciclo             | Aislado                     | Alcanzable                  | Conexo                  |
| Autobucle      | Aristas múltiples | Grafo múltiple              | Grafo simple                | Subgrafo                |
| DAG            | Árbol/Bosque      | Euleriano                   | Bipartito                   | Completo                |

Tabla 4.1: Lista de terminología de grafos importante

También damos por hecho que el lector ya conoce las diferentes formas de representación de la información de los grafos, que se ha tratado en la sección 2.4.1. Es decir, utilizaremos directamente conceptos como matriz de adyacencia, lista de adyacencia, lista de aristas o grafo implícito, sin detenernos a definirlos. Si no estás familiarizado con alguna de estas estructuras de datos de grafos, te remitimos, nuevamente, a la sección 2.4.1.

Nuestra observación, en relación a los problemas de grafos aparecidos en ediciones recientes de las fases regionales del ICPC en Asia, así como en las finales mundiales, pone de manifiesto

que siempre hay, al menos, un problema de grafos (y, posiblemente, más) entre las cuestiones planteadas. Sin embargo, como la variedad de problemas de grafos es tan amplia, la posibilidad de que aparezca uno de ellos en concreto es pequeña. Así que la pregunta es “¿en cuáles nos centramos?”. En nuestra opinión, esta cuestión no tiene una respuesta evidente. Si quieras obtener un buen resultado en el ICPC, no te queda más remedio que estudiar y dominar todas las posibilidades.

En el caso de la IOI, el temario [20] limita las tareas a un subconjunto de los tema tratados en este capítulo. Tiene toda la lógica, pues no se espera que los estudiantes de secundaria, que compiten en la IOI, estén bien formados en muchos algoritmos específicos. Para ayudar a los lectores que deseen tomar parte en la IOI, indicaremos cuándo una sección particular del capítulo queda fuera del temario.

## 4.2 Recorrido de grafos

### 4.2.1 Búsqueda en profundidad

La búsqueda en profundidad (por comodidad, DFS), es un algoritmo sencillo de recorrido de grafos. Empezando desde un vértice de origen, la DFS recorre primero el grafo en su profundidad. Cada vez que la DFS encuentra una división en ramas (un vértice con más de un vecino), elegirá uno de los vecinos no visitados y visitará su vértice. La DFS repite este proceso, descendiendo en profundidad hasta que llega a un vértice terminal. Cuando esto ocurre, la DFS ‘volverá hacia atrás’ y explorará otros vecinos todavía no visitados, si los hay.

Este comportamiento de recorrido de grafos es de fácil implementación con el código recursivo que incluimos a continuación. Nuestra versión de la DFS utiliza un vector de enteros *global vi dfs\_num*, para determinar el estado de cada vértice. En su variante más sencilla, solo utilizamos *vi dfs\_num* para distinguir entre los ‘no visitados’ (usaremos un valor constante **UNVISITED = -1**) y ‘visitados’ (mediante la constante **VISITED = 1**). Inicialmente, se establecen como ‘no visitados’ todos los valores de *dfs\_num*. Después, veremos otros usos que le podemos dar a *vi dfs\_num*. La llamada a *dfs(u)* comienza la DFS desde un vértice *u*, lo marca como ‘visitado’ y, de forma recursiva, se desplazada a cada vecino ‘no visitado’ *v* de *u* (es decir, la arista *u – v* existe en el grafo y *dfs\_num[v] == UNVISITED*).

```
1 typedef pair<int, int> ii; // En este capítulo, usaremos estos tres atajos
2 typedef vector<ii> vii; // de tipos de datos. Pueden parecer confusos, pero
3 typedef vector<int> vi; // son muy útiles en la programación competitiva
4
5 vi dfs_num; // variable global, valores fijados inicialmente a UNVISITED
6
7 void dfs(int u) { // DFS de uso normal: como algoritmo de recorrido de grafo
8 dfs_num[u] = VISITED; // importante: marcamos el vértice como visitado
9 for (int j = 0; j < (int)AdjList[u].size(); j++) { // DS estándar: AdjList
10 ii v = AdjList[u][j]; // v es un par (vecino, peso)
11 if (dfs_num[v.first] == UNVISITED) // importante para evitar ciclos
12 dfs(v.first); // visita recursiva a vecinos no visitados del vértice u
13 } } // para recorrido de grafos simples, ignoramos el peso en v.second
```

La complejidad de tiempo de esta implementación de la DFS depende de la estructura de datos de grafos utilizada. En un grafo con  $V$  vértices y  $E$  aristas, la DFS se ejecutará en  $O(V+E)$  o  $O(V^2)$ , dependiendo de si está almacenada como una lista o una matriz de adyacencia, respectivamente (ver el [ejercicio 4.2.2.2](#)).

En el grafo de ejemplo de la figura 4.1,  $\text{dfs}(0)$ , la llamada a la DFS desde el vértice inicial  $u = 0$ , provocará esta secuencia de visitas:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ . Dicha secuencia es ‘en profundidad’, es decir, la DFS se desplaza al vértice más profundo posible, partiendo del inicial, antes de viajar por otra rama (en este caso no la hay).

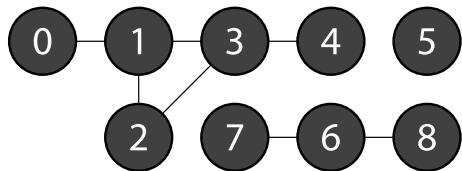


Figura 4.1: Grafo de ejemplo

Hay que destacar que esta secuencia depende mucho de la forma en que ordenemos los vecinos de un vértice<sup>1</sup>, es decir, la secuencia  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4$  (regreso a 3)  $\rightarrow 2$  sería otra posibilidad.

También es importante observar que una llamada a  $\text{dfs}(u)$  solo visitará los vértices que estén *conectados* a  $u$ . Por eso, los vértices 5, 6, 7 y 8 de la figura 4.1, quedan sin visitar después de la llamada a  $\text{dfs}(0)$ .

El código de la DFS que mostramos aquí es muy similar al del *backtracking* recursivo que vimos antes, en la sección 3.2. Si comparamos el pseudocódigo de una implementación de *backtracking* típica (aparece a continuación) con el de la DFS mostrado antes, podemos observar que la diferencia principal radica en el etiquetado de los vértices (estados) visitados. El *backtracking* quita (automáticamente) la etiqueta de vértice visitado (la devuelve al estado anterior), cuando la recursividad vuelve hacia atrás, para permitir que esos vértices (estados) vuelvan a ser visitados desde otra rama. Al no volver a visitar vértices de un grafo general (a través de las comprobaciones de  $\text{dfs\_num}$ ), la DFS se ejecuta en  $O(V+E)$ , mientras que la complejidad de tiempo del *backtracking* es exponencial. **En resumen, el *backtracking* explora todas las rutas (hasta  $V!$ ) desde un vértice origen, mientras que la DFS solo explora una.**

```
void backtrack(estado) {
 if (llega a estado final o estado no válido) // necesitamos una situación
 return; // de terminación o poda para evitar ciclos y acelerar búsqueda
 for cada vecino de este estado // probar todas las permutaciones
 backtrack(vecino);
}
```

### Aplicación de ejemplo: UVa 11902 - Dominator

Enunciado resumido del problema: el vértice  $X$  domina al vértice  $Y$  si cada ruta desde el vértice inicial (0 en este problema) de  $Y$  debe pasar por  $X$ . Si no se puede alcanzar  $Y$  desde el vértice inicial, entonces  $Y$  no tiene ningún dominante. Cada vértice alcanzable desde el inicial se domina a sí mismo. Por ejemplo, en el grafo mostrado en la figura 4.2, el vértice 3 domina al 4, ya que todas las rutas que van desde el vértice 0 al 4 deben pasar por el 3. El vértice 1 no domina al 3, ya que hay una ruta 0-2-3, que no incluye al 1. Nuestra tarea: dado un grafo dirigido, determinar los dominantes de cada vértice.

<sup>1</sup>Para simplificar esta cuestión, ordenaremos normalmente los vértices en base a su número ascendente. Por ejemplo, en la figura 4.1, el vértice 1 tiene a  $\{0, 2, 3\}$  como vecinos, en ese orden.

Este problema trata de comprobar la posibilidad de alcanzar un vértice desde otro inicial (vértice 0). Como el grafo de entrada de este problema es pequeño ( $V < 100$ ), podemos permitirnos utilizar el siguiente algoritmo  $O(V \times V^2 = V^3)$ . Ejecutamos  $\text{dfs}(0)$  en el grafo de entrada, para determinar qué vértices son alcanzables desde el 0. Después, comprobamos qué vértices están dominados por  $X$ , desactivamos (temporalmente) todas las aristas salientes de dicho vértice  $X$  y volvemos a ejecutar  $\text{dfs}(0)$ . Entonces, un vértice  $Y$  no estará dominado por otro  $X$  si, inicialmente,  $\text{dfs}(0)$  no pueden llegar a  $Y$ , o si  $\text{dfs}(0)$  sí que puede llegar a  $Y$ , aunque todas las aristas salientes del vértice  $X$  estén (temporalmente) desactivadas. En caso contrario, el vértice  $Y$  estará dominado por  $X$ . Repetimos este proceso  $\forall X \in [0 \dots V-1]$ .

No es necesario eliminar físicamente el vértice  $X$  del grafo de entrada. Basta con añadir una condición en nuestra rutina de DFS para que detenga el recorrido si llega a  $X$ .

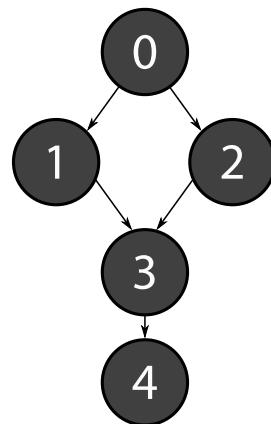


Figura 4.2: UVa 11902

#### 4.2.2 Búsqueda en anchura

La búsqueda en anchura (o BFS) es otro algoritmo de recorrido de grafos. Comenzando en un vértice de origen determinado, la BFS recorrerá el grafo ‘a la ancha’. Es decir, visitará todos los vértices que sean vecinos directos del vértice origen (primera capa), los vecinos de los vecinos directos (segunda capa), y así, sucesivamente, capa a capa.

La BFS comienza con la inserción del vértice origen  $s$  en una cola, y procesa esta de la siguiente manera: quita el vértice  $u$  más alto de la cola, añade todos los vértices vecinos de  $u$  a la cola (normalmente los vecinos estarán ordenados según su número de vértice) y los marca como visitados. Con la ayuda de la cola, la BFS visitará el vértice  $s$  y todos los vértices del componente conexo que contenga  $s$ , capa a capa. El algoritmo BFS también se ejecuta en  $O(V+E)$  o  $O(V^2)$ , según esté el grafo representado como una lista o una matriz de adyacencia, respectivamente (nuevamente, consultar el [ejercicio 4.2.2.2](#)).

Implementar una BFS es sencillo con la STL de C++ o Java. Utilizamos `queue` para ordenar la secuencia de visitas y `vector<int>` (o `vi`) para registrar si un vértice ya ha sido visitado, o no, lo que, al mismo tiempo, registra la distancia (número de capa) de cada vértice desde el origen. Esta característica de cálculo de la distancia se utilizará más adelante, para resolver un caso especial del problema del camino más corto de origen único (secciones 4.4 y 8.2.3).

```

1 // dentro de int main()---sin recursión
2 vi d(V, INF); d[s] = 0; // la distancia del origen s a s es 0
3 queue<int> q; q.push(s); // comenzar en el origen
4 while (!q.empty()) {
5 int u = q.front(); q.pop(); // cola: capa a capa
6 for (int j = 0; j < (int)AdjList[u].size(); j++) {
7 ii v = AdjList[u][j]; // para cada vecino de u
8 if (d[v.first] == INF) { // si v.first no visitado y alcanzable
9 d[v.first] = d[u]+1; // hacer d[v.first] != INF para etiquetarlo
10 q.push(v.first); // v.first a la cola para siguiente iteración
11 }
12 }
13 }

```

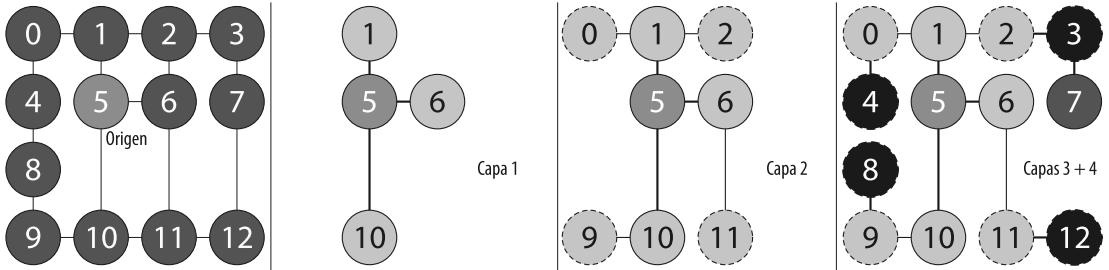


Figura 4.3: Animación de ejemplo de BFS

Si ejecutamos la BFS desde el vértice 5 (es decir, el vértice origen  $s = 5$ ) en el grafo conexo no dirigido de la figura 4.3, visitaremos los vértices en el siguiente orden:

- Capa 0: 5
- Capa 1: 1 → 6 → 10
- Capa 2: 0 → 2 → 11 → 9
- Capa 3: 4 → 3 → 12 → 8
- Capa 4: 7

### Ejercicio 4.2.2.1

Con el fin de comprobar que, para visitar todos los vértices alcanzables desde un vértice de origen, se puede utilizar tanto DFS como BFS, resuelve UVa 11902 - Dominator utilizando BFS.

### Ejercicio 4.2.2.2

¿Por qué DFS y BFS se ejecutan en  $O(V+E)$  si el grafo se almacena como una lista de adyacencia, y es más lento (en  $O(V^2)$ ) si se hace como una matriz de adyacencia? Siguiente pregunta: ¿cuál es la complejidad de tiempo de DFS y BFS, si el grafo se almacena, en su lugar, como una lista de aristas? ¿Qué deberíamos hacer si recibiésemos el grafo de entrada como una lista de aristas y quisiésemos recorrerlo de forma eficiente?

## 4.2.3 Búsqueda de componentes conexos (grafo no dirigido)

DFS y BFS no son solo útiles para recorrer un grafo. También se pueden utilizar para resolver muchos otros problemas de grafos. Los primeros, que mostramos a continuación, se pueden resolver *indistintamente* con DFS o BFS, aunque algunos de los últimos son más propios para DFS exclusivamente.

El hecho de que una sola llamada a `dfs(u)` (o `bfs(u)`) visite únicamente los vértices que son, de hecho, conexos con  $u$ , se puede utilizar para encontrar (y contar el número de) componentes conexos en un grafo *no dirigido* (más adelante, en la sección 4.2.9, veremos un problema similar con un grafo dirigido). Podemos utilizar el siguiente código para reiniciar la DFS (o BFS) desde uno de los vértices no visitados restantes, para encontrar el siguiente componente conexo. Este proceso se repite hasta que se hayan visitado todos los vértices, y tiene una complejidad de tiempo general de  $O(V+E)$ .

```

1 // dentro de int main()---esta es la solución DFS
2 numCC = 0;
3 dfs_num.assign(V, UNVISITED); // estado de todos los vértices a UNVISITED
4 for (int i = 0; i < V; i++) // para cada vértice i en [0..V-1]
5 if (dfs_num[i] == UNVISITED) // si el vértice i no se ha visitado
6 printf("CC %d:", ++numCC), dfs(i), printf("\n"); // aquí 3 líneas
7
8 // Para el grafo de ejemplo de la figura 4.1, esta es la salida:
9 // CC 1: 0 1 2 3 4
10 // CC 2: 5
11 // CC 3: 6 7 8

```

### Ejercicio 4.2.3.1

El UVa 459 - Graph Connectivity es, en esencia, este problema de búsqueda de componentes conexos en un grafo no dirigido. Resuélvelo usando la solución DFS anterior. Sin embargo, también podemos usar una estructura de datos de conjuntos disjuntos para unión-buscar (ver sección 2.4.2) o BFS (ver sección 4.2.2) para resolverlo. ¿Cómo?

#### 4.2.4 Relleno por difusión - etiquetado/coloreado de componentes conexos

La DFS (o BFS) se puede utilizar con fines diferentes a la búsqueda (y conteo) de componentes conexos. Ahora veremos cómo, con un *sencillo cambio* de la `dfs(u)` en  $O(V+E)$  (también es aplicable a la `bfs(u)`), podemos *etiquetar* (también llamado en la terminología de las ciencias de la computación ‘colorear’) y determinar el tamaño de cada componente. Esta variante es conocida como ‘relleno por difusión’, y se suele realizar sobre grafos *implícitos* (normalmente rejillas bidimensionales).

```

1 int dr[] = { 1, 1, 0,-1,-1,-1, 0, 1}; // truco para explorar rejilla 2D
2 int dc[] = { 0, 1, 1, 1, 0,-1,-1,-1}; // vecinos S,SE,E,NE,N,NO,O,SO
3
4 int floodfill(int r, int c, char c1, char c2) { // devuelve tamaño de CC
5 if (r < 0 || r >= R || c < 0 || c >= C) return 0; // fuera de la rejilla
6 if (grid[r][c] != c1) return 0; // no tiene color c1
7 int ans = 1; // suma 1 a ans porque el vértice (r, c) tiene color c1
8 grid[r][c] = c2; // colorea el vértice (r, c) a c2 para evitar ciclos

```

```

9 for (int d = 0; d < 8; d++)
10 ans += floodfill(r+dr[d], c+dc[d], c1, c2);
11 return ans; // el código es limpio porque usamos dr[] y dc[]
12 }
```

### Aplicación de ejemplo: UVa 469 - Wetlands of Florida

Veamos un ejemplo a continuación (UVa 469 - Wetlands of Florida). El grafo implícito es una rejilla bidimensional en la que los vértices representan las celdas y las aristas son las conexiones entre una celda y su vecina al S/SE/E/NE/N/NO/O/SO. Una ‘W’ indica una celda de agua y ‘L’ una de tierra. Se define un área de agua como un grupo de *celdas conexas*, etiquetadas como ‘W’. Podemos etiquetar (y, simultáneamente, calcular el tamaño de) un área de agua, utilizando el relleno por difusión. El siguiente ejemplo muestra la ejecución de un relleno por difusión desde la fila 2, columna 1 (con índices comenzando en 0), sustituyendo ‘W’ por ‘.’.

```

1 // dentro de int main()
2 // leer la rejilla como un array 2D global y leer las coordenadas (fil,col)
3 printf("%d\n", floodfill(row, col, 'W', '.'));
4 // ver extensión del agua
// la respuesta devuelta es 12
5 // LLLLLLLL LLLLLLLL
6 // LLWWLLWLL LL..LLWLL // El tamaño del componente conexo
7 // LWWLLLLLL (R2,C1) L..LLLLL // (las 'W' conexas)
8 // LWWWLWLL L...L..LL // con una 'W' en (fila 2, col. 1) es 12
9 // LLLWWWWLL ==> LLL...LLL
10 // LLLLLLLLLL LLLLLLLLLL // Todas las 'W' conexas se cambian
11 // LLLWWLLWL LLLWWLLWL // por '.' tras el relleno por difusión
12 // LLWLWLLLL LLWLWLLLL
13 // LLLLLLLLLL LLLLLLLLLL
```

#### 4.2.5 Orden topológico (grafo acíclico dirigido)

El orden topológico de un grafo acíclico dirigido (DAG), consiste en la ordenación lineal de los vértices del DAG, de forma que el vértice  $u$  quede situado antes del vértice  $v$ , si existe una arista  $(u \rightarrow v)$  en el DAG. Todo DAG tiene, al menos, un orden topológico, y *posiblemente* más.

Una aplicación del orden topológico es la búsqueda de una posible secuencia de las asignaturas en las que un estudiante universitario se debe matricular, para completar las materias que permitan su graduación. Cada asignatura tiene una serie de requisitos previos. Esos requisitos nunca son cílicos, así que se pueden modelar como un DAG. Al ordenar topológicamente el DAG de requisitos previos de una asignatura, el alumno obtendrá la lista lineal de las asignaturas a las que debe asistir, una tras otra, sin incumplir las restricciones de dichos requisitos previos.

Existen varios algoritmos para ordenar topológicamente. El más sencillo, consiste en modificar ligeramente la implementación de la DFS que hemos mostrado antes, en la sección 4.2.1.

```

1 vi ts; // vector global para guardar el orden topológico invertido
2
3 void dfs2(int u) { // nombre de función diferente en relación a la original
4 dfs_num[u] = VISITED;
5 for (int j = 0; j < (int)AdjList[u].size(); j++) {
6 ii v = AdjList[u][j];
7 if (dfs_num[v.first] == UNVISITED)
8 dfs2(v.first);
9 }
10 ts.push_back(u); // y ya está, este es el único cambio
11
12 // inside int main()
13 ts.clear();
14 dfs_num.assign(V, UNVISITED);
15 for (int i = 0; i < V; i++) // esta parte es igual a buscar CC
16 if (dfs_num[i] == UNVISITED)
17 dfs2(i);
18 // alternativa, llamar primero a: reverse(ts.begin(), ts.end());
19 for (int i = (int)ts.size()-1; i >= 0; i--) // leer hacia atrás
20 printf(" %d", ts[i]);
21 printf("\n");
22
23 // Para el grafo de ejemplo de la figura 4.4, la salida es:
24 // 7 6 0 1 2 5 3 4 (recuerda que puede haber más de un orden válido)

```

En `dfs2(u)`, añadimos  $u$  al final de una lista (vector) de vértices explorados, una vez hayamos visitado todos los subárboles dependientes de  $u$  en el árbol de expansión DFS<sup>2</sup>. Añadimos  $u$  al *final* del vector, porque *vector* de la STL de C++ (Vector en Java) solo permite la *inserción eficiente en  $O(1)$*  desde atrás. La lista estará en orden inverso, pero podemos solucionarlo invirtiendo, a su vez, la salida. Este sencillo algoritmo, para encontrar un orden topológico (válido), se lo debemos a Robert Endre Tarjan. Se ejecuta en  $O(V+E)$ , como la DFS, ya que hace el mismo trabajo que la DFS original, más una operación constante.

Para finalizar el tema de la ordenación topológica, mostramos otro algoritmo utilizado para encontrar el orden topológico: el algoritmo de Kahn [36]. Se parece a una ‘BFS modificada’. Algunos problemas, como el UVa 11060 - Beverages, requieren que sea este algoritmo de Kahn, en vez del basado en DFS que hemos visto antes, el que proporcione el orden topológico necesario.

```

poner los vértices con grado de entrada cero en una cola (de prioridad) Q;
while (Q no está vacía) {
 vértice u = Q.dequeue(); poner u en una lista de ordenación topológica;

```

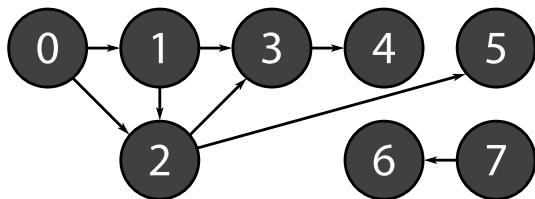


Figura 4.4: Ejemplo de un DAG

<sup>2</sup>El árbol de expansión DFS se trata en más detalle en la sección 4.2.7.

```
 eliminar este vértice u y todas las aristas que salgan del mismo;
 si esa eliminación provoca que el vértice v tenga grado de entrada cero
 Q.enqueue(v); }
```

### Ejercicio 4.2.5.1

¿Por qué añadir el vértice  $u$  al final de `vi ts`, es decir, `ts.push_back(u)`, en el código DFS estándar, es suficiente para encontrar el orden topológico de un DAG?

### Ejercicio 4.2.5.2

¿Puedes identificar otra estructura de datos que permita la inserción eficiente en  $O(1)$  por delante, para que no tengamos que invertir el contenido de `vi ts`?

### Ejercicio 4.2.5.3

¿Qué ocurre si ejecutamos el código de ordenación topológica anterior en un grafo que no sea un DAG?

### Ejercicio 4.2.5.4

El código de ordenación topológica mostrado antes, solo puede generar *un único* orden topológico válido de los vértices de un DAG. ¿Qué deberíamos hacer si queremos mostrar (o contar) *todos* los órdenes topológicos válidos de los vértices de un DAG?

## 4.2.6 Comprobación de grafo bipartito

El grafo bipartito tiene importantes aplicaciones, que veremos más adelante, en la sección 4.7.4. En esta subsección, nos limitaremos a comprobar si un grafo es bipartito (se puede colorear con solo dos colores), para resolver problemas como el UVa 10004 - Bicoloring. Podemos utilizar tanto BFS como DFS para realizar esta comprobación, pero creemos que BFS resulta más natural. El código BFS modificado, que se muestra a continuación, comienza coloreando el vértice de origen (primera capa) con el valor 0, colorea sus vecinos directos (segunda capa) con el valor 1, colorea los vecinos directos de estos últimos (tercera capa) nuevamente con 0, y así sucesivamente, alternando entre 0 y 1, como únicos colores válidos. Si, en algún momento, esto no resulta posible, por ejemplo encontrando una arista con dos extremos del mismo color, entonces podemos concluir que el grafo dado no es bipartito.

```

1 // dentro de int main()
2 queue<int> q; q.push(s);
3 vi color(V, INF); color[s] = 0;
4 bool isBipartite = true; // etiqueta booleana adicional, ahora verdadera
5 while (!q.empty() && isBipartite) { // como en la rutina BFS original
6 int u = q.front(); q.pop();
7 for (int j = 0; j < (int)AdjList[u].size(); j++) {
8 ii v = AdjList[u][j];
9 if (color[v.first] == INF) { // pero en vez de guardar la distancia
10 color[v.first] = 1 - color[u]; // solo guardamos dos colores {0, 1}
11 q.push(v.first); }
12 else if (color[v.first] == color[u]) { // u y v.first mismo color
13 isBipartite = false; break; } } } // hay un conflicto de colores

```

### Ejercicio 4.2.6.1\*

Implementa la comprobación de un grafo bipartito, utilizando DFS.

### Ejercicio 4.2.6.2\*

Se determina que un grafo *sencillo* con  $V$  vértices es bipartito. ¿Cuál es el número máximo de aristas que puede tener ese grafo?

### Ejercicio 4.2.6.3

Demuestra esta afirmación: “un grafo bipartito no tiene ciclos de longitud impar”.

## 4.2.7 Comprobación de las propiedades de las aristas de un grafo

Ejecutar DFS en un grafo conexo genera un *árbol de expansión*<sup>3</sup> DFS (o *bosque de expansión*<sup>4</sup> si el grafo no es conexo). Con la ayuda de un estado de vértice adicional: EXPLORED = 2 (visitado *pero no completado*) además de VISITED (visitado *y completado*), podemos utilizar este árbol (o bosque) de expansión DFS, para clasificar las aristas de un grafo en tres tipos:

1. Arista de árbol: la arista recorrida por DFS, es decir, una arista de un vértice con estado EXPLORED a un vértice con estado UNVISITED.

<sup>3</sup>El árbol de expansión de un grafo conexo  $G$  es un árbol que se extiende por todos (cubre) los vértices de  $G$ , pero utilizando solo un subconjunto de las aristas de  $G$ .

<sup>4</sup>Un grafo no conexo  $G$  tiene varios componentes conexos. Cada uno tiene sus propios subárboles de expansión. Todos los subárboles de expansión de  $G$ , uno por cada componente, forman el denominado bosque de expansión.

2. Arista inversa: una arista que es parte de un ciclo, es decir, una arista de un vértice con estado EXPLORED a un vértice también EXPLORED. Esta es una aplicación importante de este algoritmo. Hay que tener en cuenta que, normalmente, no consideramos que las aristas bidireccionales tengan un ‘ciclo’ (debemos recordar `dfs_parent` para distinguir esto, ver el código a continuación).
3. Aristas adelante/cruzadas de un vértice con estado EXPLORED a otro con estado VISITED. Estos dos tipos de aristas no se suelen comprobar en los concursos de programación.

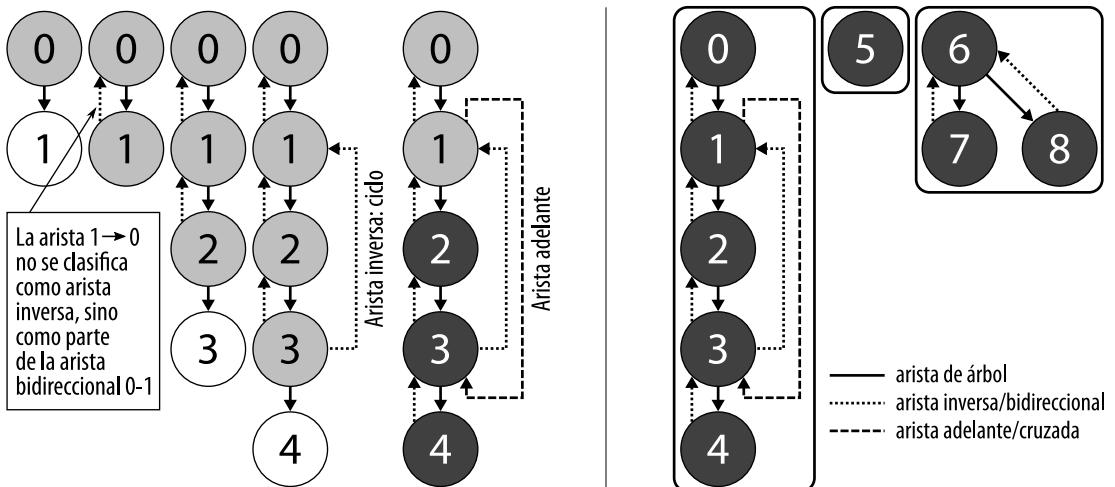


Figura 4.5: Animación de un DFS cuando se ejecuta sobre el grafo de ejemplo de la figura 4.1

La figura 4.5 muestra una animación (de izquierda a derecha) de la llamada a `dfs(0)` (mostrada en más detalle), después `dfs(5)` y, finalmente, `dfs(6)`, sobre el grafo de ejemplo de la figura 4.1. Podemos ver que  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$  es un ciclo (verdadero) y clasificamos la arista  $(3 \rightarrow 1)$  como inversa, mientras que  $0 \rightarrow 1 \rightarrow 0$  no es un ciclo, sino únicamente una arista bidireccional  $(0-1)$ . A continuación, incluimos el código de esta variante de DFS:

```

1 void graphCheck(int u) { // DFS para comprobar las propiedades de la arista
2 dfs_num[u] = EXPLORED; // color u como EXPLORED en vez de VISITED
3 for (int j = 0; j < (int)AdjList[u].size(); j++) {
4 ii v = AdjList[u][j];
5 if (dfs_num[v.first] == UNVISITED) { // arista árbol EXPLORED->UNVISITED
6 dfs_parent[v.first] = u; // el padre de este hijo soy yo
7 graphCheck(v.first);
8 }
9 else if (dfs_num[v.first] == EXPLORED) { // EXPLORED->EXPLORED
10 if (v.first == dfs_parent[u]) // para diferenciar estos dos casos
11 printf(" Two ways (%d, %d)-(%d, %d)\n", u, v.first, v.first, u);
12 else // la aplicación más normal: comprobar si el grafo es cíclico
13 printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
14 }
15 else if (dfs_num[v.first] == VISITED) // EXPLORED->VISITED

```

```

16 printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
17 }
18 dfs_num[u] = VISITED; // tras la recursión, color u como VISITED (HECHO)
19 }
20
21 // dentro de int main()
22 dfs_num.assign(V, UNVISITED);
23 dfs_parent.assign(V, 0); // nuevo vector
24 for (int i = 0; i < V; i++)
25 if (dfs_num[i] == UNVISITED)
26 printf("Component %d:\n", ++numComp), graphCheck(i); // 2 líneas en 1
27
28 // Para el grafo de ejemplo de la figura 4.1, la salida es:
29 // Component 1:
30 // Two ways (1, 0) - (0, 1)
31 // Two ways (2, 1) - (1, 2)
32 // Back Edge (3, 1) (Cycle)
33 // Two ways (3, 2) - (2, 3)
34 // Two ways (4, 3) - (3, 4)
35 // Forward/Cross Edge (1, 3)
36 // Component 2:
37 // Component 3:
38 // Two ways (7, 6) - (6, 7)
39 // Two ways (8, 6) - (6, 8)

```

### Ejercicio 4.2.7.1

Realiza comprobaciones de las propiedades de las aristas sobre el grafo de la figura 4.9. Asume que empiezas la DFS desde el vértice 0. ¿Cuántas aristas inversas encuentras?

#### 4.2.8 Búsqueda de puntos de articulación y puentes (grafo no dirigido)

Problema de referencia: dado un mapa de carreteras (grafo no dirigido) con costes de sabotaje asociados a todas las intersecciones (vértices) y carreteras (aristas), sabotear una única intersección o una única carretera, de forma que la red de carreteras se rompa (desconecte), y hacerlo de la forma menos costosa. Este problema consiste en encontrar el punto de articulación (intersección) o puente (carretera) de menor coste en un grafo no dirigido (mapa de carreteras).

Un ‘punto de articulación’ se define como *un vértice* en un grafo  $G$ , cuya eliminación (todas las aristas incidentes a este vértice también se eliminan) desconecte  $G$ . Un grafo que no tenga ningún punto de articulación se denomina ‘biconexo’. Igualmente, un ‘puente’ se define como *una arista* de un grafo  $G$  cuya eliminación desconecte  $G$ . Estos dos problemas se definen, normalmente, para grafos no dirigidos (son más complejos para grafos dirigidos y su solución requiere otro algoritmo, ver [35]).

A continuación, se describe un algoritmo ingenuo para encontrar puntos de articulación (se puede modificar para encontrar puentes):

1. Ejecutar una DFS (o BFS) en  $O(V+E)$ , para contar el número de componentes conexos (CC) en el grafo original. Normalmente, la entrada será un grafo conexo, así que esta comprobación nos devolverá un componente conexo.
2. Para cada vértice  $v \in V$  en  $O(V)$ :
  - a) Cortar (eliminar) el vértice  $v$  y sus aristas incidentes.
  - b) Ejecutar DFS (o BFS) en  $O(V+E)$  y ver si el número de CC aumenta.
  - c) Si la respuesta es afirmativa,  $v$  es un punto de articulación/vértice de corte; restaurar  $v$  y sus aristas incidentes.

Este algoritmo ingenuo llama a DFS (o BFS)  $O(V)$  veces, por lo que se ejecuta en  $O(V \times (V+E)) = O(V^2 + VE)$ . Pero *no es* el mejor algoritmo ya que, en realidad, podemos ejecutar una DFS en  $O(V+E)$  *una sola vez*, para identificar todos los puntos de articulación y puentes.

Esta variante de DFS, expresada por John Edward Hopcroft y Robert Endre Tarjan (ver [63] y el problema 22.2 en [7]), no es más que otra extensión del código de DFS mostrado anteriormente.

Ahora, mantenemos dos números:  $\text{dfs\_num}(u)$  y  $\text{dfs\_low}(u)$ . En este caso,  $\text{dfs\_num}(u)$  almacena el contador de iteraciones cuando se visita el vértice  $u$  *por primera vez* (no solo para distinguir UNVISITED de EXPLORED/VISITED). El otro número,  $\text{dfs\_low}(u)$ , almacena el menor  $\text{dfs\_num}$  alcanzable desde el subárbol de expansión DFS actual de  $u$ . Al principio,  $\text{dfs\_low}(u) = \text{dfs\_num}(u)$ , cuando se visita el vértice  $u$  por primera vez. Despues,  $\text{dfs\_low}(u)$  solo se puede reducir si hay un ciclo (existe una arista inversa). No actualizamos  $\text{dfs\_low}(u)$  con una arista inversa  $(u, v)$  si  $v$  es un parente directo de  $u$ . Ver la figura 4.6 para mayor claridad.

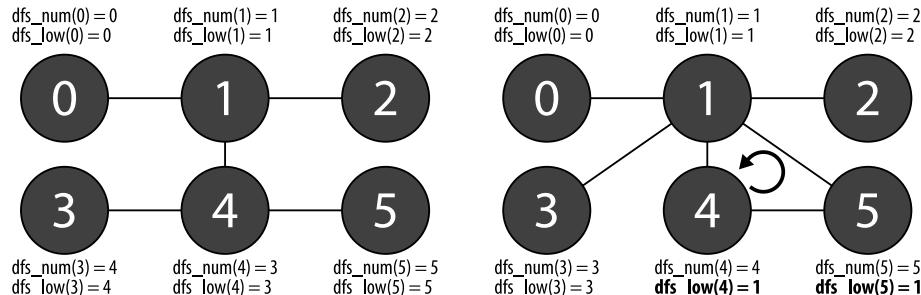


Figura 4.6: Introducimos dos atributos adicionales a la DFS:  $\text{dfs\_num}$  y  $\text{dfs\_low}$

En ambos grafos ejecutamos la variante DFS desde el vértice 0. Supongamos que, para el grafo de la parte izquierda de la figura 4.6, la secuencia de visita es 0 (en la iteración 0)  $\rightarrow$  1 (1)  $\rightarrow$  2 (2) (volvemos a 1)  $\rightarrow$  4 (3)  $\rightarrow$  3 (4) (volvemos a 4)  $\rightarrow$  5 (5). Hay que notar que estos contadores de iteraciones aparecen correctamente en  $\text{dfs\_num}$ . Como en este grafo no hay ninguna arista inversa, en todos los casos  $\text{dfs\_low} = \text{dfs\_num}$ .

Ahora, supongamos que, para el grafo de la parte derecha de la figura 4.6, la secuencia de visita es 0 (iteración 0)  $\rightarrow$  1 (1)  $\rightarrow$  2 (2) (volvemos a 1)  $\rightarrow$  3 (3) (volvemos a 1)  $\rightarrow$  4 (4)  $\rightarrow$  5 (5). En este punto del árbol de expansión DFS, hay una arista inversa importante, que forma un ciclo, es decir, la arista 5-1 es parte de 1-4-5-1. Esto provoca que los vértices 1, 4 y 5 sean capaces de llegar al vértice 1 (con  $\text{dfs\_num}$  1). Por lo tanto, todos los  $\text{dfs\_low}$  de {1, 4, 5} son 1.

Cuando estamos en un vértice  $u$ , con  $v$  como su vecino y  $\text{dfs\_low}(v) \geq \text{dfs\_num}(u)$ , sabemos que  $u$  es un vértice de articulación. Esto se debe a que el hecho de que  $\text{dfs\_low}(v)$  *no sea menor*

que  $\text{dfs\_num}(u)$ , implica que *no hay una arista inversa* desde el vértice  $v$  que puede alcanzar otro vértice  $w$  con un  $\text{dfs\_num}(w)$  menor que  $\text{dfs\_num}(u)$ . Un vértice  $w$  con un  $\text{dfs\_num}(w)$  menor que el vértice  $u$  con  $\text{dfs\_num}(u)$ , implica que  $w$  es un antecesor de  $u$  en el árbol de expansión de la DFS. Esto significa que, para llegar a los antecesores de  $u$  desde  $v$ , *debemos* pasar por el vértice  $u$ . Por lo tanto, eliminar el vértice  $u$  desconectará el grafo.

Sin embargo, existe un **caso especial**: la raíz del árbol de expansión de la DFS (el vértice elegido como inicio de la llamada a la DFS) es un punto de articulación solo si tiene más de un hijo en el árbol de expansión de la DFS (un caso trivial que no es detectado por este algoritmo). Ver la figura 4.7 para más detalles.

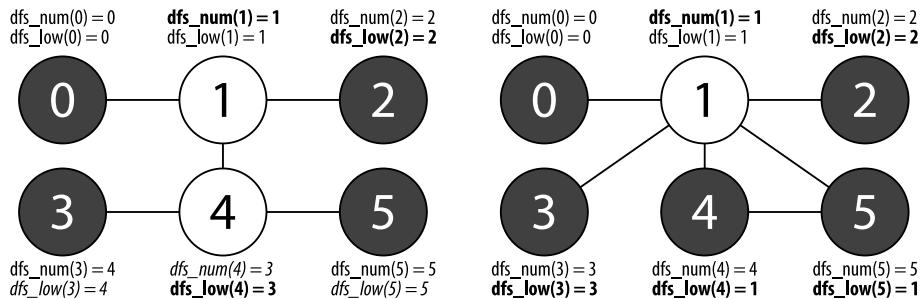


Figura 4.7: Búsqueda de puntos de articulación con  $\text{dfs\_num}$  y  $\text{dfs\_low}$

En el grafo de la parte izquierda de la figura 4.7, los vértices 1 y 4 son puntos de articulación, porque, por ejemplo, en la arista 1-2, vemos que  $\text{dfs\_low}(2) \geq \text{dfs\_num}(1)$ , y en la arista 4-5, también vemos que  $\text{dfs\_low}(5) \geq \text{dfs\_num}(4)$ . En el grafo de la parte derecha de la figura 4.7, solo el vértice 1 es un punto de articulación, porque, por ejemplo, en la arista 1-5,  $\text{dfs\_low}(5) \geq \text{dfs\_num}(1)$ .

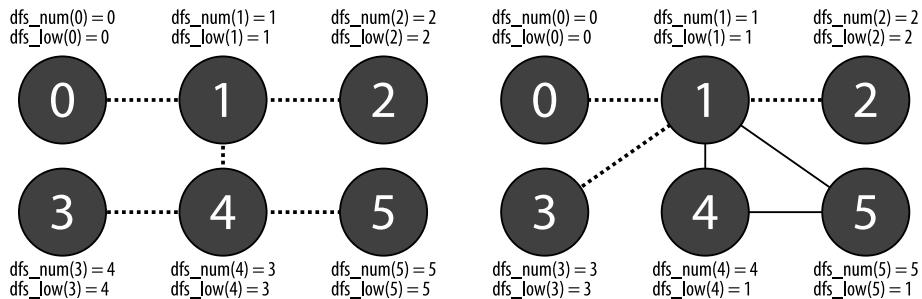


Figura 4.8: Búsqueda de puentes, también con  $\text{dfs\_num}$  y  $\text{dfs\_low}$

El proceso de encontrar puentes es similar. Cuando  $\text{dfs\_low}(v) > \text{dfs\_num}(u)$ , entonces la arista  $u-v$  es un puente (eliminamos la prueba de igualdad ‘ $=$ ’ al buscar puentes). En la figura 4.8, casi todas las aristas son puentes para los grafos izquierdo y derecho. Solo las aristas 1-4, 4-5 y 5-1 no son puentes en el grafo derecho (de hecho, forman un ciclo). Esto se debe a que, por ejemplo, en la arista 4-5 tenemos  $\text{dfs\_low}(5) \leq \text{dfs\_num}(4)$ , es decir, incluso si esta arista 4-5 se elimina, sabemos con seguridad que el vértice 5 puede llegar hasta el vértice 1 a través de *otro camino*, que evita el vértice 4, ya que  $\text{dfs\_low}(5) = 1$  (el otro camino es, realidad, la arista 5-1). Se incluye el código a continuación:

```

1 void articulationPointAndBridge(int u) {
2 dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
3 for (int j = 0; j < (int)AdjList[u].size(); j++) {
4 ii v = AdjList[u][j];
5 if (dfs_num[v.first] == UNVISITED) { // un arista de árbol
6 dfs_parent[v.first] = u;
7 if (u == dfsRoot) rootChildren++; // caso especial si u es una raiz
8
9 articulationPointAndBridge(v.first);
10
11 if (dfs_low[v.first] >= dfs_num[u]) // para un punto de articulación
12 articulation_vertex[u] = true; // guardar primero esta información
13 if (dfs_low[v.first] > dfs_num[u]) // para un puente
14 printf(" Edge (%d, %d) is a bridge\n", u, v.first);
15 dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); //actualiza dfs_low[u]
16 }
17 else if (v.first != dfs_parent[u]) // arista inversa y no ciclo directo
18 dfs_low[u] = min(dfs_low[u], dfs_num[v.first]); //actualiza dfs_low[u]
19 } }
20 // dentro de int main()
21 dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
22 dfs_parent.assign(V, 0); articulation_vertex.assign(V, 0);
23 printf("Bridges:\n");
24 for (int i = 0; i < V; i++)
25 if (dfs_num[i] == UNVISITED) {
26 dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
27 articulation_vertex[dfsRoot] = (rootChildren > 1); } // caso especial
28 printf("Articulation Points:\n");
29 for (int i = 0; i < V; i++)
30 if (articulation_vertex[i])
31 printf(" Vertex %d\n", i);
32
33 // Para el caso de ejemplo de las fig. 4.6/4.7/4.8 DERECHA, la salida es:
34 // Bridges:
35 // Edge (1, 2) is a bridge
36 // Edge (1, 3) is a bridge
37 // Edge (0, 1) is a bridge
38 // Articulation Points:
39 // Vertex 1

```

### Ejercicio 4.2.8.1

Examina el grafo de la figura 4.1 sin ejecutar el algoritmo visto. ¿Qué vértices son puntos de articulación y qué aristas son puentes? Ahora, ejecuta el algoritmo y comprueba si los `dfs_num` y `dfs_low` calculados de cada vértice de la figura 4.1 se pueden utilizar para identificar los mismos puntos de articulación y vértices que has encontrado a mano.

#### 4.2.9 Búsqueda de componentes fuertemente conexos (grafo dirigido)

Una aplicación más de la DFS es la búsqueda de componentes *fuertemente* conexos en un grafo *dirigido*, por ejemplo en UVa 11838 - Come and Go. Este es un problema diferente al de la búsqueda de componentes conexos en un grafo no dirigido. En la figura 4.9, tenemos un grafo similar al de la figura 4.1, con la diferencia de que ahora las aristas son dirigidas. Aunque el grafo de la figura 4.9 parece que tiene un componente ‘conexo’, en realidad no está ‘fuertemente conexo’. En los grafos dirigidos nos interesa más el concepto de ‘componente fuertemente conexo (SCC)’. Un SCC se define de la siguiente manera: si elegimos cualquier par de vértices  $u$  y  $v$  en el SCC, podemos encontrar un camino de  $u$  a  $v$  y viceversa. En realidad, hay tres SCC en la figura 4.9, como está resaltado con las tres cajas:  $\{0\}$ ,  $\{1, 3, 2\}$  y  $\{4, 5, 7, 6\}$ . Si se contraen estos SCC (se sustituyen por vértices mayores), forman un DAG (ver sección 8.4.3).

Existen, al menos, dos algoritmos conocidos para encontrar los SCC: el de Kosaraju, explicado en [7], y el de Tarjan [63]. En esta sección, utilizaremos la segunda, ya que es una extensión natural del contenido anterior sobre la búsqueda de puntos de articulación y puentes, también gracias a Tarjan. Veremos el algoritmo de Kosaraju más adelante, en la sección 9.17.

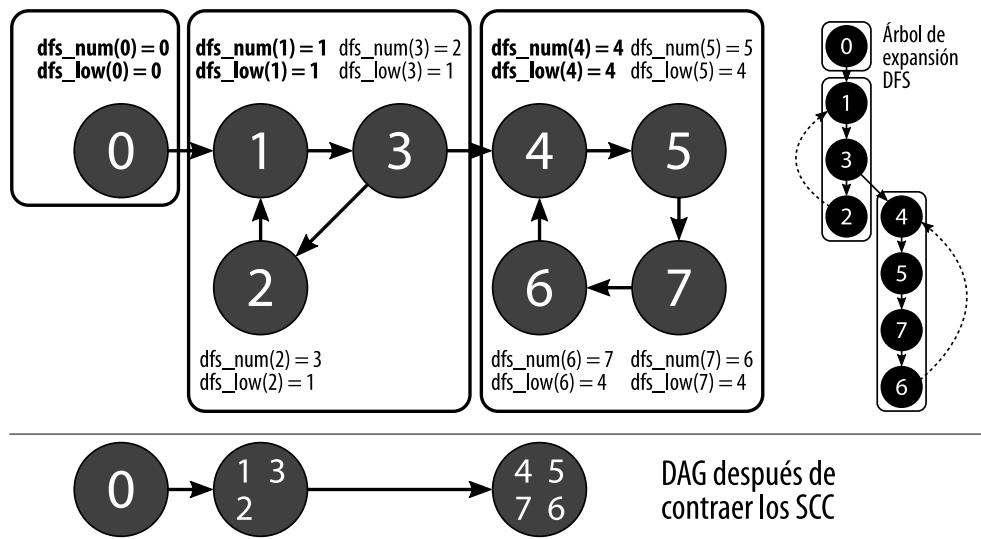


Figura 4.9: Ejemplo de un grafo dirigido y sus SCC

La idea básica del algoritmo es que los SCC forman subárboles en el árbol de expansión de la DFS (comparar el grafo dirigido original y el de expansión DFS de la figura 4.9). Además de calcular  $\text{dfs\_num}(u)$  y  $\text{dfs\_low}(u)$  para cada vértice, añadimos el vértice  $u$  al final de una pila  $S$  (aquí la pila se implementa mediante un vector) y mantenemos la información de qué vértices están siendo explorados, mediante `vi visited`. La condición para actualizar  $\text{dfs\_low}(u)$  es ligeramente diferente a la del algoritmo DFS anterior, de búsqueda de puntos de articulación y puentes. En este caso, solo los vértices que están marcados como `visited` (parte del SCC actual) pueden actualizar  $\text{dfs\_low}(u)$ . Ahora, si tenemos el vértice  $u$  en este árbol de expansión DFS con  $\text{dfs\_low}(u) = \text{dfs\_num}(u)$ , podemos concluir que  $u$  es la raíz (el inicio) de un SCC (observar los vértices 0, 1 y 4 en la figura 4.9) y los miembros de esos SCC se pueden identificar obteniendo el contenido actual de la pila  $S$ , hasta que volvamos a llegar al vértice  $u$  (la raíz).

En la figura 4.9, el contenido de  $S$  es  $\{0, 1, 3, 2, \underline{4}, 5, 7, 6\}$ , cuando el vértice 4 está identificado como la raíz de un SCC ( $\text{dfs\_low}(4) = \text{dfs\_num}(4) = 4$ ), por lo que obtenemos los elementos de  $S$ , de uno en uno, hasta que lleguemos al vértice 4, y tendremos este SCC:  $\{6, 7, 5, 4\}$ . A continuación, el contenido de  $S$  será  $\{0, \underline{1}, \underline{3}, 2\}$ , cuando el vértice 1 sea identificado como la raíz de otro SCC ( $\text{dfs\_low}(1) = \text{dfs\_num}(1) = 1$ ), así que, igualmente, obtendremos los elementos de  $S$ , de uno en uno, hasta alcanzar el vértice 1, y tendremos el SCC:  $\{2, 3, 1\}$ . Por último, nos quedará un SCC con un solo miembro:  $\{0\}$ .

El siguiente código explora el grafo dirigido e informa de sus SCC. Este código es, básicamente, una modificación del código DFS estándar. La parte recursiva es similar a la de la DFS y la parte de identificación de los SCC se ejecutará en un tiempo amortizado de  $O(V)$ , ya que cada vértice solo puede pertenecer a un SCC y, por lo tanto, aparecerá una sola vez. El tiempo de ejecución total de este algoritmo sigue siendo  $O(V+E)$ .

```

1 vi dfs_num, dfs_low, S, visited; // variables globales
2
3 void tarjanSCC(int u) {
4 dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
5 S.push_back(u); // guarda u en un vector según el orden de visita
6 visited[u] = 1;
7 for (int j = 0; j < (int)AdjList[u].size(); j++) {
8 ii v = AdjList[u][j];
9 if (dfs_num[v.first] == UNVISITED)
10 tarjanSCC(v.first);
11 if (visited[v.first]) // condición para actualización
12 dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }

13
14 if (dfs_low[u] == dfs_num[u]) { // si es la raíz (inicio) de un SCC
15 printf("SCC %d:", ++numSCC); // se realiza después de la recursión
16 while (1) {
17 int v = S.back(); S.pop_back(); visited[v] = 0;
18 printf(" %d", v);
19 if (u == v) break; }
20 printf("\n");
21 } }
22
23 // dentro de int main()
24 dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
25 dfsNumberCounter = numSCC = 0;
26 for (int i = 0; i < V; i++)
27 if (dfs_num[i] == UNVISITED)
28 tarjanSCC(i);

```



ch4\_01\_dfs.cpp



ch4\_02\_UVa469.cpp



ch4\_01\_dfs.java



ch4\_02\_UVa469.java

### Ejercicio 4.2.9.1

Demuestra como cierta (o falsa) esta afirmación: “si dos vértices están en el mismo SCC, no hay ningún camino entre ellos que pueda salir de ese SCC”.

### Ejercicio 4.2.9.2\*

Escribe un código que tome como entrada un grafo dirigido y lo convierta en un grafo acíclico dirigido (DAG), mediante la contracción de los SCC (por ejemplo, figura 4.9, de arriba a abajo). Ver una aplicación de ejemplo en la sección 8.4.3.

## Recorrido de grafos en concursos de programación

Es notable que los sencillos algoritmos de recorrido de grafos DFS y BFS tengan tantas variantes interesantes que permitan su uso para resolver varios problemas de grafos, además de los de sus formas básicas de recorridos. En el ICPC, puede aparecer cualquiera de estas variantes. En la IOI, también pueden aparecer tareas creativas que implican recorrido de grafos.

Es raro que se pida el uso de DFS (o BFS) *per se* para encontrar componentes conexos en un grafo no dirigido, aunque su variante de relleno por difusión ha sido uno de los tipos de problema más frecuentes *en el pasado*. Nos da la sensación de que el número de problemas (nuevos) de relleno por difusión está disminuyendo.

La ordenación topológica tampoco se utiliza habitualmente de forma independiente, pero es un paso de procesamiento previo muy útil para la ‘DP en un DAG (implícito)’, ver la sección 4.7.1. La versión más sencilla del código de ordenación topológica es muy fácil de recordar, ya que es una variante simple de la DFS. Su alternativa, el algoritmo de Kahn (la ‘BFS modificada’, que solo añade a la cola vértices con grados de entrada 0) es igual de sencilla.

También es bueno recordar soluciones eficientes en  $O(V + E)$ , para la comprobación de grafos bipartitos, comprobación de propiedades de aristas y búsqueda de puntos de articulación y puentes pero, como se ve en el juez UVa (y en muchos ICPC regionales asiáticos recientes), ya no se utilizan en muchos problemas.

El conocimiento del algoritmo de SCC de Tarjan puede resultar útil para resolver problemas modernos, en los que uno de sus subproblemas implique grafos dirigidos que ‘necesiten transformarse’ a DAG, mediante ciclos de contracción, ver la sección 8.4.3. El código que mostramos en este libro puede ser un contenido interesante para aquellos concursos de programación que permiten utilizar bibliotecas de código impresas, como el ICPC. Sin embargo, en la IOI, el tema de los componentes fuertemente conexos está excluido del temario de 2009 [20].

Aunque muchos de los problemas de grafos tratados en esta sección se pueden resolver tanto por DFS como por BFS, pensamos que, en bastantes casos, es más fácil hacerlo mediante la primera, recursiva y con menor consumo de memoria. Normalmente, no utilizamos BFS para problemas de recorrido de grafos puro, pero sí para problemas de caminos más cortos de origen único en grafos no ponderados (ver sección 4.4). La tabla 4.2 muestra una comparativa importante entre estos dos populares algoritmos de recorrido de grafos.

|                | $O(V+E)$ DFS                                                                  | $O(V+E)$ BFS                                                |
|----------------|-------------------------------------------------------------------------------|-------------------------------------------------------------|
| <b>Pros</b>    | Normalmente usa menos memoria, encuentra puntos de articulación, puentes, SCC | Puede resolver el SSSP en grafos no ponderados              |
| <b>Contras</b> | No puede resolver el SSSP en grafos no ponderados                             | Normalmente usa más memoria (no es bueno en grafos grandes) |
| <b>Código</b>  | Más fácil de escribir                                                         | Un poco más largo                                           |

Tabla 4.2: Tabla de decisión de algoritmos de recorrido de grafos

En la dirección web indicada a continuación, proporcionamos una animación del algoritmo DFS/BFS. Puedes utilizarla para mejorar tu compresión sobre esta materia.



## Ejercicios de programación

### Ejercicios de programación relativos al recorrido de grafos:

#### Solo recorrido de grafos

1. UVa 00118 - Mutant Flatworld Explorers (recorrido de un grafo *implícito*)
2. UVa 00168 - Theseus and the ... (matriz de adyacencia; procesado; recorrido)
3. UVa 00280 - Vertex (comprobación sencilla de si es alcanzable; recorrido del grafo)
4. UVa 00318 - Domino Effect (recorrido del grafo; cuidado con los casos límite)
5. UVa 00614 - Mapping the Route (recorrido de grafo *implícito*)
6. UVa 00824 - Coast Tracker (recorrido de grafo *implícito*)
7. UVa 10113 - Exchange Rates (solo recorrido de grafos; pero utiliza fracciones y GCD; ver las secciones relevantes en el capítulo 5)
8. UVa 10116 - Robot Motion (recorrido de grafo *implícito*)
9. UVa 10377 - Maze Traversal (recorrido de grafo *implícito*)
10. UVa 10687 - Monitoring the Amazon (construir grafo; geometría; alcanzable)
11. **UVa 11831 - Sticker Collector ... \*** (grafo *implícito*; el orden de entrada es 'NSEO')
12. UVa 11902 - Dominator (desactivar vértices de uno en uno; comprobar si el acceso desde el vértice 0 cambia)
13. **UVa 11906 - Knight in a War Grid \*** (DFS/BFS para comprobar si es alcanzable, varios casos complicados; cuidado cuando  $M = 0$ ,  $N = 0$  o  $M = N$ )
14. UVa 12376 - As Long as I Learn, I Live (recorrido voraz simulado en un DAG)
15. **UVa 12442 - Forwarding Emails \*** (DFS modificada; grafo especial)
16. UVa 12582 - Wedding of Sultan (recorrido DFS del grafo dado; contar el grado de cada vértice)
17. IOI 2011 - Tropical Garden (recorrido de grafos; DFS; implica círculos)

#### Relleno por difusión/Búsqueda de componentes conexos

1. UVa 00260 - Il Gioco dell'X (6 vecinos por celda)
2. UVa 00352 - The Seasonal War (contar el número de CC; ver UVa 572)

3. UVa 00459 - Graph Connectivity  
(también se resuelve con UFDS)
4. UVa 00469 - Wetlands of Florida  
(contar el tamaño de un CC)
5. UVa 00572 - Oil Deposits  
(contar el número de CC)
6. UVa 00657 - The Die is Cast  
(aquí hay tres 'colores')
7. UVa 00722 - Lakes  
(contar el tamaño de los CC)
8. UVa 00758 - The Same Game  
(relleno por difusión y más)
9. UVa 00776 - Monkeys in a Regular ...  
(etiquetar CC con índices; formato de salida)
10. UVa 00782 - Countour Painting  
(sustituir los espacios en la rejilla; similar a UVa 784 y 785)
11. UVa 00784 - Maze Exploration  
(similar a UVa 782 y 785)
12. UVa 00785 - Grid Colouring  
(similar a UVa 782 y 784)
13. UVa 00852 - Deciding victory in Go  
(interesante juego de mesa 'Go')
14. UVa 00871 - Counting Cells in a Blob  
(buscar el tamaño del CC más grande)
15. **UVa 01103 - Ancient Messages \***  
(LA 5130, World Finals Orlando11; pista importante: cada jeroglífico tiene un número único de componentes conexos blancos; es un ejercicio de implementación para procesar la entrada y ejecutar relleno por difusión para determinar el número de CC blancos dentro de cada jeroglífico)  
(contar y clasificar CC con color similar)
16. UVa 10336 - Rank the Languages  
(encontrar CC y clasificarlos por tamaño)
17. UVa 10946 - You want what filled?  
(comprobar isomorfismo del grafo: tedioso; con componentes conexos)
18. UVa 10707 - 2D - Nim  
(relleno por difusión complicado, ya que implica desplazamiento)
19. **UVa 11094 - Continents \***  
(relleno por difusión y cumplir con las restricciones dadas)
20. UVa 11110 - Equidivisions  
(contar el número de CC)
21. UVa 11244 - Counting Stars  
(se puede hacer relleno por difusión capa a capa; sin embargo, hay otras formas de resolverlo como, por ejemplo, buscando los patrones)
22. UVa 11470 - Square Sums  
(a diferencia de UVa 11504, tratamos los SCC como CC)
23. UVa 11518 - Dominos 2  
(relleno por difusión con restricción adicional)
24. UVa 11561 - Getting Gold  
(buscar CC más grande con mayor PPA media)
25. UVa 11749 - Poor Trade Advisor  
(interesante variación de relleno por difusión)
26. **UVa 11953 - Battleships \***

### Orden topológico

1. UVa 00124 - Following Orders  
(usar *backtracking* para generar ordenaciones topológicas válidas)
2. UVa 00200 - Rare Order  
(orden topológico)
3. **UVa 00872 - Ordering \***  
(similar a UVa 124; usar *backtracking*)
4. **UVa 10305 - Ordering Tasks \***  
(basta con ejecutar algoritmo de ordenación topológica)
5. **UVa 11060 - Beverages \***  
(algoritmo de Kahn; ordenación topológica BFS modificada)
6. UVa 11686 - Pick up sticks  
(ordenación topológica y comprobación de ciclos)

Ver también DP en problemas de DAG (implícito) (sección 4.7.1).

### Comprobación de grafo bipartito

1. **UVa 10004 - Bicoloring \***  
(comprobación de grafo bipartito)
2. UVa 10505 - Montesco vs Capuleto  
(bipartito; tomar max(izquierda, derecha))
3. **UVa 11080 - Place the Guards \***  
(comprobación de grafo bipartito; casos complicados)
4. **UVa 11396 - Claw Decomposition \***  
(solo comprobación de grafo bipartito)

### Búsqueda de puntos de articulación y puentes

- |                                         |                                      |
|-----------------------------------------|--------------------------------------|
| 1. <b>UVa 00315 - Network *</b>         | (búsqueda de puntos de articulación) |
| 2. UVa 00610 - Street Directions        | (búsqueda de puentes)                |
| 3. <b>UVa 00796 - Critical Links *</b>  | (búsqueda de puentes)                |
| 4. UVa 10199 - Tourist Guide            | (búsqueda de puntos de articulación) |
| 5. <b>UVa 10765 - Doves and Bombs *</b> | (búsqueda de puntos de articulación) |

### Búsqueda de componentes fuertemente conexos

- |                                         |                                                                                                                                                  |
|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. <b>UVa 00247 - Calling Circles *</b> | (SCC y mostrar solución)                                                                                                                         |
| 2. UVa 01229 - Sub-dictionary           | (LA 4099 - Iran07; identificar SCC del grafo; esos vértices y los que tienen caminos hacia ellos (para entender las palabras), son la respuesta) |
| 3. UVa 10731 - Test                     | (SCC y mostrar solución)                                                                                                                         |
| 4. <b>UVa 11504 - Dominos *</b>         | (problema interesante: contar el número de SCC sin arista entrante desde un vértice fuera de ese SCC)                                            |
| 5. UVa 11709 - Trust Groups             | (hallar el número de SCC)                                                                                                                        |
| 6. UVa 11770 - Lighting Away            | (similar a UVa 11504)                                                                                                                            |
| 7. <b>UVa 11838 - Come and Go *</b>     | (comprobar si el grafo es fuertemente conexo)                                                                                                    |

## 4.3 Árbol de expansión mínimo (MST)

### 4.3.1 Introducción y motivación

Problema de referencia: dado un grafo conexo, no dirigido y ponderado  $G$  (ver el grafo más a la izquierda de la figura 4.10), seleccionar un subconjunto de aristas  $E' \in G$  de forma que el grafo  $G$  se mantenga conexo y que el peso total de las aristas seleccionadas sea mínimo.

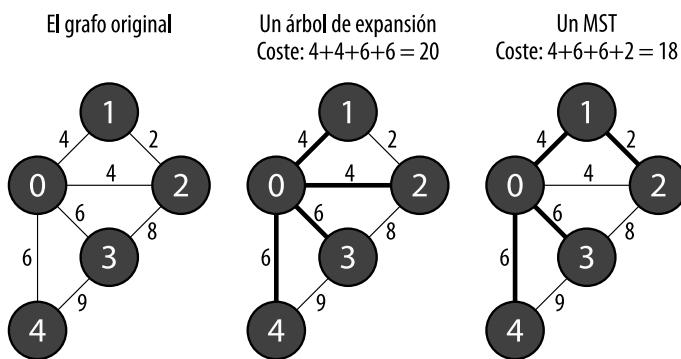


Figura 4.10: Ejemplo de un problema de MST

Para satisfacer el criterio de conectividad, necesitamos, al menos,  $V - 1$  aristas que formen un *árbol*, y este árbol, que se debe expandir sobre (cubrir) todos los  $V \in G$ , es el árbol de expansión. Puede haber varios árboles de expansión válidos en  $G$ , ver el centro y la derecha de la figura

4.10, incluyendo los árboles de expansión DFS y BFS que hemos visto en la pasada sección 4.2. Entre los árboles de expansión posibles de  $G$ , hay algunos (al menos uno) que satisface el criterio de peso mínimo.

Este problema es conocido como el del árbol de expansión mínimo (MST), y tiene muchas aplicaciones prácticas. Por ejemplo, podemos modelar el problema de la construcción de una red de carreteras en pueblos remotos como un problema de MST. Los vértices son los pueblos. Las aristas son las carreteras que se podrían construir entre esos pueblos. El coste de construir una carretera que conecte los pueblos  $i$  y  $j$  es el peso de la arista  $(i, j)$ . El MST de este grafo es, por lo tanto, la red de carreteras de coste mínimo que conecte todos los pueblos. El juez UVa [49] cuenta con algunos problemas de MST básicos como, por ejemplo, los números 908, 1174, 1208, 10034, 11631, etc.

Este problema de MST se puede resolver utilizando algunos algoritmos bien conocidos, como los de Prim y Kruskal. Ambos son voraces, explicados en muchos libros de texto sobre ciencias de la computación [7, 58, 40, 60, 42, 1, 38, 8]. El peso del MST generado por ambos es único, pero puede haber más de un árbol de expansión con el mismo peso de MST.

### 4.3.2 Algoritmo de Kruskal

El algoritmo de Joseph Bernard Kruskal Jr., comienza ordenando las  $E$  aristas en base al peso no decreciente. Esto es fácil de hacer almacenando las aristas en una estructura de datos de lista de aristas (ver la sección 2.4.1) y, luego, ordenando las aristas en base a su peso no decreciente. Después, el algoritmo de Kruskal intenta añadir *vorazmente* cada arista al MST, siempre que esa acción no forme un ciclo. Esta comprobación de los ciclos se lleva a cabo fácilmente, utilizando conjuntos disjuntos para unión-buscar, tratados en la sección 2.4.2. El código es corto, porque hemos separado la implementación de los conjuntos disjuntos para unión-buscar en otra clase. El tiempo de ejecución total de este algoritmo es  $O(\text{ordenación} + \text{intento de añadir cada arista} \times \text{coste de las operaciones de unión-buscar}) = O(E \log E + E \times (\approx 1)) = O(E \log E) = O(E \log V^2) = O(2 \times E \log V) = O(E \log V)$ .

```
1 // dentro de int main()
2 vector< pair<int, ii> > EdgeList; // (peso, dos vértices) de la arista
3 for (int i = 0; i < E; i++) {
4 scanf("%d %d %d", &u, &v, &w); // leer la 3-tupla: (u, v, w)
5 EdgeList.push_back(make_pair(w, ii(u, v))); } // (w, u, v)
6 sort(EdgeList.begin(), EdgeList.end()); // por peso de arista O(E log E)
7 // nota: el objeto pair tiene función de comparación integrada
8 int mst_cost = 0, num_taken = 0;
9 UnionFind UF(V); // todos los V son inicialmente conjuntos disjuntos
10 for (int i = 0; i < E && num_taken < V-1; i++) { // por cada arista, O(E)
11 pair<int, ii> front = EdgeList[i];
12 if(!UF.isSameSet(front.second.first, front.second.second)) { //comprobar
13 num_taken++; // se toma una arista más como parte del MST
14 mst_cost += front.first; // se añade el peso de e al MST
15 UF.unionSet(front.second.first, front.second.second); // enlazarlos
16 } } // nota: el coste de tiempo de ejecución de UDFS es muy bajo
17 // nota: el número de conjuntos disjuntos llegará a 1 en un MST válido
18 printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

La figura 4.11 muestra la ejecución, paso a paso, del algoritmo de Kruskal sobre el grafo mostrado en la parte izquierda de la figura 4.10. El MST final no es único.

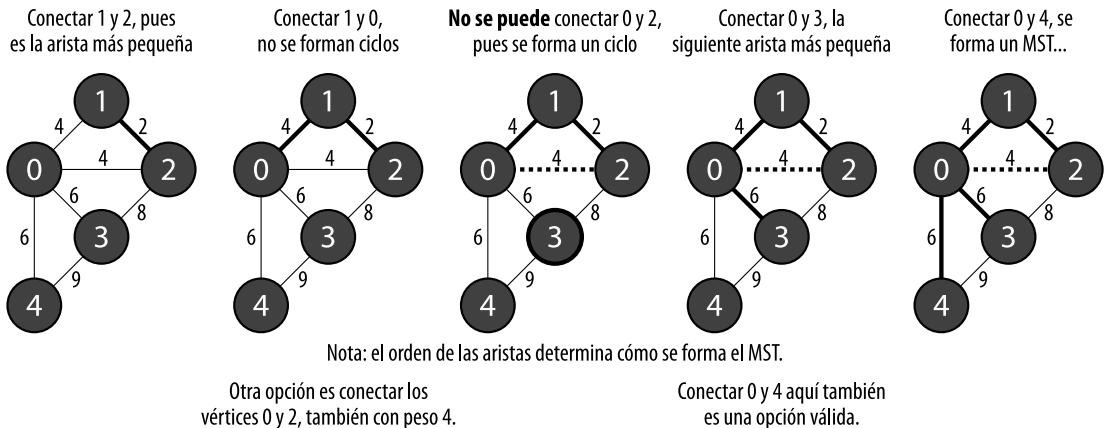


Figura 4.11: Animación del algoritmo de Kruskal para un problema de MST

### Ejercicio 4.3.2.1

En el código mostrado, detenemos el algoritmo de Kruskal en el momento en el que ha puesto  $V - 1$  aristas en el MST. ¿Por qué esta finalización temprana no afecta al resultado del algoritmo? ¿Hay alguna otra forma de implementar la misma optimización, utilizando la estructura de datos de conjuntos disjuntos para unión-buscar?

### Ejercicio 4.3.2.2\*

¿Eres capaz de resolver el problema del MST *más rápido* que en  $O(E \log V)$ , si tienes la garantía de que el grafo de entrada tiene pesos en sus aristas que se encuentran dentro de un rango pequeño de enteros  $[0 \dots 100]$ ? ¿Es la mejora de velocidad significativa?

### 4.3.3 Algoritmo de Prim

El algoritmo de Robert Clay *Prim* (o de Vojtěch Jarník), comienza tomando un vértice de inicio (por comodidad, elegiremos el 0), lo etiqueta como ‘tomado’, y añade una pareja de datos a una cola de prioridad: el peso  $w$  y el otro extremo  $u$  de la arista  $0 \rightarrow u$ , que todavía no ha sido tomado. Estas parejas se ordenan en la cola de prioridad, en base a su peso creciente y, en caso de empate, por vértice creciente. Entonces, el algoritmo de Prim selecciona *vorazmente* la pareja  $(w, u)$  del principio de la cola de prioridad, que tiene el menor peso  $w$ , si el extremo de esta arista, que es  $u$ , no ha sido tomado antes. Esto se hace para evitar ciclos. Si esta pareja  $(w, u)$  es válida, entonces el peso  $w$  se suma al coste del MST,  $u$  se marca como tomado, y la pareja  $(w', v)$  de cada arista  $u \rightarrow v$  de peso  $w'$ , que sea incidente a  $u$ , se añade a la cola de prioridad,

si  $v$  no ha sido tomado antes. Este proceso se repite hasta que se vacía la cola de prioridad. La longitud del código es similar a la del algoritmo de Kruskal, y también se ejecuta en  $O(\text{procesar cada arista una vez} \times \text{coste de añadir/quitar de la cola}) = O(E \times \log E) = O(E \log V)$ .

```

1 vi taken; // etiqueta booleana global para evitar ciclos
2 priority_queue<ii> pq; // cola de prioridad para elegir aristas más cortas
3 // nota: priority_queue de la STL de C++ es un montículo máximo por defecto
4 void process(int vtx) { // así, usamos signo negativo para invertir orden
5 taken[vtx] = 1;
6 for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
7 ii v = AdjList[vtx][j];
8 if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
9 } } // ordenar por peso (inc) y por id (inc)
10
11 // dentro de int main(), grafo almacenado como AdjList y pq está vacía
12 taken.assign(V, 0); // no tomamos ningún vértice al principio
13 process(0); // tomamos vértice 0 y procesamos todas sus aristas incidentes
14 int mst_cost = 0, num_taken = 0; // ninguna arista tomada en este punto
15 while (!pq.empty() && num_taken < V-1) { // hasta que se tomen V-1 aristas
16 ii front = pq.top(); pq.pop();
17 int u = -front.second, w = -front.first; // peso e id negativos
18 if (!taken[u]) // todavía no hemos conectado este vértice
19 num_taken++; // se toma una arista más como parte del MST
20 mst_cost += w; // se añade el peso de esta arista al MST
21 process(u); // tomar u, procesar todas las aristas incidentes a u
22 } } // cada arista está en pq una sola vez
23 printf("MST cost = %d (Prim's)\n", mst_cost);

```

La figura 4.12 muestra la ejecución, paso a paso, del algoritmo de Prim sobre el mismo grafo mostrado en la parte izquierda de la figura 4.10. Compáralo con la figura 4.11, para estudiar las similitudes y diferencias entre ambos algoritmos.

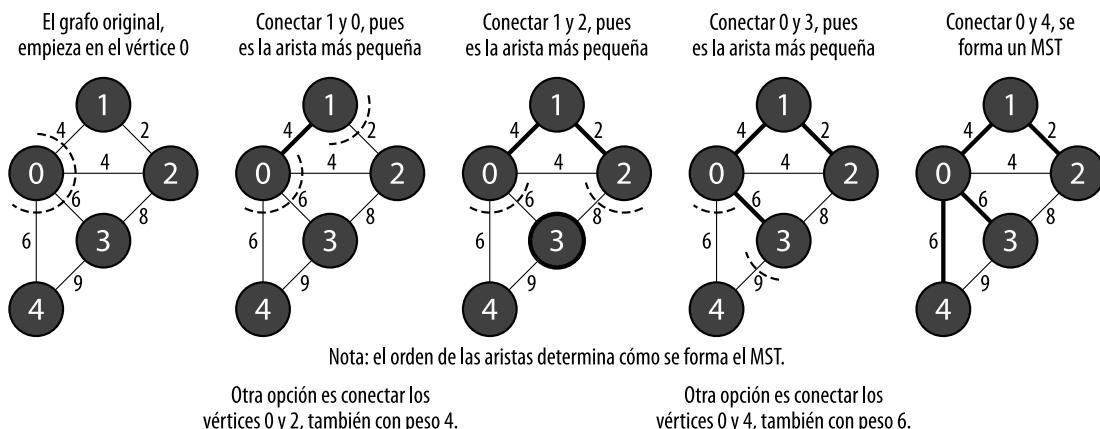


Figura 4.12: Animación del algoritmo de Prim en el grafo de la izquierda de la figura 4.10



visualgo.net/mst



ch4\_03\_kruskal\_prim.cpp



ch4\_03\_kruskal\_prim.java

#### 4.3.4 Otras aplicaciones

Hay variantes interesantes del problema del MST. En esta sección conoceremos algunas de ellas.

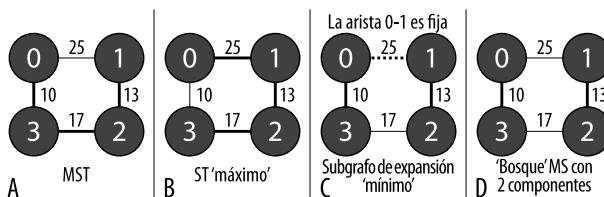


Figura 4.13: De izquierda a derecha: MST, ST ‘máximo’, SS ‘mínimo’, ‘bosque’ MS

#### Árbol de expansión ‘máximo’

Esta es una variante sencilla, donde buscamos el árbol de expansión máximo, en vez del mínimo, por ejemplo: UVa 1234 - RACING (cuidado, porque este problema está redactado de forma que no parece un problema de MST). En la figura 4.13.B, vemos un ejemplo de un árbol de expansión máximo. Compáralo con el MST correspondiente (figura 4.13.A).

La solución a esta variante es muy simple: basta modificar un poco el algoritmo de Kruskal y, ahora, ordenar las aristas en base a su peso *no creciente*.

#### Subgrafo de expansión ‘mínimo’

En esta variante no empezamos con una hoja en blanco. Algunas de las aristas del grafo dado ya han sido determinadas y deben tomarse como parte de la solución, por ejemplo UVa 10147 - Highways. Estas aristas predeterminadas pueden, de entrada, formar un *no árbol*. Nuestra tarea consiste en continuar seleccionando las aristas restantes (si es necesario), para llegar a un grafo conexo con el menor coste posible. El subgrafo de expansión resultante puede no ser un árbol e, incluso si lo es, puede no ser un MST. Por eso hemos escrito el término ‘mínimo’ entre comillas y utilizado ‘subgrafo’ en vez de ‘árbol’. En la figura 4.13.C, podemos ver un ejemplo donde la arista 0-1 ya existe. El MST correspondiente es  $10+13+17 = 40$ , que omite la arista 0-1 (figura 4.13.A). Sin embargo, la solución para este ejemplo debe ser  $(25)+10+13 = 48$ , utilizando la arista 0-1.

La solución a esta variante es sencilla. Después de tener en cuenta todas las aristas predeterminadas y sus costes, continuamos ejecutando el algoritmo de Kruskal sobre las aristas libres, hasta que tengamos un subgrafo de expansión (o árbol de expansión).

#### ‘Bosque de expansión’ mínimo

En esta variante, queremos formar un bosque de  $K$  componentes conexos ( $K$  subárboles) de la forma menos costosa, donde conocemos  $K$  de antemano por el enunciado del problema,

por ejemplo, UVa 10369 - Arctic Networks. En la figura 4.13.A, observamos que el MST de este grafo es  $10+13+17 = 40$ . Pero si estamos satisfechos con un bosque de expansión de dos componentes conexos, la solución es, simplemente,  $10+13 = 23$ , como muestra la figura 4.13.D. Esto es, omitimos la arista 2-3, con peso 17, que convertiría, si la utilizásemos, estos dos componentes en un árbol de expansión.

Obtener el bosque de expansión mínimo es sencillo. Se ejecuta el algoritmo de Kruskal con normalidad pero, en cuanto el número de componentes conexos iguale al número predeterminado  $K$ , podemos finalizar la ejecución.

### Segundo mejor árbol de expansión

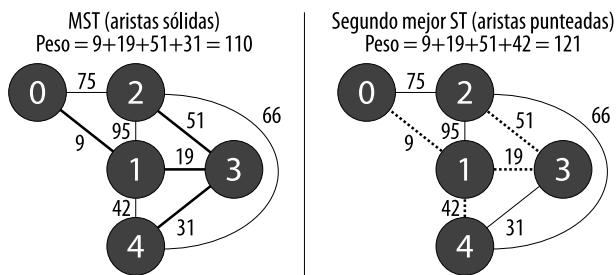


Figura 4.14: Segundo mejor árbol de expansión (de UVa 10600 [49])

Las soluciones alternativas son, en ocasiones, importantes. En el contexto de la búsqueda del MST, quizás no queramos solo el propio MST, sino también el segundo mejor árbol de expansión, en caso de que ese MST no nos sirva, por ejemplo, en UVa 10600 - ACM contest and blackout. La figura 4.14 muestra el MST (a la izquierda) y el segundo mejor árbol de expansión (a la derecha). Podemos ver que el segundo mejor árbol de expansión es, en realidad, el MST con solo dos aristas diferentes, es decir, se quita una arista del MST y se añade una arista cordal<sup>5</sup> al mismo. Aquí se elimina la arista 3-4 y se añade la 1-4.

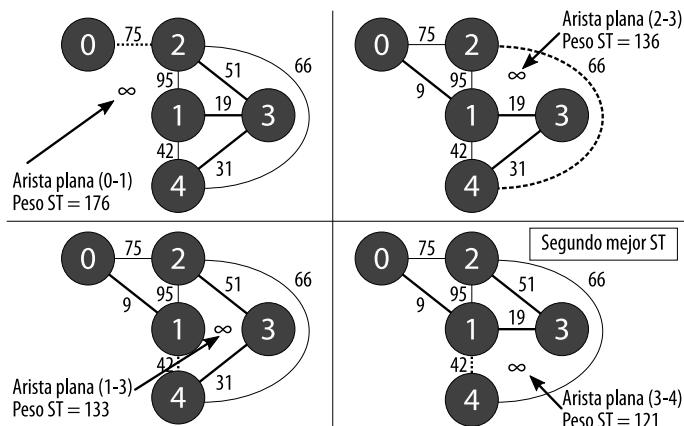


Figura 4.15: Búsqueda del segundo mejor árbol de expansión de un MST

<sup>5</sup>Una arista cordal, se define como una arista en un grafo  $G$  que no forma parte del MST de  $G$ .

Una solución para esta variante es una modificación del algoritmo de Kruskal: ordenar las aristas en  $O(E \log E) = O(E \log V)$ , después encontrar el MST utilizando el algoritmo de Kruskal en  $O(E)$ . A continuación, para cada arista del MST (tiene un máximo de  $V-1$  aristas), marcarla temporalmente para que no pueda ser elegida, entonces intentar encontrar otra vez el MST en  $O(E)$ , pero ahora *excluyendo* la arista marcada. En este punto, no es necesario volver a ordenar las aristas. El mejor árbol de expansión que encontraremos en este proceso, será el segundo mejor para el grafo. La figura 4.15 muestra este algoritmo sobre el grafo dado. Globalmente, este algoritmo se ejecuta en  $O(\text{ordenar las aristas una vez} + \text{encontrar el MST original} + \text{encontrar el segundo mejor árbol de expansión}) = O(E \log V + E + VE) = O(VE)$ .

### Minimax (y maximin)

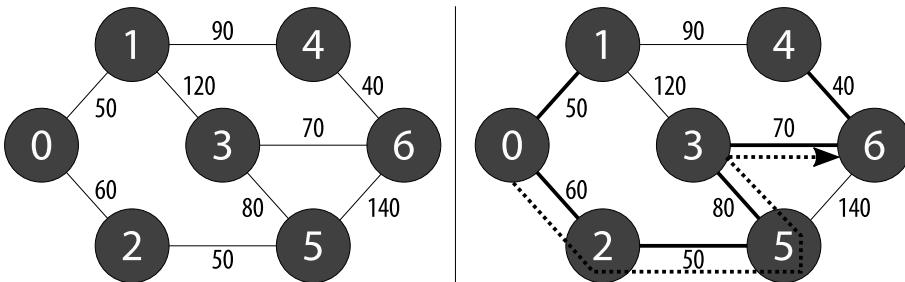


Figura 4.16: Minimax (UVa 10048 [49])

El problema del camino *minimax* consiste en encontrar el mínimo del peso máximo de las aristas, de entre todos los posibles caminos entre dos vértice  $i$  y  $j$ . El coste de un camino de  $i$  a  $j$  viene determinado por el peso de aristas máximo a lo largo de ese camino. De entre todos los caminos posibles entre  $i$  y  $j$ , elegir el que tenga el mínimo peso de aristas máximo. El problema inverso, llamado *maximin*, se define de forma similar.

El problema del camino *minimax* entre los vértices  $i$  y  $j$ , se puede resolver modelándolo como un problema de MST. Partiendo del razonamiento de que el problema prefiere un camino con pesos de aristas individuales bajos, aunque el camino sea más largo en términos de número de vértices/aristas implicados, obtener el MST (usando los algoritmos de Kruskal o Prim) del grafo ponderado dado es un paso correcto. El MST es conexo, lo que asegura que siempre hay un camino entre cualquier par de vértices. La solución del camino *minimax* es, por lo tanto, el peso de aristas máximo a lo largo del único camino entre los vértices  $i$  y  $j$  de este MST.

La complejidad de tiempo total es  $O(\text{construir el MST} + \text{un recorrido del árbol resultante})$ . Como  $E = V - 1$  en un árbol, cualquier recorrido del mismo supone solo  $O(V)$ . Así, la complejidad de esta técnica es  $O(E \log V + V) = O(E \log V)$ .

La parte izquierda de la figura 4.16 es un caso de prueba de ejemplo del problema UVa 10048 - Audiophobia. Tenemos un grafo con 7 vértices y 9 aristas. Las 6 aristas elegidas del MST aparecen como líneas más gruesas en la parte derecha de la figura 4.16. Si ahora se nos pide encontrar el camino *minimax* entre los vértices 0 y 6 de la parte derecha de figura 4.16, nos basta con recorrer el MST desde el vértice 0 al 6. Solo hay un camino para ello: 0-2-5-3-6. El peso máximo de las aristas que encontraremos durante el recorrido, es el coste *minimax* solicitado: 80 (debido a la arista 5-3).

### Ejercicio 4.3.4.1

Resuelve las cinco variantes del problema del MST mencionadas, utilizando el algoritmo de Prim. ¿Qué variantes no son adecuadas para este algoritmo?

### Ejercicio 4.3.4.2\*

Existen mejores soluciones para el problema del segundo mejor árbol de expansión. Resuelve el problema con una solución que mejore a  $O(VE)$ . Consejo: puedes usar el ancestro común mínimo (LCA) o conjuntos disjuntos para unión-buscar.

## Comentarios sobre MST en concursos de programación

En nuestra opinión, para resolver la mayoría de los problemas de MST, en los concursos de programación actuales, podemos depender exclusivamente del algoritmo de Kruskal y olvidarnos del de Prim (y otros). Es fácil de entender y se relaciona bien con la estructura de datos de conjuntos disjuntos para unión-buscar (ver la sección 2.4.2) que utilizamos para detectar ciclos. Sin embargo, como nos encanta la variedad, hemos incluido la referencia al otro algoritmo popular para los MST, el de Prim.

La intención (y el uso más común) del algoritmo de Kruskal (o el de Prim), es la de resolver el problema del árbol de expansión mínimo (UVa 908, 1174, 1208, 11631), pero también sirve para la sencilla variante del árbol de expansión ‘máximo’ (UVa 1234, 10842). Prácticamente todos los problemas de MST de los concursos de programación piden solo el coste MST *único*, y no el propio MST. Esto se debe a que puede haber diferentes MST con el mismo coste mínimo y, normalmente, es complicado escribir un programa de comprobación especial para evaluar las respuestas que no son únicas.

Las otras variantes del MST tratadas en este libro, como el subgrafo de expansión ‘mínimo’ (UVa 10147, 10397), el ‘bosque de expansión’ mínimo (UVa 1216, 10369), el segundo mejor árbol de expansión (UVa 10462, 10600) o *minimax/maximin* (UVa 534, 544, 10048, 10099) son, ciertamente, muy poco frecuentes.

Hoy día, la tendencia general, en relación a los problemas de MST, es que los autores los escriban de forma que no esté claro que el problema es, en realidad, un problema de MST (por ejemplo UVa 1013, 1216, 1234, 1235, 1265, 10457). Sin embargo, una vez que los concursantes se dan cuenta de ese detalle, el problema se puede convertir en ‘fácil’.

También hay problemas de MST más complejos, que pueden necesitar algoritmos más sofisticados, como el problema de arborescencia, el árbol de Steiner, el MST con limitación de grados,  $k$ -MST, etc.

## Ejercicios de programación

Ejercicios de programación relativos al árbol de expansión mínimo:

### Generales

- |                                            |                                                                                                                                  |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00908 - Re-connecting ...           | (problema básico de MST)                                                                                                         |
| 2. UVa 01174 - IP-TV                       | (LA 3988 - SouthWesternEurope07; MST clásico; basta hacer un mapa de nombres de ciudades a índices)<br>(LA 3171 - Manila06; MST) |
| 3. UVa 01208 - Oreon                       | (LA 4138 - Jakarta08; el problema subyacente es MST)                                                                             |
| 4. UVa 01235 - Anti Brute Force Lock       | (problema MST directo)                                                                                                           |
| 5. UVa 10034 - Freckles                    | (dividir la salida para aristas cortas frente a largas)                                                                          |
| <b>6. UVa 11228 - Transportation ... *</b> | (peso de (todas las aristas - todas las aristas MST))                                                                            |
| <b>7. UVa 11631 - Dark Roads *</b>         | (mostrar 'Impossible' si el grafo no es conexo después de ejecutar MST)                                                          |
| 8. UVa 11710 - Expensive Subway            | (mantener coste en cada actualización)                                                                                           |
| 9. UVa 11733 - Airports                    | (sumar los pesos de arista de las cuerdas)                                                                                       |
| <b>10. UVa 11747 - Heavy Cycle Edges *</b> | (buscar el peso de la última arista añadida al MST con Kruskal)                                                                  |
| 11. UVa 11857 - Driving Range              | (usar MST incremental eficiente)                                                                                                 |
| 12. IOI 2003 - Trail Maintenance           |                                                                                                                                  |

### Variantes

- |                                              |                                                                                                                                               |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00534 - Frogger                       | ( <i>minimax</i> ; también se resuelve con Floyd Warshall)                                                                                    |
| 2. UVa 00544 - Heavy Cargo                   | ( <i>maximin</i> ; también se resuelve con Floyd Warshall)                                                                                    |
| 3. UVa 01160 - X-Plosives                    | (LA 3644 - SouthWesternEurope06; contar el número de aristas no tomadas por Kruskal)<br>(LA 3678 - Kaohsiung06; 'bosque de expansión' mínimo) |
| 4. UVa 01216 - The Bug Sensor Problem        | (LA 4110 - Singapore07; bosque de expansión 'máximo')                                                                                         |
| 5. UVa 01234 - RACING                        | (problema clásico de camino <i>minimax</i> ; también con Floyd Warshall)                                                                      |
| <b>6. UVa 10048 - Audiophobia *</b>          | ( <i>maximin</i> ; también se resuelve con Floyd Warshall)                                                                                    |
| 7. UVa 10099 - Tourist Guide                 | (subgrafo de expansión 'mínimo')                                                                                                              |
| 8. UVa 10147 - Highways                      | ('bosque' de expansión mínimo)                                                                                                                |
| <b>9. UVa 10369 - Arctic Networks *</b>      | (subgrafo de expansión 'mínimo')                                                                                                              |
| 10. UVa 10397 - Connect the Campus           | (segundo mejor árbol de expansión)                                                                                                            |
| 11. UVa 10462 - Is There A Second ...        | (segundo mejor árbol de expansión)                                                                                                            |
| <b>12. UVa 10600 - ACM Contest and ... *</b> | (buscar arista de peso mínimo en árbol de expansión 'máximo')                                                                                 |
| 13. UVa 10842 - Traffic Flow                 |                                                                                                                                               |

## Perfiles de los inventores de algoritmos

**Robert Endre Tarjan** (nacido en 1948) es un científico de la computación estadounidense. Es el descubridor de varios algoritmos de grafos importantes. Su mayor aportación al campo de la programación competitiva es el algoritmo de **búsqueda de componentes fuertemente conexos** en un grafo dirigido y el algoritmo de búsqueda de **puntos de articulación y puentes** en un grafo no dirigido (sección 4.2, junto con otras variantes de la DFS inventadas por él y sus colaboradores [63]). También ha inventado el algoritmo del **ancestro común mínimo fuera de línea**, la estructura de datos de **árbol biselado**, y ha analizado la complejidad de tiempo de la estructura de datos de **conjuntos disjuntos para unión-buscar** (sección 2.4.2).

**John Edward Hopcroft** (nacido en 1939) es un científico de la computación estadounidense. Es profesor de ciencias de la computación en la Cornell University. Hopcroft fue galardonado en 1986 con el Premio Turing, el reconocimiento más prestigioso en su campo y conocido frecuentemente como el ‘Nobel de la informática’, *ex aequo* con Robert Endre Tarjan, por sus logros fundamentales en el diseño y análisis de algoritmos y estructuras de datos. Junto a su trabajo con Tarjan en grafos planos (y otros algoritmos de grafos, como la **búsqueda de puntos de articulación/puentes utilizando DFS**), también es conocido por el **algoritmo de Hopcroft-Karp**, para la búsqueda de coincidencias en grafos bipartitos, inventado junto a Richard Manning Karp [28] (ver la sección 9.12).

**Joseph Bernard Kruskal, Jr.** (1928-2010) fue un científico de la computación estadounidense. Su trabajo más conocido relativo a la programación competitiva es el **algoritmo de Kruskal**, que calcula el árbol de expansión mínimo (MST) de un grafo ponderado. Tiene aplicaciones interesantes en la construcción y para *establecer precios* de redes de comunicación.

**Robert Clay Prim** (nacido en 1921) es un matemático y científico de la computación estadounidense. En 1957, en Bell Laboratories, desarrolló el algoritmo de Prim para resolver el problema del MST. Prim conoció a Kruskal, pues trabajaron juntos en los Bell Laboratories. El algoritmo de Prim ya había sido descubierto por Vojtěch Jarník en 1930, y fue redescubierto de forma independiente por Prim. Por eso, en ocasiones, el algoritmo de Prim es conocido como el algoritmo de Jarník-Prim.

**Vojtěch Jarník** (1897-1970) fue un matemático checo. Desarrolló el algoritmo de grafos conocido ahora como algoritmo de Prim. Hoy día, gracias a la rápida y extensa difusión de las publicaciones científicas, el algoritmo de Prim se habría atribuido correctamente a Jarník.

**Edsger Wybe Dijkstra** (1930-2002) fue un científico de la computación holandés. Una de sus famosas contribuciones a la ciencia de la computación es el algoritmo del camino más corto, o **algoritmo de Dijkstra** [10]. No le gustaba la expresión ‘GOTO’ y supuso una influencia fundamental en la desaparición generalizada de la misma y su sustitución por construcciones de control estructuradas. Una de sus frases más famosas: “si tienes dos o más, usa **for**”.

**Richard Ernest Bellman** (1920-1984) fue un matemático aplicado estadounidense. Además de inventar el **algoritmo de Bellman Ford**, para la búsqueda de los caminos más cortos en grafos que tienen aristas ponderadas de peso negativo (y, posiblemente, ciclos de peso negativo), Richard Bellman es especialmente conocido por su invención de la técnica de la *programación dinámica* en 1953.

**Lester Randolph Ford, Jr.** (nacido en 1927) es un matemático estadounidense especializado en problemas de flujo de red. El artículo de 1956 de Ford, junto a Fulkerson, sobre el problema del flujo máximo, y el **método Ford Fulkerson** para resolverlo, estableció el teorema del flujo máximo corte mínimo.

**Delbert Ray Fulkerson** (1924-1976) fue un matemático estadounidense, coautor del **método Ford Fulkerson**, un algoritmo para resolver el problema del flujo máximo en redes. En 1956 publicó su artículo sobre el método Ford Fulkerson, junto a Lester R. Ford.

## 4.4 Caminos más cortos de origen único (SSSP)

### 4.4.1 Introducción y motivación

Problema de referencia: dado un grafo *ponderado*  $G$  y un vértice de inicio  $s$ , ¿cuáles son los *caminos más cortos* desde  $s$  a *todos los otros vértices* de  $G$ ?

Este problema se denomina de *caminos más cortos de origen único*<sup>6</sup> (SSSP) en un *grafo ponderado*. Es un problema clásico dentro de la teoría de grafos y tiene muchas aplicaciones en la vida real. Por ejemplo, podemos modelar la ciudad en la que vivimos como un grafo. Los vértices son los cruces de calles. Las aristas son las calles. El tiempo que toma recorrer una calle es el peso de la arista. Te encuentras en un cruce de calles, ¿cuál es el tiempo mínimo imprescindible para llegar a otro cruce determinado?

Hay algoritmos eficientes para resolver este problema. Si el grafo no es ponderado (o todas las aristas tienen peso igual o constante), podemos utilizar el muy eficiente algoritmo BFS en  $O(V+E)$ , visto en la sección 4.2.2. Para un grafo ponderado general, la BFS no funciona correctamente y debemos utilizar algoritmos como el de Dijkstra en  $O((V+E)\log V)$  o el de Bellman Ford en  $O(VE)$ . A continuación, veremos estos algoritmos.

#### Ejercicio 4.4.1.1\*

Demuestra que el camino más corto entre dos vértices  $i$  y  $j$  en un grafo  $G$ , que no tiene ciclos de peso negativo, debe ser un camino *sencillo* (acíclico).

#### Ejercicio 4.4.1.2\*

Demuestra que los subcaminos de los caminos más cortos entre  $u$  y  $v$  son, a su vez, los caminos más cortos.

### 4.4.2 SSSP en un grafo no ponderado

Volvamos a tratar de nuevo lo visto en la sección 4.2.2. El hecho de que la BFS visite los vértices de un grafo capa a capa, desde un vértice origen (ver la figura 4.3), la convierte en la elección natural para resolver los problemas de los SSSP en grafos *no ponderados*. En un grafo no ponderado, la distancia entre dos vértices vecinos conectados por una arista es, simplemente, una unidad. Por lo tanto, el número de capas para un vértice que hemos visto en la sección 4.2.2 es, precisamente, la longitud del camino más corto desde el origen hasta ese vértice. Por ejemplo, en la figura 4.3, el camino más corto entre el vértice 5 y el vértice 7 es 4, ya que el 7 está en la cuarta capa de la secuencia de visitas de la BFS, empezando en el vértice 5.

<sup>6</sup>Este problema SSSP genérico se puede utilizar también para resolver: 1) el problema del camino más corto del par único (*u* origen y destino únicos) y 2) el problema del camino más corto de destino único, en el que se invierte el papel de los vértices de origen y destino.

Algunos problemas de programación nos piden *reconstruir* el camino más corto real, no solo informar de su longitud. Por ejemplo, en la figura 4.3, el camino más corto entre 5 y 7 es  $5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$ . Esta reconstrucción es sencilla si almacenamos el árbol de expansión del camino más corto (en realidad, la BFS)<sup>7</sup>. Esto se puede lograr de forma sencilla, utilizando un vector de enteros  $\text{vi } p$ . Cada vértice  $v$  recuerda a su padre  $u$  ( $p[v] = u$ ) en el árbol de expansión del camino más corto. En nuestro ejemplo, el vértice 7 recuerda que 3 es su padre, el vértice 3 recuerda que lo es 2, el vértice 2 que lo es 1, el vértice 1 que lo es 5 (el origen). Para reconstruir el camino más corto, podemos hacer una recursión fácil desde el último vértice 7, hasta que encontremos el vértice origen 5. El código BFS modificado (ver los comentarios) es relativamente simple:

```

1 void printPath(int u) { // extraer información de 'vi p'
2 if (u == s) { printf("%d", s); return; } // caso base, en el origen s
3 printPath(p[u]); // recursivo: para el formato de salida: s -> ... -> t
4 printf(" %d", u); }
5
6 // dentro de int main()
7 vi dist(V, INF); dist[s] = 0; // la distancia del origen s a s es 0
8 queue<int> q; q.push(s);
9 vi p; // añadido: el vector de predecesores/padres
10 while (!q.empty()) {
11 int u = q.front(); q.pop();
12 for (int j = 0; j < (int)AdjList[u].size(); j++) {
13 ii v = AdjList[u][j];
14 if (dist[v.first] == INF) {
15 dist[v.first] = dist[u]+1;
16 p[v.first] = u; // añadido: el padre del vértice v.first es u
17 q.push(v.first);
18 } } }
19 printPath(t), printf("\n"); // añadido: llamar printPath desde vértice t

```



ch4\_04\_bfs.cpp



ch4\_04\_bfs.java

Nos gustaría poner de relieve que, en los concursos de programación más recientes, los problemas que implican BFS ya no están escritos directamente como problemas de SSSP, sino de formas mucho más creativas. Algunas variantes posibles incluyen BFS en grafos implícitos (rejilla bidimensional, UVa 10653, o tridimensional, UVa 532), BFS indicando expresamente el camino más corto (UVa 11049), BFS en un grafo con algunos vértices bloqueados (UVa 10977), BFS desde orígenes múltiples (UVa 11101, 11624), BFS con destino único (que se resuelve invirtiendo los roles del origen y el destino, UVa 11513), BFS con estados no triviales (UVa 10150) y otros que mencionamos en la sección 8.2.3, etc. Como existen muchas variantes interesantes de la BFS, recomendamos al lector que trate de resolver la mayor cantidad posible de los ejercicios de programación correspondientes a esta sección.

<sup>7</sup>La reconstrucción del camino más corto no aparece en las dos siguientes subsecciones (Dijkstra/Bellman Ford), pero la idea es la misma que la que mostramos aquí (y en la reconstrucción de la solución de DP en la sección 3.5.1).

### Ejercicio 4.4.2.1

Podemos ejecutar la BFS desde más de un origen. Denominamos a esta variante el problema de los caminos más cortos de origen múltiple (MSSP) en un grafo no ponderado. Intenta resolver los problemas UVa 11101 y 11624, para hacerte una idea de MSSP en grafos no ponderados. Una solución ingenua es llamar a BFS varias veces. Si hay  $k$  orígenes posibles, esa solución se ejecutará en  $O(k \times (V+E))$ . ¿Puedes mejorarlo?

### Ejercicio 4.4.2.2

Sugiere una mejora sencilla al código BFS dado, si se te pide que resuelvas el problema del camino más corto de origen único y *destino único*, en un grafo no ponderado. Así es, se proporciona tanto el vértice de origen como el de destino.

### Ejercicio 4.4.2.3

Explica la razón por la que podemos utilizar BFS para resolver un problema SSSP en un grafo ponderado donde *todas las aristas* tienen el mismo peso  $C$ .

### Ejercicio 4.4.2.4\*

Dado un mapa en rejilla de tamaño  $R \times C$ , como el que se muestra a continuación, determina el camino más corto desde cualquier celda etiquetada como ‘A’ hasta cualquier celda etiquetada como ‘B’. Solo te puedes desplazar por las celdas etiquetadas como ‘.’ en las direcciones NESO (contadas como *una* unidad) y las celdas etiquetadas con el alfabeto ‘A’-‘Z’ (contadas como *cero* unidades). ¿Puedes resolverlo en  $O(R \times C)$ ?

```
.....CCCC. // La respuesta en este caso es 13 unidades
AAAAA.....CCCC. //Solución: al este desde la A más a la dcha.
AAAAAA.AAA.....CCCC. // hasta la C más a la izda. en esta fila,
AAAAAAAAA....###....CCCC. // sur desde la C más a la dcha. de esta fila
AAAAAAAAA..... // hacia abajo
.....DD.....BB // hasta
.....DD.....BB // la B más a la izda. en esta fila
```

#### 4.4.3 SSSP en un grafo ponderado

Si el grafo dado es *ponderado*, la BFS no funciona. Esto se debe a que puede haber rutas ‘más largas’ (en número de vértices y aristas implicados), pero con un peso total menor que los ‘más cortos’, encontrados por la BFS. Por ejemplo, en la figura 4.17, el camino más corto desde el vértice de origen 2 al vértice 3, no es a través de la arista directa  $2 \rightarrow 3$ , de peso 7 y que encontraría la BFS, sino haciendo el ‘desvío’  $2 \rightarrow 1 \rightarrow 3$ , con un peso total  $2 + 3 = 5$ , menor.

Para resolver el problema SSSP en un grafo ponderado, utilizamos el algoritmo voraz de Edsger Wybe Dijkstra. Hay muchas formas de implementar este algoritmo clásico. De hecho, el artículo original de Dijkstra que describe este algoritmo [10], no menciona una implementación específica. Muchos otros investigadores han propuesto variantes de implementación basadas en el trabajo original de Dijkstra. Aquí utilizaremos una de las implementaciones más sencillas, que emplea la *función integrada priority\_queue* de la STL de C++ (o *PriorityQueue* de Java). Esto se debe a nuestra intención de mantener *bajo mínimos* la longitud del código, algo muy necesario en la programación competitiva.

Esta variante de Dijkstra mantiene una cola de **prioridad** llamada *pq*, que almacena información de parejas de vértices. Los primer y segundo elementos de las parejas, son la distancia del vértice desde el origen y el número de vértice, respectivamente. La *pq* se ordena en base a la *distancia creciente* desde el origen y, si hay una igualdad, por el número de vértice. Esta implementación es diferente a otras, que utilizan la característica de montículos binarios, que no está soportada por la biblioteca integrada<sup>8</sup>.

Incialmente, esta *pq* solo contiene un elemento: el caso base  $(0, s)$ , que es verdadero para el vértice de origen. Después, nuestra variante repite el siguiente proceso, hasta que *pq* se vacía: toma vorazmente parejas de datos de vértices  $(d, u)$  del principio de *pq*. Si la distancia *u* desde el origen, registrada en *d*, es mayor que *dist[u]*, ignora *u*; en caso contrario, procesa *u*. La razón de esta comprobación se muestra a continuación.

Cuando este algoritmo procesa *u*, intenta relajar<sup>9</sup> todos los vecinos *v* de *u*. Cada vez que relaja una arista  $u \rightarrow v$ , insertará una pareja (distancia nueva/más corta a *v* desde el origen, *v*) en *pq* y dejará la pareja inferior (distancia vieja/más larga a *v* desde el origen, *v*) dentro de *pq*. Esto se denomina ‘eliminación perezosa’ y provoca que haya *más de una copia* del mismo vértice en *pq*, con *diferentes distancias* desde el origen. Por eso tenemos que procesar solo la *primera* pareja de información de vértices extraída de la cola, que es la que tiene la distancia correcta/más corta (las otras copias tendrán distancias no actualizadas/más largas). El breve código se incluye a continuación y resulta muy similar a los códigos de BFS y Prim, de las secciones 4.2.2 y 4.3.3.

```
1 vi dist(V, INF); dist[s] = 0; // INF = 1000M para evitar desbordamientos
2 priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
3 while (!pq.empty()) { // bucle principal
4 ii front = pq.top(); pq.pop(); // voraz: el vértice mínimo no visitado
5 int d = front.first, u = front.second;
6 if (d > dist[u]) continue; // esta comprobación es muy importante
```

<sup>8</sup>La implementación habitual de Dijkstra (ver, por ejemplo, [7, 38, 8]) requiere la operación *heapDecreaseKey* en una estructura de datos de montículo binario, que no está incluida en la cola de prioridad de la STL de C++ o en Java. La implementación de Dijkstra tratada en esta sección utiliza solo dos operaciones básicas de la cola de prioridad: *enqueue* y *dequeue*.

<sup>9</sup>La operación *relax(u, v, w\_u\_v)* establece *dist[v] = min(dist[v], dist[u] + w\_u\_v)*.

```

7 for (int j = 0; j < (int)AdjList[u].size(); j++) {
8 ii v = AdjList[u][j]; // todas las aristas salientes de u
9 if (dist[u] + v.second < dist[v.first]) {
10 dist[v.first] = dist[u] + v.second; // operación de relajación
11 pq.push(ii(dist[v.first], v.first));
12 } } } // esta variante puede causar elementos duplicados en la cola

```



En la figura 4.17 mostramos un ejemplo, paso a paso, de ejecución de nuestra variante de implementación del algoritmo de Dijkstra en un grafo pequeño con  $s = 2$ . Mira detenidamente el contenido de  $\text{pq}$  en cada paso.

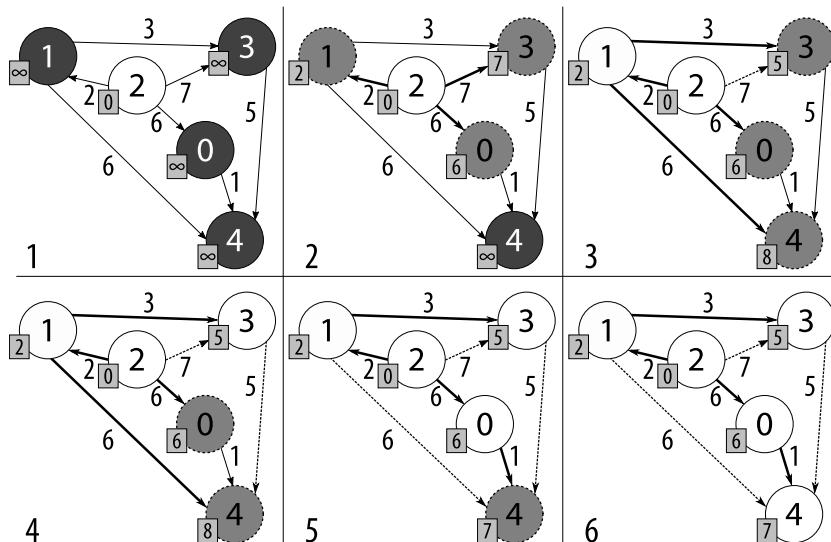


Figura 4.17: Animación de Dijkstra en un grafo ponderado (de UVa 341 [49])

1. Al principio, solo  $\text{dist}[s] = \text{dist}[2] = 0$ , la priority\\_queue  $\text{pq}$  es  $\{(0,2)\}$ .
2. La pareja de datos del vértice  $(0,2)$  sale de la cola  $\text{pq}$ . Se relajan las aristas incidentes al vértice 2, para obtener  $\text{dist}[0] = 6$ ,  $\text{dist}[1] = 2$  y  $\text{dist}[3] = 7$ . Ahora  $\text{pq}$  contiene  $\{(2,1), (6,0), (7,3)\}$ .
3. Entre las parejas sin procesar en  $\text{pq}$ ,  $(2,1)$  está al principio de la cola. Sale  $(2,1)$  de la cola y se relajan las aristas incidentes al vértice 1 para obtener  $\text{dist}[3] = \min(\text{dist}[3], \text{dist}[1] + \text{weight}(1,3)) = \min(7, 2+3) = 5$  y  $\text{dist}[4] = 8$ . Ahora  $\text{pq}$  contiene  $\{(5,3), (6,0), (7,3), (8,4)\}$ . Vemos que tenemos 2 entradas del vértice 3 en nuestra  $\text{pq}$ , con distancia creciente desde el origen  $s$ . No eliminamos inmediatamente la pareja inferior  $(7,3)$  de la cola, y delegamos en una iteración posterior de nuestra variante de Dijkstra la elección de la pareja correcta con menor distancia,  $(5,3)$ . Esta técnica se llama ‘eliminación perezosa’.

4. Sale  $(5,3)$  de la cola e intentamos `relax(3,4,5)`, es decir,  $5+5 = 10$ . Pero  $\text{dist}[4] = 8$  (del camino 2-1-4), así que  $\text{dist}[4]$  permanece sin cambios. Ahora  $\text{pq}$  contiene  $\{(6,0), (7,3), (8,4)\}$ .
5. Sale  $(6,0)$  de la cola y ejecutamos `relax(0,4,1)`, haciendo que  $\text{dist}[4] = 7$  (el camino más corto de 2 a 4 pasa a ser 2-0-4, en vez de 2-1-4).  $\text{pq}$  contiene  $\{(7,3), (7,4), (8,4)\}$ , con 2 entradas del vértice 4. Es otro caso de ‘eliminación perezosa’.
6. Ahora podemos ignorar  $(7,3)$ , porque sabemos que su  $d > \text{dist}[3]$  (es decir,  $7 > 5$ ). En esta sexta iteración es cuando se produce la eliminación real de la pareja inferior  $(7,3)$ , en lugar de en la tercera. Al diferir la operación hasta la sexta iteración, la pareja inferior  $(7,3)$  se encuentra situada en la posición más fácil para la eliminación estándar en  $O(\log n)$  en el montículo mínimo: la raíz del montículo, es decir, al principio de la cola de prioridad.
7. Se procesa  $(7,4)$  como antes, pero no cambia nada. Ahora  $\text{pq}$  solo contiene  $\{(8,4)\}$ .
8. Por último, se vuelve a ignorar  $(8,4)$ , ya que su  $d > \text{dist}[4]$  (es decir,  $8 > 7$ ). Esta implementación del algoritmo de Dijkstra se detiene aquí, ya que  $\text{pq}$  ahora está vacía.

#### Aplicación de ejemplo: UVa 11367 - Full Tank?

Enunciado resumido: dada la *longitud* de un grafo conexo ponderado, que almacena la longitud de las carreteras entre  $E$  pares de ciudades  $i$  y  $j$  ( $1 \leq V \leq 1000, 0 \leq E \leq 10000$ ), el precio  $p[i]$  del combustible en cada ciudad  $i$  y la capacidad del depósito  $c$  de un coche ( $1 \leq c \leq 100$ ), determinar el coste del viaje más barato desde la ciudad de origen  $s$  a la de destino  $e$ , utilizando un coche con capacidad de combustible  $c$ . Todos los coches utilizan una unidad de combustible por cada unidad de distancia y comienzan con el depósito vacío.

Con este problema ,pretendemos tratar la importancia del *modelado de grafos*. El grafo proporcionado explícitamente en este problema es un grafo ponderado de la red de carreteras. Sin embargo, no podemos resolver el problema solo con el grafo. Esto se debe a que el estado<sup>10</sup> de este problema requiere no solo la ubicación actual (ciudad), sino también el nivel de combustible en esa ubicación. De otro modo, no podremos determinar si el coche tiene suficiente combustible para realizar un viaje por determinada carretera (porque no podemos repostar en mitad del recorrido). Por lo tanto, usamos una pareja de datos para representar el estado (*ubicación, combustible*) y, al hacerlo, el número total de vértices del grafo modificado *explota*, al pasar de 1000 a  $1000 \times 100 = 100000$  vértices. Llamamos al grafo modificado ‘grafo estado-espacio’.

En el grafo estado-espacio, el vértice de origen es el estado  $(s, 0)$ , una ciudad de inicio  $s$  con el depósito vacío, y los vértices de destino son los estados  $(e, \text{cualquiera})$ , una ciudad de destino  $e$  con un nivel de combustible entre  $[0..c]$ . Hay dos tipos de aristas en el grafo estado-espacio: las de peso 0 que van desde el vértice  $(x, \text{combustible}_x)$  al  $(y, \text{combustible}_x - \text{longitud}(x, y))$ , si el coche tiene combustible suficiente para viajar del vértice  $x$  al  $y$ , y las de peso  $p[x]$  que van desde el vértice  $(x, \text{combustible}_x)$  al  $(x, \text{combustible}_x + 1)$ , si el coche puede repostar en el vértice  $x$  una unidad de combustible (el nivel de combustible no puede exceder de la capacidad del depósito  $c$ ). La ejecución del algoritmo de Dijkstra en este grafo estado-espacio, nos dará la solución del problema (ver también la sección 8.2.3, para más información).

---

<sup>10</sup>Recordatorio: el estado es un subconjunto de parámetros del problema, que puede describir implícitamente al propio problema.

### Ejercicio 4.4.3.1

La variante de implementación del algoritmo de Dijkstra modificada que hemos visto puede ser diferente a la que encuentres en otros libros (por ejemplo en [7, 38, 8]). Analiza si esta variante se ejecuta, efectivamente, en  $O((V+E) \log V)$ , en varios tipos de grafos ponderados (ver también el [ejercicio 4.4.3.2\\*](#)).

### Ejercicio 4.4.3.2\*

Construye un grafo que tenga aristas de peso negativo, pero ningún ciclo negativo, que pueda ralentizar significativamente esta implementación de Dijkstra.

### Ejercicio 4.4.3.3

La única razón por la que esta variante permite vértices duplicados en la cola de prioridad, es para poder utilizar la cola de prioridad integrada en el compilador tal y como se presenta. Hay otra implementación alternativa, que también utiliza muy poco código. Se realiza utilizando `set`. Implementa esta variante y comenta las ventajas y desventajas de ambas opciones.

### Ejercicio 4.4.3.4

El código fuente mostrado anteriormente utiliza `priority_queue< ii, vector<ii>, greater<ii> > pq;` para ordenar las parejas de enteros por distancia creciente al origen  $s$ . ¿Podemos lograr el mismo efecto sin definir el operador de comparación de la `priority_queue`? Consejo: hemos utilizado un truco similar en la implementación del algoritmo de Kruskal, en la sección 4.3.2.

### Ejercicio 4.4.3.5

En el [ejercicio 4.4.2.2](#), hemos visto una forma de acelerar la solución de un problema de caminos más cortos, si se nos dan los vértices tanto de origen como de destino. ¿Podemos utilizar el mismo truco de aceleración para todos los tipos de grafos ponderados?

### Ejercicio 4.4.3.6

El modelado del grafo del problema UVa 11367 mencionado, transforma el problema de los SSSP en un grafo ponderado en un problema de SSSP en un grafo *estado-espacio* ponderado. ¿Podemos resolver este problema mediante programación dinámica? Si es posible, ¿por qué? Si no lo es, ¿por qué no? Consejo: lee la sección 4.7.1.

#### 4.4.4 SSSP en un grafo con ciclo de peso negativo

Si el grafo de entrada tiene pesos de aristas negativos, la implementación típica de Dijkstra (por ejemplo [7, 38, 8]) puede resultar en una respuesta incorrecta. Sin embargo, la variante de implementación de Dijkstra que aparece en la sección 4.4.3 funcionará bien, aunque es más lenta. Pruébalo en el grafo de la figura 4.18.

Esto es debido a que la variante de implementación de Dijkstra insertará en la cola de prioridad nuevas parejas de información de vértices, cada vez que hace una operación de relajación. Así, si el grafo no tiene *ciclos* de peso negativo, la variante seguirá propagando información de distancia del camino más corto, hasta que no sea posible realizar más operaciones de relajación (lo que implica que ya se han encontrado todos los caminos más cortos desde el origen). Sin embargo, cuando tenemos un grafo con *ciclos* de peso negativo, la variante, si se implementa como en la sección 4.4.3, se verá atrapada en un bucle infinito.

Ejemplo: veamos el grafo de la figura 4.19. El ciclo 1-2-1 tiene un peso negativo. El peso de este ciclo es  $15 + (-42) = -27$ .

Para resolver el problema de los SSSP con la presencia potencial de *ciclos* de peso negativo, podemos utilizar el algoritmo de Bellman Ford, más generalista, pero más lento. Este algoritmo lo inventaron Richard Ernest Bellman (el pionero de las técnicas de DP) y Lester Randolph Ford, Jr (la misma persona que inventó el método Ford Fulkerson de la sección 4.6.2). La idea de este algoritmo es sencilla: relajar todas las  $E$  aristas (en orden arbitrario)  $V-1$  veces.

Inicialmente  $\text{dist}[s] = 0$ , que es el caso base. Si relajamos una arista  $s \rightarrow u$ , entonces  $\text{dist}[u]$  tendrá el valor correcto. Si, después, relajamos una arista  $u \rightarrow v$ ,  $\text{dist}[v]$  también tendrá el valor correcto. Si hemos relajado las  $E$  aristas  $V-1$  veces, el camino más corto desde el vértice de origen a su vértice más alejado (que será un camino sencillo con  $V-1$  aristas), debería haberse calculado correctamente. La parte principal del código del algoritmo de Bellman Ford es más sencilla que los de BFS y Dijkstra:

```

1 vi dist(V, INF); dist[s] = 0;
2 for (int i = 0; i < V-1; i++) // relajar las E aristas V-1 veces
3 for (int u = 0; u < V; u++) // los dos bucles = O(E), total O(VE)
4 for (int j = 0; j < (int)AdjList[u].size(); j++) {
5 ii v = AdjList[u][j]; // guardar la expansión SP si es necesario
6 dist[v.first] = min(dist[v.first], dist[u]+v.second); // relajar
7 }

```

La complejidad del algoritmo de Bellman Ford es  $O(V^3)$ , si el grafo se almacena como una matriz de adyacencias, o  $O(VE)$ , si se hace como una lista de adyacencias o de aristas. Esto se debe, sencillamente, a que si usamos una matriz de adyacencias, necesitamos  $O(V^2)$  para enumerar todas las aristas del grafo. Ambas complejidades de tiempo son (mucho) más lentas que Dijkstra. Sin embargo, la forma en que funciona Bellman Ford, asegura que nunca se verá atrapado en un bucle infinito, aunque el grafo dado tenga un ciclo negativo. De hecho, se puede utilizar el algoritmo de Bellman Ford para detectar la *presencia* de ciclos negativos (por ejemplo en UVa 558 - Wormholes), aunque ese problema de SSSP está mal definido.

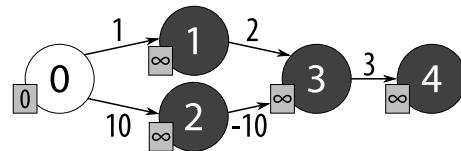


Figura 4.18: Peso negativo

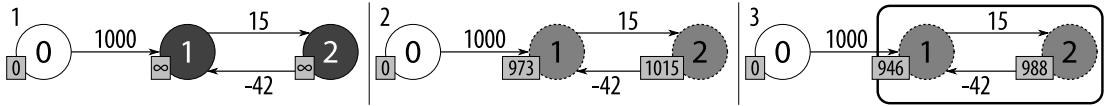


Figura 4.19: Bellman Ford puede detectar la presencia de ciclos negativos (de UVa 558 [49])

En el **ejercicio 4.4.4.1** demostramos que, después de relajar todas las  $E$  aristas  $V-1$  veces, el problema de los SSSP debería quedar resuelto, es decir, no podemos relajar ninguna arista más. Corolario: si aún se puede relajar una arista, habrá un ciclo negativo en el grafo ponderado.

Por ejemplo, en la parte izquierda de la figura 4.19, vemos un grafo simple con un ciclo negativo. Después de 1 pasada,  $\text{dist}[1] = 973$  y  $\text{dist}[2] = 1015$  (centro). Después de  $V-1 = 2$  pasadas,  $\text{dist}[1] = 946$  y  $\text{dist}[2] = 988$  (derecha). Como hay un ciclo negativo, podemos hacer esto una y otra vez, es decir, todavía podemos relajar  $\text{dist}[2] = 946 + 15 = 961$ . Este valor es menor que el actual de  $\text{dist}[2] = 988$ . La presencia de un ciclo negativo provoca que los vértices alcanzables por el mismo tengan información errónea de los caminos más cortos. Se debe a que es posible recorrer ese ciclo negativo un número infinito de veces, para hacer que todos los vértices alcanzables desde el ciclo negativo tengan información de caminos más cortos negativa infinita. El código para comprobar la existencia de un ciclo negativo es sencillo:

```

1 // después de ejecutar el algoritmo de Bellman Ford en $O(VE)$ visto antes
2 bool hasNegativeCycle = false;
3 for (int u = 0; u < V; u++) // una pasada más para comprobar
4 for (int j = 0; j < (int)AdjList[u].size(); j++) {
5 int v = AdjList[u][j];
6 if (dist[v.first] > dist[u]+v.second) // si esto todavía es posible
7 hasNegativeCycle = true; // es que existe un ciclo negativo
8 }
9 printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

```

En los concursos de programación, la lentitud del algoritmo de Bellman Ford, y su característica de detección de ciclos negativos, provocan que se utilice solo para resolver el problema de los SSSP en grafos *pequeños* que *no tengan garantizada* la ausencia de ciclos de peso negativo.

### Ejercicio 4.4.4.1

¿Por qué nos basta con relajar todas las  $E$  aristas de un grafo ponderado  $V - 1$  veces para obtener la información correcta de los SSSP? Demuéstralos.

### Ejercicio 4.4.4.2

La complejidad de tiempo de  $O(VE)$ , para el peor caso posible, es demasiado alta en la práctica. En la mayoría de los casos podemos detener el algoritmo de Bellman Ford (mucho) antes. Sugiere una mejora simple del código anterior, para hacer que Bellman Ford se ejecute *normalmente* más rápido que en  $O(VE)$ .



## Ejercicios de programación

Ejercicios de programación relativos a caminos más cortos de origen único:

En grafos no ponderados: BFS, fáciles

1. UVa 00336 - A Node Too Far  
(SSSP sencillo que se resuelve con BFS)
2. UVa 00383 - Shipping Routes  
(SSSP sencillo que se resuelve con BFS; usar mapa)
3. UVa 00388 - *Galactic Import*  
(idea clave: queremos minimizar los movimientos planetarios porque cada arista usada reduce el valor en un 5 %)
4. **UVa 00429 - Word Transformation \***  
(cada palabra es un vértice, conectar dos palabras con una arista si difieren por una letra)  
(mostrar también el camino)
5. UVa 00627 - The Net  
(SSSP sencillo que se resuelve con BFS; usar mapa)
6. UVa 00762 - We Ship Cheap  
(la distribución es como un recorrido BFS)
7. **UVa 00924 - Spreading the News \***  
(LA 3502 - SouthWesternEurope05; problema de camino más corto de origen y destino únicos excluyendo los extremos)
8. *UVa 01148 - The mysterious X network*  
(SSSP sencillo que se resuelve con BFS)
9. UVa 10009 - All Roads Lead Where?  
(SSSP sencillo que se resuelve con BFS)
10. UVa 10422 - Knights in FEN  
(SSSP sencillo que se resuelve con BFS)
11. UVa 10610 - Gopher and Hawks  
(SSSP sencillo que se resuelve con BFS)
12. **UVa 10653 - Bombs; NO they ... \***  
(necesita implementación de BFS eficiente)
13. UVa 10959 - The Party, Part I  
(SSSP desde origen 0 al resto)

En grafos no ponderados: BFS, difíciles

1. **UVa 00314 - Robot \***  
(procesar previamente el grafo de entrada; después ejecutar BFS con estados/vértices (fila, columna, dirección) y 5 transiciones/aristas: giro izquierda/derecha o moverse 1/2/3 pasos)
2. UVa 00532 - Dungeon Master  
(BFS tridimensional)
3. *UVa 00859 - Chinese Checkers*  
(interesante variación de estructura de datos de un grafo)
4. UVa 00949 - Getaway  
(el procesado de la entrada es complicado; consulta la sección 6.2)
5. UVa 10044 - Erdos numbers  
(grafo implícito en el enunciado del problema)
6. UVa 10067 - Playing with Wheels  
(el estado BFS es una cadena)
7. UVa 10150 - Doublets  
(BFS con estados bloqueados)
8. UVa 10977 - Enchanted Forest  
(algunos movimientos restringidos; mostrar el camino)
9. UVa 11049 - Basic Wall Maze  
(BFS de origen múltiple desde m1; obtener mínimo en el borde de m2)
10. **UVa 11101 - Mall Mania \***  
(primero filtrar el grafo; entonces se convierte en SSSP)
11. UVa 11352 - Crazy King  
(BFS con orígenes múltiples)
12. UVa 11624 - Fire  
(cuidado con la 'estación importante')
13. UVa 11792 - Krochanska is Here  
(LA 4408 - KualaLumpur08; s: (el número de 4 dígitos); enlazar dos números a una arista si podemos utilizar un botón para transformar uno en el otro; usar BFS)
14. **UVa 12160 - Unlock the Lock \***

### En grafos ponderados: Dijkstra, fáciles

1. **UVa 00929 - Number Maze \***  
(laberinto bidimensional/grafo implícito)
2. **UVa 01112 - Mice and Maze \***  
(LA 2425 - SouthwesternEurope01; ejecutar Dijkstra desde el destino)  
(geometría básica para construir el grafo ponderado; después Dijkstra)  
(aplicación directa de Dijkstra)

### En grafos ponderados: Dijkstra, difíciles

1. UVa 01202 - Finding Nemo  
(LA 3133 - Beijing04; SSSP; Dijkstra en una rejilla: tratar cada celda como un vértice; la idea es sencilla, pero cuidado con la implementación)  
(se puede modelar como un problema de SSSP)
2. UVa 10166 - Travel  
(casos especiales: inicio = destino: 0 litros; ciudad de origen o destino no encontradas o ciudad de destino no alcanzable: no hay ruta; resto: Dijkstra)
3. *UVa 10187 - From Dusk Till Dawn*  
(Dijkstra desde los parques de bomberos a todas las intersecciones; necesita poda para cumplir con el tiempo límite)  
(podemos añadir un dato adicional a cada vértice: si hemos llegado al mismo utilizando un ciclo o no; después, ejecutar Dijkstra para resolver los SSSP)  
(estado: (a, b, c), origen: (0, 0, c), 6 transiciones posibles)
4. UVa 10278 - Fire Station  
(modelar el grafo con cuidado)
5. *UVa 10356 - Rough Roads*  
(parece que los casos de prueba son más fáciles de lo que indica el enunciado ( $n \leq 10000$ ); utilizamos un bucle  $O(n^2)$  para construir el grafo ponderado y se ejecuta Dijkstra sin recibir TLE)  
(tratado en esta sección)
6. UVa 10603 - Fill  
(de una ciudad a otra sin/con aeropuerto tiene un peso de arista de 1/0, respectivamente; Dijkstra (o BFS con deque); si la ciudad de inicio y fin son la misma y no hay aeropuerto, la respuesta debe ser 0)
7. **UVa 10801 - Lift Hopping \***  
(modelado de grafos; cada palabra es un vértice; conectar dos vértices con una arista si comparten un idioma común y su primer carácter es diferente; conectar un vértice de origen a todas las palabras que pertenecen a un idioma de inicio; conectar todas las palabras que pertenecen a un idioma de finalización a un vértice desagüe; podemos transferir el peso del vértice al de la arista; entonces, SSSP desde el vértice de origen al desagüe)  
(detener Dijkstra en el camino de la ruta de servicio y alguna modificación)  
(uso inteligente de Dijkstra; ejecutar Dijkstra desde el origen y desde el destino; probar todas las aristas  $(u, v)$  si  $\text{dist}[\text{origen}][u] + \text{peso}(u, v) + \text{dist}[v][\text{destino}] \leq p$ ; guardar el mayor peso de arista encontrado)
8. *UVa 10967 - The Great Escape*  
(Dijkstra; guardar predecesores múltiples)  
(se puede modelar como un problema de SSSP)
9. *UVa 11338 - Minefield*  
(tratado en esta sección)
10. UVa 11367 - Full Tank?  
(de una ciudad a otra sin/con aeropuerto tiene un peso de arista de 1/0, respectivamente; Dijkstra (o BFS con deque); si la ciudad de inicio y fin son la misma y no hay aeropuerto, la respuesta debe ser 0)
11. UVa 11377 - Airport Setup  
(modelado de grafos; cada palabra es un vértice; conectar dos vértices con una arista si comparten un idioma común y su primer carácter es diferente; conectar un vértice de origen a todas las palabras que pertenecen a un idioma de inicio; conectar todas las palabras que pertenecen a un idioma de finalización a un vértice desagüe; podemos transferir el peso del vértice al de la arista; entonces, SSSP desde el vértice de origen al desagüe)  
(detener Dijkstra en el camino de la ruta de servicio y alguna modificación)  
(uso inteligente de Dijkstra; ejecutar Dijkstra desde el origen y desde el destino; probar todas las aristas  $(u, v)$  si  $\text{dist}[\text{origen}][u] + \text{peso}(u, v) + \text{dist}[v][\text{destino}] \leq p$ ; guardar el mayor peso de arista encontrado)
12. **UVa 11492 - Babel \***  
(tratado en esta sección)
13. UVa 11833 - Route Change  
(de una ciudad a otra sin/con aeropuerto tiene un peso de arista de 1/0, respectivamente; Dijkstra (o BFS con deque); si la ciudad de inicio y fin son la misma y no hay aeropuerto, la respuesta debe ser 0)
14. **UVa 12047 - Highest Paid Toll \***  
(tratado en esta sección)
15. *UVa 12144 - Almost Shortest Path*  
(Dijkstra; guardar predecesores múltiples)  
(se puede modelar como un problema de SSSP)
16. IOI 2011 - Crocodile

### SSSP en grafos con ciclos de peso negativo (Bellman Ford)

1. **UVa 00558 - Wormholes \***  
(comprobar si existe un ciclo negativo)
2. **UVa 10449 - Traffic \***  
(buscar el camino de peso mínimo, que puede ser negativo; cuidado:  $\infty +$  peso negativo es menos que  $\infty$ )
3. **UVa 10557 - XYZZY \***  
(comprobar ciclo 'positivo'; comprobar conectividad)
4. UVa 11280 - Flying to Fredericton  
(Bellman Ford modificado)

## 4.5 Caminos más cortos entre todos los pares

### 4.5.1 Introducción y motivación

Problema de referencia: dado un grafo conexo y ponderado  $G$ , con  $V \leq 100$ , y dos vértices  $s$  y  $d$ , encontrar el valor máximo posible de  $\text{dist}[s][i] + \text{dist}[i][d]$ , de entre todos los  $i \in [0 \dots V-1]$ . Esta es la idea clave para resolver UVa 11463 - Commandos. Sin embargo, ¿cuál es la mejor manera de implementar la solución a este problema?

El problema necesita la información de los caminos más cortos desde todos los orígenes (todos los vértices) posibles de  $G$ . Podemos hacer  $V$  llamadas al algoritmo de Dijkstra, que hemos visto antes en la sección 4.4.3. Pero, ¿se puede resolver este problema con un código más corto? La respuesta es que sí. Si el grafo ponderado dado tiene  $V \leq 400$ , existe otro algoritmo que es *mucho más fácil de programar*.

Cargamos el pequeño grafo en una matriz de adyacencia y ejecutamos el siguiente código de cuatro líneas, con tres bucles anidados. Cuando termine,  $\text{AdjMat}[i][j]$  contendrá la distancia del camino más corto entre el par de vértices  $i$  y  $j$  de  $G$ . Ahora, el problema original (UVa 11463), se ha convertido en fácil.

```
1 // dentro de int main()
2 // condición previa: AdjMat[i][j] contiene peso de la arista (i, j)
3 // o INF (1000M) si no existe esa arista
4 // AdjMat es un array de enteros con signo de 32 bits
5 for (int k = 0; k < V; k++) // el orden de los bucles es k->i->j
6 for (int i = 0; i < V; i++)
7 for (int j = 0; j < V; j++)
8 AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k]+AdjMat[k][j]);
```



ch4\_07\_floyd\_marshall.cpp



ch4\_07\_floyd\_marshall.java

Este es el algoritmo de Floyd Warshall, inventado por Robert W. *Floyd* [19] y Stephen *Warshall* [70]. El algoritmo de Floyd Warshall es un algoritmo de programación dinámica que, evidentemente, se ejecuta en  $O(V^3)$ , debido a sus tres bucles anidados<sup>11</sup>. Por lo tanto, en concursos de programación, solo puede utilizarse con grafos donde  $V \leq 400$ . De forma generalista, Floyd Warshall resuelve otro problema de grafos clásico: el problema de los caminos más cortos entre todos los pares (APSP). Es una alternativa (para grafos pequeños) a llamar al algoritmo SSSP varias veces:

1.  $V$  llamadas a Dijkstra en  $O((V+E) \log V) = O(V^3 \log V)$  si  $E = O(V^2)$ .
2.  $V$  llamadas a Bellman Ford en  $O(VE) = O(V^4)$  si  $E = O(V^2)$ .

El principal atractivo de Floyd Warshall, en los concursos de programación, reside en la velocidad de su implementación, con solo cuatro líneas. Si el grafo dado es pequeño ( $V \leq 400$ ), no dudes en utilizarlo, aunque solo necesites una solución para el problema del SSSP.

<sup>11</sup>Floyd Warshall utilizará una matriz de adyacencia para que el peso de la arista  $(i, j)$  sea accesible en  $O(1)$ .

### Ejercicio 4.5.1.1

¿Hay alguna razón por la que  $\text{AdjMat}[i][j]$  se deba establecer a mil millones ( $10^9$ ) para indicar que no hay ninguna arista entre  $i$  y  $j$ ? ¿Por qué no utilizar  $2^{31} - 1$  (`MAX_INT`)?

### Ejercicio 4.5.1.2

En la sección 4.4.4, diferenciamos los grafos con aristas de peso negativo, pero sin ciclos de peso negativo, y grafos con ciclos de peso negativo. ¿Funcionará este algoritmo de Floyd Warshall en grafos de peso negativo y/o ciclos de peso negativo?

## 4.5.2 Explicación de la solución de DP de Floyd Warshall

Incluimos esta sección en beneficio de los lectores interesados en saber por qué funciona el algoritmo de Floyd Warshall. Puedes no leerla, si te das por satisfecho con utilizar el algoritmo. Sin embargo, interesarte por esta sección puede fortalecer tus habilidades de programación dinámica. Ten en cuenta que hay problemas de grafos para los que todavía no existe un algoritmo clásico y deben ser resueltos mediante técnicas de DP (ver la sección 4.7.1).

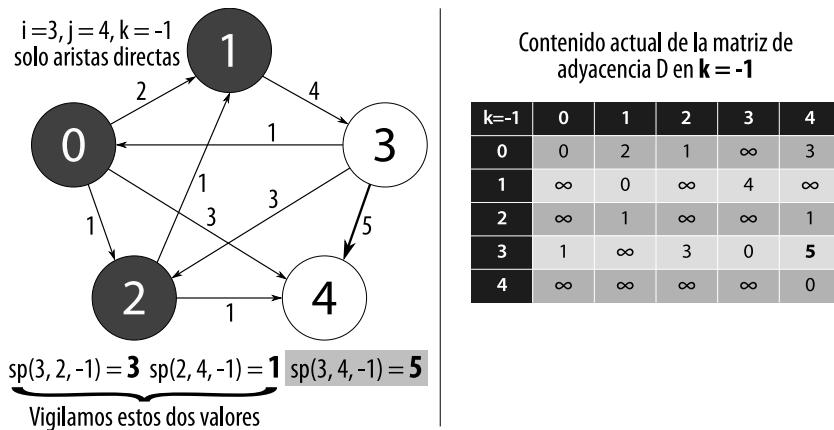


Figura 4.20: Explicación de Floyd Warshall 1

La idea básica detrás del algoritmo de Floyd Warshall está en permitir, de forma gradual, el uso de vértices intermedios (vértice  $[0..k]$ ), para formar los caminos más cortos. Identificamos el camino más corto del vértice  $i$  al  $j$  utilizando solo vértices intermedios  $[0..k]$ , como  $\text{sp}(i, j, k)$ . Vamos a etiquetar los vértices desde 0 hasta  $V-1$ . Comenzamos solo con aristas directas cuando  $k = -1$ , es decir,  $\text{sp}(i, j, -1) =$  peso de la arista  $(i, j)$ . Después, encontramos los caminos más cortos entre dos vértices cualesquiera, con la ayuda de vértices intermedios restringidos desde el vértice  $[0..k]$ . En la figura 4.20, queremos encontrar  $\text{sp}(3, 4, 4)$ , el camino más corto desde el vértice 3 al 4, utilizando cualquier vértice intermedio del grafo (vértice  $[0..4]$ ). El camino más corto resultará ser 3-0-2-4, con coste 3. Pero, ¿cómo llegamos a esta solución? Sabemos que, utilizando solo aristas directas,  $\text{sp}(3, 4, -1) = 5$ , como se muestra en la figura 4.20. Llegaremos

a la solución para  $\text{sp}(3,4,4)$ , en algún momento, desde  $\text{sp}(3,2,2)+\text{sp}(2,4,2)$ . Pero al usar solo aristas directas,  $\text{sp}(3,2,-1)+\text{sp}(2,4,-1) = 3+1 = 4$  sigue siendo  $> 3$ .

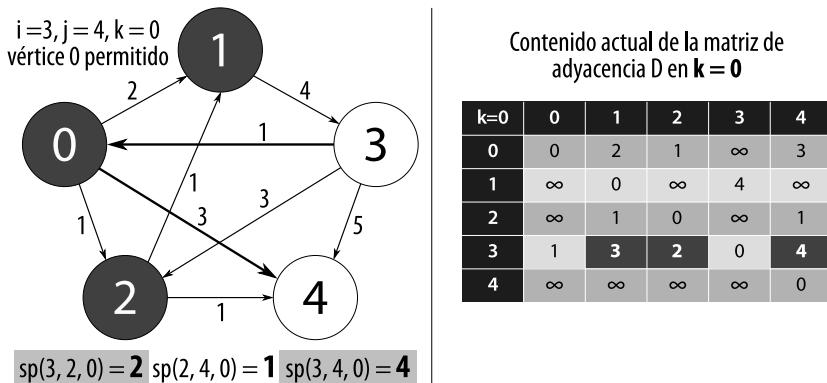


Figura 4.21: Explicación de Floyd Warshall 2

Entonces, Floyd Warshall va permitiendo gradualmente  $k = 0$ , seguido de  $k = 1, k = 2 \dots$  hasta  $k = V-1$ . Cuando permitimos  $k = 0$ , es decir, cuando el vértice 0 se puede utilizar como intermedio,  $\text{sp}(3,4,0)$  queda reducido a  $\text{sp}(3,4,0) = \text{sp}(3,0,-1) + \text{sp}(0,4,-1) = 1+3 = 4$ , como se muestra en la figura 4.21. También, al utilizar  $k = 0$ ,  $\text{sp}(3,2,0)$ , que necesitaremos después, se reduce de 3 a  $\text{sp}(3,0,-1) + \text{sp}(0,2,-1) = 1+1 = 2$ . Floyd Warshall procesará  $\text{sp}(i,j,0)$  para el resto de pares, teniendo en consideración solo el vértice 0 como intermedio, pero hay un cambio más:  $\text{sp}(3,1,0)$  de  $\infty$  a 3.

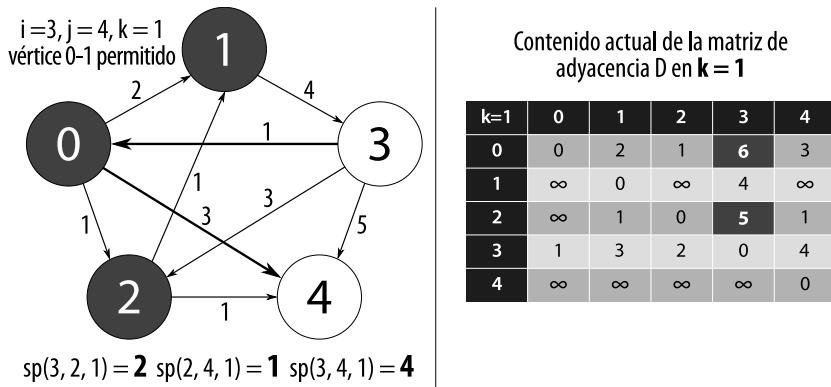
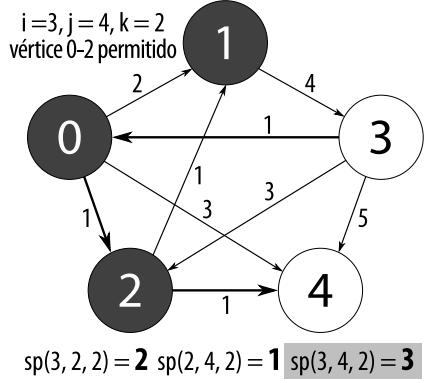


Figura 4.22: Explicación de Floyd Warshall 3

Cuando permitimos  $k = 1$ , es decir, los vértices 0 y 1 se pueden utilizar ahora como vértices intermedios, ocurre que no hay cambios en  $\text{sp}(3,2,1)$ ,  $\text{sp}(2,4,1)$  ni  $\text{sp}(3,4,1)$ . Sin embargo, cambian los valores de  $\text{sp}(0,3,1)$  y  $\text{sp}(2,3,1)$ , como se muestra en la figura 4.22, pero estos no afectarán al cálculo final del camino más corto entre los vértices 3 y 4.

Al permitir  $k = 2$ , es decir, se pueden utilizar los vértices 0, 1 y 2 como intermedios,  $\text{sp}(3,4,2)$  se vuelve a reducir a  $\text{sp}(3,4,2) = \text{sp}(3,2,2)+\text{sp}(2,4,2) = 2+1 = 3$ , como se muestra en la figura 4.23. Floyd Warshall repite este proceso para  $k = 3$  y, finalmente,  $k = 4$ , pero  $\text{sp}(3,4,4)$  se mantiene en 3, y esta es la respuesta final.



Contenido actual de la matriz de adyacencia D en  $k=2$

| $k=2$ | 0        | 1        | 2        | 3        | 4        |
|-------|----------|----------|----------|----------|----------|
| 0     | 0        | 2        | 1        | 6        | 2        |
| 1     | $\infty$ | 0        | $\infty$ | 4        | $\infty$ |
| 2     | $\infty$ | 1        | 0        | 5        | 1        |
| 3     | 1        | 3        | 2        | 0        | 3        |
| 4     | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Figura 4.23: Explicación de Floyd Warshall 4

Formalmente, definimos las recurrencias de programación dinámica de Floyd Warshall de la siguiente manera. Pongamos que  $D_{i,j}^k$  es la distancia más corta entre  $i$  y  $j$  con solo  $[0..k]$  como vértices intermedios. A partir de ahí, el caso base y las recurrencias de Floyd Warshall son:

- $D_{i,j}^{-1} = \text{peso}(i,j)$ . Este es el caso base cuando no utilizamos ningún vértice intermedio.
- $D_{i,j}^k = \min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}) = \min(\text{sin vértice } k, \text{ con vértice } k)$ , para  $k \geq 0$ .

Esta formulación de DP se debe llenar capa a capa (incrementando  $k$ ). Para completar una entrada en la tabla  $k$ , debemos hacer uso de las entradas en la tabla  $k-1$ . Por ejemplo, para calcular  $D_{3,4}^2$ , (fila 3, columna 4,  $k = 2$  en la tabla, con el índice empezando en 0), consultamos el mínimo de  $D_{3,4}^1$ , o la suma de  $D_{3,2}^1 + D_{2,4}^1$  (ver tabla 4.3). La implementación ingenua consiste en utilizar una matriz de tres dimensiones  $D[k][i][j]$ , de tamaño  $O(V^3)$ . Sin embargo, como para calcular la capa  $k$ , solo necesitamos conocer los valores de la capa  $k-1$ , podemos abandonar la dimensión  $k$  y calcular  $D[i][j]$  ‘al vuelo’ (ver el truco de ahorro de espacio de la sección 3.5.1). Por lo tanto, el algoritmo de Floyd Warshall solo necesita un espacio de  $O(V^2)$ , aunque se sigue ejecutando en tiempo  $O(V^3)$ .

|       |  | $k$ |          | $j$      |          |          |          |  |  |
|-------|--|-----|----------|----------|----------|----------|----------|--|--|
|       |  | 0   | 1        | 2        | 3        | 4        |          |  |  |
| $k=1$ |  | 0   | 0        | 2        | 1        | 6        | 3        |  |  |
|       |  | 1   | $\infty$ | 0        | $\infty$ | 4        | $\infty$ |  |  |
| $k$   |  | 2   | $\infty$ | 1        | 0        | 5        | 1        |  |  |
| $i$   |  | 3   | 1        | 3        | 2        | 0        | 4        |  |  |
|       |  | 4   | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |  |  |

|       |  | $k=1$ |          |          |          | $k=2$    |          |
|-------|--|-------|----------|----------|----------|----------|----------|
|       |  | 0     | 1        | 2        | 3        | 4        |          |
| $k=2$ |  | 0     | 0        | 2        | 1        | 6        | 2        |
|       |  | 1     | $\infty$ | 0        | $\infty$ | 4        | $\infty$ |
| $k$   |  | 2     | $\infty$ | 1        | 0        | 5        | 1        |
| $i$   |  | 3     | 1        | 3        | 2        | 0        | 3        |
|       |  | 4     | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Tabla 4.3: Tabla de DP de Floyd Warshall

### 4.5.3 Otras aplicaciones

El uso más habitual de Floyd Warshall es resolver el problema de los APSP. Sin embargo, se utiliza frecuentemente en otros problemas, siempre que el grafo de entrada sea pequeño. A continuación, enumeramos algunas variantes de problemas que también se pueden resolver mediante este algoritmo.

#### Solución al problema de los SSSP en un grafo ponderado pequeño

Si tenemos la información de los caminos más cortos entre todos los pares (APSP), también podemos obtener la información de los caminos más cortos de origen único (SSSP) de cualquier origen posible. Si el grafo ponderado dado es pequeño,  $V \leq 400$ , puede resultar beneficioso, en relación al tiempo de programación, utilizar el código de cuatro líneas de Floyd Warshall frente al algoritmo de Dijkstra, más largo.

#### Mostrar los caminos más cortos

Un problema habitual, que encuentran los programadores que utilizan la implementación de cuatro líneas de Floyd Warshall sin entender su funcionamiento, se pone de relevancia cuando se les pide que, además, indiquen cuáles son los caminos más cortos. En los algoritmos BFS, Dijkstra o Bellman Ford, nos basta con recordar el árbol de expansión de los caminos más cortos, utilizando un vector unidimensional, para almacenar la información de los padres de cada vértice. En Floyd Warshall, debemos utilizar una matriz bidimensional para ello. Reproducimos a continuación el código modificado:

```
1 // dentro de int main()
2 // p = matriz padre 2D, donde p[i][j] es el último vértice antes de j en
3 // un camino mínimo desde i a j, como en i -> ... -> p[i][j] -> j
4 for (int i = 0; i < V; i++)
5 for (int j = 0; j < V; j++)
6 p[i][j] = i; // inicializar la matriz padre
7 for (int k = 0; k < V; k++)
8 for (int i = 0; i < V; i++)
9 for (int j = 0; j < V; j++) // esta vez hay que usar la expresión if
10 if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
11 AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
12 p[i][j] = p[k][j]; // actualizar la matriz padre
13 }
14 -----
15 // cuando debemos mostrar los caminos más cortos, podemos usar la llamada:
16 void printPath(int i, int j) {
17 if (i != j) printPath(i, p[i][j]);
18 printf(" %d", j);
19 }
```

#### Clausura transitiva (algoritmo de Warshall)

Stephen Warshall [70] desarrolló un algoritmo para el problema de la clausura transitiva: dado un grafo, determinar si el vértice  $i$  está conectado al  $j$ , directa o indirectamente. Esta variante

utiliza operadores lógicos de bits, que son (mucho) más rápidos que los operadores aritméticos. Inicialmente,  $\text{AdjMat}[i][j]$  contiene 1 (verdadero), si el vértice  $i$  está conectado *directamente* al vértice  $j$  y 0 (falso) en caso contrario. Después de ejecutar el siguiente código del algoritmo de Warshall en  $O(V^3)$ , podemos comprobar si dos vértices  $i$  y  $j$  cualesquiera están conectados directa o indirectamente, comprobando  $\text{AdjMat}[i][j]$ .

```

1 for (int k = 0; k < V; k++)
2 for (int i = 0; i < V; i++)
3 for (int j = 0; j < V; j++)
4 AdjMat[i][j] |= (AdjMat[i][k] & AdjMat[k][j]);

```

### **Minimax y maximin (revisitado)**

Hemos visto el problema del camino *minimax* (y *maximin*) en la sección 4.3.4. La solución utilizando el algoritmo de Floyd Warshall se incluye a continuación. En primer lugar, inicializamos  $\text{AdjMat}[i][j]$  al peso de la arista  $(i, j)$ . Este es el coste *minimax* predeterminado, para dos vértices que están conectados directamente. Para la pareja  $i - j$ , sin ninguna arista directa, establecemos  $\text{AdjMat}[i][j] = \text{INF}$ . Después, probamos todos los  $k$  vértices intermedios posibles. El coste *minimax*  $\text{AdjMat}[i][j]$  es el mínimo entre sí mismo y el máximo entre  $\text{AdjMat}[i][k]$  o  $\text{AdjMat}[k][j]$ . Sin embargo, esta técnica solo se puede utilizar si el grafo de entrada es lo suficientemente pequeño ( $V \leq 400$ ).

```

1 for (int k = 0; k < V; k++)
2 for (int i = 0; i < V; i++)
3 for (int j = 0; j < V; j++)
4 AdjMat[i][j] = min(AdjMat[i][j], max(AdjMat[i][k], AdjMat[k][j]));

```

### **Búsqueda del ciclo más barato/negativo**

En la sección 4.4.4, hemos visto cómo el algoritmo de Bellman Ford finaliza después de  $O(VE)$ , independientemente del tipo del grafo de entrada (ya que relaja todas las  $E$  aristas un máximo de  $V-1$  veces), y cómo se puede utilizar para comprobar si un grafo dado tiene ciclos negativos. Floyd Warshall también finaliza después de  $O(V^3)$  pasos, independientemente del tipo del grafo de entrada. Esta característica permite utilizar Floyd Warshall para detectar si el grafo (pequeño) tiene ciclos, ciclos negativos e, incluso, encontrar el ciclo más barato (no negativo) de entre todos los posibles (la cintura del grafo).

Para hacerlo, fijamos inicialmente la *diagonal principal* de la matriz de adyacencia a un valor muy grande, como  $\text{AdjMat}[i][i] = \text{INF}$  (1000M). Después, ejecutamos el algoritmo de Floyd Warshall en  $O(V^3)$ . En este momento, comprobamos el valor de  $\text{AdjMat}[i][i]$ , que ahora indica el peso del camino cíclico más corto desde el vértice  $i$ , que pasa por hasta  $V-1$  vértices intermedios y vuelve a  $i$ . Si  $\text{AdjMat}[i][i]$  ya no es  $\text{INF}$  en ningún  $i \in [0..V-1]$ , tenemos un ciclo. El  $\text{AdjMat}[i][i]$  no negativo menor,  $\forall i \in [0..V-1]$  es el ciclo *más barato*. Si  $\text{AdjMat}[i][i] < 0$  para cualquier  $i \in [0..V-1]$ , entonces tenemos un ciclo *negativo*, porque si tomamos este camino cíclico más de una vez, obtendremos un camino más corto que el ‘más corto’.

## Búsqueda del diámetro de un grafo

El diámetro de un grafo se define como la longitud máxima de los caminos más cortos entre cualquier par de vértices de ese grafo. Para encontrar el diámetro de un grafo, primero buscamos el camino más corto entre cada par de vértices (es decir, el problema de los APSP). La distancia máxima de las encontradas es el diámetro del grafo. UVa 1056 - Degrees of Separation, que apareció en la final mundial de 2006 del ICPC es, precisamente, este problema. Para resolverlo, podemos ejecutar inicialmente el algoritmo de Floyd Warshall en  $O(V^3)$ , para calcular la información APSP necesaria. Después, podemos identificar el diámetro del grafo buscando el valor máximo en  $\text{AdjMat}$  en  $O(V^2)$ . Pero solo podremos hacerlo en un grafo pequeño con  $V \leq 400$ .

## Búsqueda de los SCC de un grafo dirigido

En la sección 4.2.1, hemos visto cómo el algoritmo de Tarjan en  $O(V+E)$  puede utilizarse para identificar los SCC de un grafo dirigido. Sin embargo, el código resulta un tanto largo. Si el grafo de entrada es pequeño (por ejemplo, UVa 247 - Calling Circles, UVa 1229 - Sub-dictionary, UVa 10731 - Test), también podemos identificar los SCC del grafo en  $O(V^3)$ , utilizando el algoritmo de clausura transitiva de Warshall y, después, realizando la siguiente comprobación: para encontrar todos los miembros de un SCC que contengan un vértice  $i$ , comprobar el resto de vértices  $j \in [0..V-1]$ . Si  $\text{AdjMat}[i][j] \& \& \text{AdjMat}[j][i]$  es verdadero, entonces los vértices  $i$  y  $j$  pertenecen al mismo SCC.

### Ejercicio 4.5.3.1

¿Cómo se puede encontrar la clausura transitiva de un grafo con  $V \leq 1000, E \leq 100000$ ? Supongamos que solo hay  $Q$  ( $1 \leq 100 \leq Q$ ) consultas de clausura transitiva en este problema, mediante esta pregunta: ¿está el vértice  $u$  conectado al vértice  $v$ , directa o indirectamente? ¿Qué ocurre si el grafo de entrada es *dirigido*? ¿Simplifica esta propiedad dirigida el problema?

### Ejercicio 4.5.3.2\*

Resuelve el problema del camino *maximin*, utilizando el algoritmo de Floyd Warshall.

### Ejercicio 4.5.3.3

El arbitraje es el intercambio de una divisa por otra, con la esperanza de obtener ventaja de pequeñas diferencia en las tasas de conversión entre diferentes divisas, para obtener un beneficio. Por ejemplo (UVa 436 - Arbitrage II): si 1,0 dólares de Estados Unidos (USD) se pueden cambiar por 0,5 libras esterlinas (GBP), 1,0 GBP puede comprar 10,0 francos franceses (FRF), y 1,0 FRF se pueden cambiar por 0,21 USD, un negociador de arbitraje puede empezar con 1,0 USD y comprar  $1,0 \times 0,5 \times 10,0 \times 0,21 = 1,05$  USD, obteniendo un beneficio del 5 %. Este problema es, en realidad, un problema de búsqueda de un *ciclo rentable*. Es muy similar al problema de búsqueda de ciclos con Floyd Warshall, tratado en esta sección. Resuelve este problema utilizando Floyd Warshall.

## Comentarios sobre caminos más cortos en concursos de programación

Los tres algoritmos tratados en las dos últimas secciones: Dijkstra, Bellman Ford y Floyd Warshall, se utilizan para resolver el *caso general* de los problemas de caminos más cortos (SSSP o APSP) en grafos ponderados. De estos tres, el de Bellman Ford en  $O(VE)$  es poco habitual en concursos de programación, dada su alta complejidad de tiempo. Solo resulta útil si el autor del problema nos da un grafo de ‘tamaño razonable’, con ciclos de peso negativo. En los casos generales, la variante de implementación de Dijkstra en  $O((V+E) \log V)$  (modificada por nosotros), es la mejor solución del problema de los SSSP para grafos ponderados ‘de tamaño razonable’, sin ciclos de peso negativo. Sin embargo, cuando el grafo dado es pequeño ( $V \leq 400$ ), lo que ocurre muchas veces, está claro que la opción de Floyd Warshall en  $O(V^3)$  es la mejor.

Una razón posible para justificar que el algoritmo de Floyd Warshall sea tan popular en concursos de programación es que, en ocasiones, el autor del problema incluye los caminos más cortos como *subproblemas* de un problema principal (mucho) más complejo. Para hacer que el problema sea resoluble durante el concurso, el autor establece, intencionadamente, un tamaño de entrada pequeño, para que el subproblema de caminos más cortos se pueda resolver con el Floyd Warshall de cuatro líneas (por ejemplo, UVa 10171, 10793, 11463). Un programador no competitivo tomará un camino más sinuoso para enfrentarse a este subproblema. Según nuestra experiencia, muchos problemas de caminos más cortos no son sobre grafos ponderados, que requieran los algoritmos de Dijkstra o Floyd Warshall. Si te fijas en los ejercicios de programación de la sección 4.4 (y, después, de la sección 8.2), podrás ver que muchos de ellos tratan con grafos no ponderados, que se pueden resolver con BFS (ver la sección 4.4.2).

También hemos observado que la tendencia actual, en relación a los problemas de caminos más cortos, implica un *modelado de grafos* cuidadoso (UVa 10067, 10801, 11367, 11492, 12160). Por lo tanto, para obtener buenos resultados en concursos de programación, debes asegurarte de que tienes las habilidad de detectar el grafo en el enunciado del problema. En este capítulo, hemos mostrado varios ejemplos de esa habilidad de modelado de grafos que, esperamos, sepas apreciar e, idealmente, asimilar como propia. En la sección 4.7.1 volveremos a visitar algunos de los problemas de caminos más cortos en grafos acíclicos dirigidos (DAG). Esta importante variante se puede resolver utilizando técnicas de programación dinámica (DP) genéricas, que se han tratado en la sección 3.5. Y veremos otro aspecto de la DP, como ‘algoritmo para DAG’.

En la tabla 4.4, tratamos la elección de algoritmos para SSSP y APSP, que ayudará al lector a decidir cuál utilizar en base a diferentes criterios de los grafos. La terminología utilizada es la siguiente: ‘mejor’ → el algoritmo más adecuado, ‘sí’ → un algoritmo correcto pero no el mejor, ‘malo’ → un algoritmo (muy) lento, WA → un algoritmo incorrecto, y ‘excesivo’ → un algoritmo correcto pero innecesario.

| Criterio del grafo     | BFS en<br>$O(V+E)$ | Dijkstra en<br>$O((V+E) \log V)$ | Bellman Ford en<br>$O(VE)$ | Floyd Warshall en<br>$O(V^3)$ |
|------------------------|--------------------|----------------------------------|----------------------------|-------------------------------|
| Tamaño máximo          | $V, E \leq 10M$    | $V, E \leq 300K$                 | $VE \leq 10M$              | $V \leq 400$                  |
| No ponderado           | Mejor              | Sí                               | Malo                       | Malo en general               |
| Ponderado              | WA                 | Mejor                            | Sí                         | Malo en general               |
| Peso negativo          | WA                 | Nuestra variante sí              | Sí                         | Malo en general               |
| Ciclo de peso negativo | No lo detecta      | No lo detecta                    | Lo detecta                 | Lo detecta                    |
| Grafo pequeño          | WA si es ponderado | Excesivo                         | Excesivo                   | Mejor                         |

Tabla 4.4: Tabla de decisión del algoritmo para SSSP/APSP

## Ejercicios de programación

### Ejercicios de programación para el algoritmo de Floyd Warshall:

#### Aplicación estándar de Floyd Warshall (para APSP o SSSP en grafos pequeños)

1. UVa 00341 - Non-Stop Travel  
(el grafo es pequeño)
2. UVa 00423 - MPI Maelstrom  
(el grafo es pequeño)
3. UVa 00567 - Risk  
(SSSP simple que se resuelve con BFS; pero el grafo es pequeño, así que es más fácil hacerlo con Floyd Warshall)
4. **UVa 00821 - Page Hopping \***  
(LA 5221 - WorldFinals Orlando00; uno de los problemas más fáciles de las finales del ICPC)
5. UVa 01233 - USHER  
(LA 4109 - Singapore07; se puede usar Floyd Warshall para encontrar el ciclo de coste mínimo en el grafo; el tamaño máximo del grafo de entrada es  $p \leq 500$ , pero es factible en el juez UVa)
6. UVa 01247 - Interstar Transport  
(LA 4524 - Hsinchu09; Floyd Warshall con modificaciones: preferencia por los caminos más cortos con menos vértices intermedios)
7. UVa 10171 - Meeting Prof. Miguel  
(fácil con información de los APSP)
8. *UVa 10354 - Avoiding Your Boss*  
(buscar y eliminar las aristas implicadas en los caminos más cortos del jefe; volver a ejecutar los caminos más cortos desde casa al mercado)
9. *UVa 10525 - New to Bangladesh?*  
(2 matrices de adyacencia: tiempo y longitud; Floyd Warshall modificado)
10. UVa 10724 - Road Construction  
(añadir una arista solo cambia algunas cosas)
11. UVa 10793 - The Orc Attack  
(Floyd Warshall simplifica este problema)
12. UVa 10803 - Thunder Mountain  
(el grafo es pequeño)
13. UVa 10947 - Bear with me, again..  
(el grafo es pequeño)
14. UVa 11015 - 05-32 Rendezvous  
(el grafo es pequeño)
15. **UVa 11463 - Commandos \***  
(la solución es fácil con información de los APSP)
16. *UVa 12319 - Edgetown's Traffic Jams*  
(Floyd Warshall dos veces y comparar)

#### Variantes

1. **UVa 00104 - Arbitrage \***  
(pequeño problema de arbitraje que se resuelve con Floyd Warshall)
2. *UVa 00125 - Numbering Paths*  
(Floyd Warshall modificado)
3. UVa 00186 - Trip Routing  
(el grafo es pequeño; mostrar camino)
4. *UVa 00274 - Cat and Mouse*  
(variante del problema de clausura transitiva)
5. UVa 00436 - Arbitrage (II)  
(otro problema de arbitraje)
6. **UVa 00334 - Identifying Concurrent ... \***  
(clausura transitiva y más)
7. UVa 00869 - Airline Comparison  
(ejecutar Warshall dos veces; comparar AdjMatrices)
8. *UVa 00925 - No more prerequisites ...*  
(clausura transitiva y más)
9. **UVa 01056 - Degrees of Separation \***  
(LA 3569 - WorldFinals SanAntonio06; buscar el diámetro de un grafo pequeño con Floyd Warshall)
10. *UVa 01198 - Geodetic Set Problem*  
(LA 2818 - Kaohsiung03; clausura transitiva y más)
11. UVa 11047 - The Scrooge Co Problem  
(mostrar el camino; especial: si origen = destino, escribirlo dos veces)

## Perfiles de los inventores de algoritmos

**Robert W. Floyd** (1936-2001) fue un eminente científico de la computación estadounidense. Las aportaciones de Floyd incluyen el diseño del **algoritmo de Floyd** [19], que encuentra de forma eficiente todos los caminos más cortos en un grafo. Floyd trabajó mano a mano con Donald Ervin Knuth, siendo, concretamente, el principal revisor de la obra fundamental de Knuth, ‘The Art of Computer Programming’, y es la persona más citada en ese libro.

**Stephen Warshall** (1935-2006) fue un científico de la computación, que inventó el **algoritmo de clausura transitiva**, conocido ahora como **algoritmo de Warshall** [70]. Este algoritmo fue bautizado, posteriormente, como Floyd Warshall, pues Floyd inventó, en esencia y de forma independiente, el mismo algoritmo.

**Jack R. Edmonds** (nacido en 1934) es un matemático. Con Richard Karp, inventó el **algoritmo Edmonds Karp**, que calcula el flujo máximo en una red de flujo en  $O(VE^2)$  [14]. También inventó un algoritmo para el MST en grafos dirigidos (problema de arborescencia). Este fue propuesto inicialmente, de forma independiente, por Chu y Liu (1965) y, posteriormente, por Edmonds (1967), de ahí que se le llame **algoritmo de Chu Liu/Edmonds** [6]. Sin embargo, su aportación más importante es, probablemente, el **algoritmo de emparejamiento** de Edmonds, uno de los artículos de ciencias de la computación más citados [13].

**Richard Manning Karp** (nacido en 1935) es un científico de la computación. Ha realizado muchos descubrimientos importantes para las ciencias de la computación, en el ámbito de los algoritmos de combinatoria. En 1971, publicó, junto a Edmonds, el **algoritmo Edmonds Karp**, para resolver el problema del flujo máximo [14]. En 1973, John Hopcroft y él publicaron el **algoritmo Hopcroft Karp**, que sigue siendo el método más rápido conocido para encontrar el emparejamiento bipartito de cardinalidad máxima [28].

## 4.6 Flujo de red

### 4.6.1 Introducción y motivación

Problema de referencia: imagina un grafo conexo, ponderado (con enteros) y dirigido<sup>12</sup>, como una red de tuberías, donde las aristas son las tuberías y los vértices son los puntos de bifurcación. Cada arista tiene un peso equivalente a la capacidad de la tubería. También existen dos vértices especiales: el origen  $s$  y el desagüe  $t$ . ¿Cuál es el flujo (caudal) máximo desde el origen  $s$  hasta el desagüe  $t$  en este grafo (imaginemos que hay agua fluyendo por la red de tuberías y que queremos saber el volumen máximo de agua que puede pasar por ella)? Este problema es conocido como el de flujo máximo, un problema de flujo de red. La figura 4.24 muestra una ilustración del flujo máximo, y hay más detalles en la sección 4.6.2.

### 4.6.2 Método de Ford Fulkerson

Una solución al problema del flujo máximo es el método Ford Fulkerson, inventado por el mismo Lester Randolph *Ford*, Jr. del algoritmo de Bellman Ford, y por Delbert Ray *Fulkerson*.

<sup>12</sup>Una arista ponderada no dirigida en un grafo no dirigido se puede transformar en dos aristas dirigidas con el mismo peso, pero de direcciones opuestas.

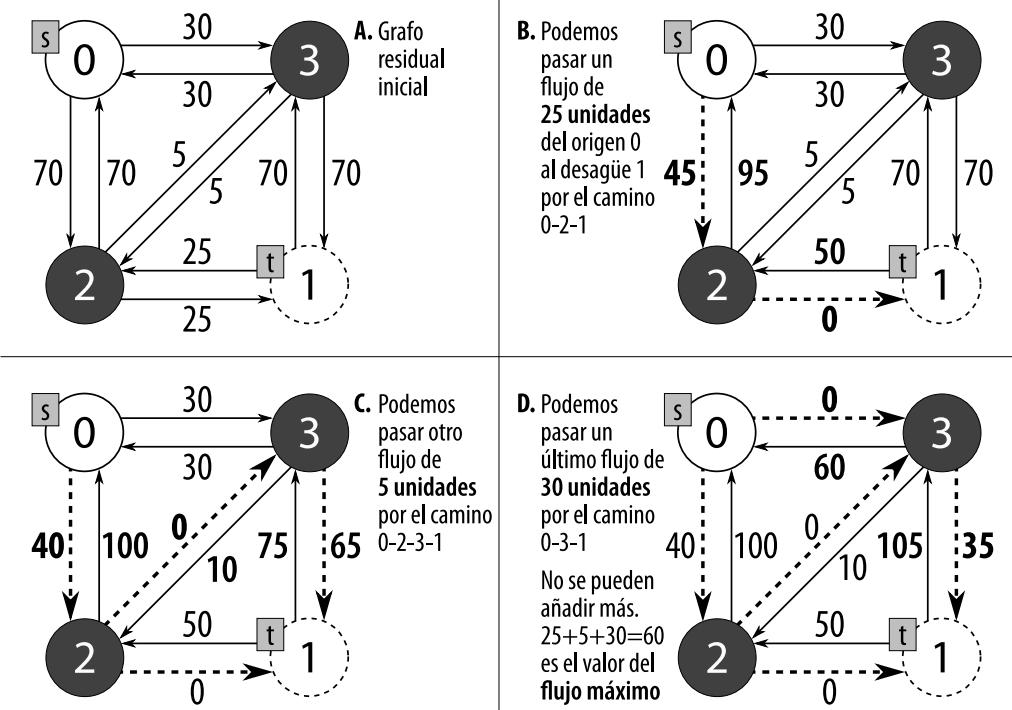


Figura 4.24: Ilustración del flujo máximo (UVa 820 [49] - problema E en la final ICPC 2000)

```

grafo residual dirigido con capacidad de arista = pesos grafo original
mf = 0 // es un algoritmo iterativo, mf significa flujo_máximo
while (existe un aumento de camino p desde s a t) {
 // p es un camino desde s a t que pasa por aristas positivas en grafo
 aumentar/enviar flujo f junto al camino p (s -> ... -> i -> j -> ... t)
 1. buscar f, el peso de arista mínimo en el camino p
 2. reducir en f la capacidad de aristas adelante (i -> j) en camino p
 3. aumentar en f capacidad de aristas inversas (j -> i) en camino p
 mf += f // podemos enviar flujo de tamaño f de s a t, aumentar mf
}
mostrar mf
 // es el valor del flujo máximo

```

El método Ford Fulkerson es un algoritmo iterativo que encuentra repetidamente el aumento de camino  $p$ : un camino desde el origen  $s$  al desagüe  $t$ , que pasa por las aristas de peso positivo en el grafo residual<sup>13</sup>. Después de encontrar un aumento de camino  $p$ , que tenga  $f$  como peso mínimo de la arista en el camino  $p$  (la arista que es el cuello de botella en este camino), el método Ford Fulkerson realizará dos pasos importantes: disminuir/aumentar la capacidad de

<sup>13</sup>Utilizamos el término ‘grafo residual’ porque, inicialmente, el peso de cada arista  $res[i][j]$  es el mismo que la capacidad original de la arista  $(i, j)$ , en el grafo original. Si esta arista  $(i, j)$  se utiliza en un aumento de camino y, por ella, pasa un flujo de peso  $f \leq res[i][j]$  (un flujo no puede exceder esta capacidad), entonces la capacidad remanente (o residual) de la arista  $(i, j)$  será  $res[i][j] - f$ .

las aristas adelante ( $i \rightarrow j$ )/inversas ( $j \rightarrow i$ ) del camino  $p$  en  $f$ , respectivamente. El método repetirá este proceso hasta que no queden más aumentos de camino posibles, desde el origen  $s$  hasta el desagüe  $t$ , lo que implica que el flujo total hasta ese momento es el flujo máximo. Una vez explicado esto, podemos volver a ver la figura 4.24.

La razón para disminuir la capacidad de la siguiente arista es evidente. Al enviar un flujo a través del aumento de camino  $p$ , reduciremos la capacidad restante (residual) de las (siguientes) aristas utilizadas en  $p$ . La razón para aumentar la capacidad de las aristas inversas puede no ser tan obvia, pero este paso es importante para la correcta ejecución del método Ford Fulkerson. Al aumentar la capacidad de una arista inversa ( $j \rightarrow i$ ), el método permite la *iteración (flujo) futura* para cancelar (parte de) la capacidad utilizada por una arista adelante ( $i \rightarrow j$ ), que haya sido utilizada incorrectamente en algún flujo anterior.

Hay varias formas para encontrar un aumento de camino  $s-t$  en el pseudocódigo anterior, cada una con un comportamiento diferente. En esta sección vemos dos de ellas, usando DFS o BFS.

La implementación del método Ford Fulkerson, utilizando DFS, se puede ejecutar en  $O(|f^*|E)$ , donde  $|f^*$  es el valor  $\text{mf}$  del flujo máximo. Esto se debe a que podemos tener un grafo como el de la figura 4.25. Después, nos podemos encontrar en una situación en la que dos aumentos de camino,  $s \rightarrow a \rightarrow b \rightarrow t$  y  $s \rightarrow b \rightarrow a \rightarrow t$ , únicamente reduzcan la capacidad de la arista (adelante<sup>14</sup>) en el camino por 1. En el peor de los casos, esto se repite  $|f^*$  veces (son 200 veces en la figura 4.25). Dado que la DFS se ejecuta en  $O(E)$  en un grafo de flujo<sup>15</sup>, la complejidad de tiempo total es  $O(|f^*|E)$ . Esta incertidumbre no es deseable en un concurso de programación, ya que el autor del problema podría haber utilizando un valor de  $|f^*$  (muy) grande.

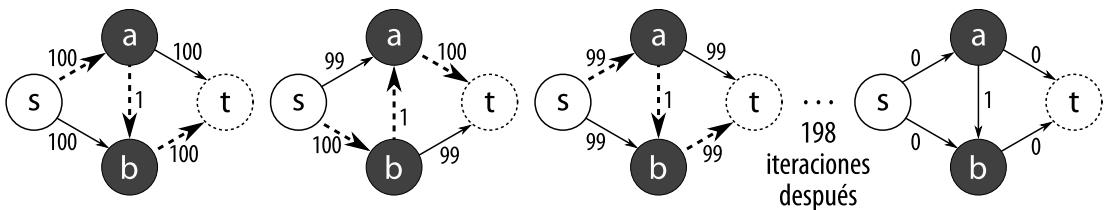


Figura 4.25: El método Ford Fulkerson puede ser lento si se implementa con DFS

### 4.6.3 Algoritmo de Edmonds Karp

Una implementación mejor del método Ford Fulkerson consiste en utilizar BFS para encontrar el camino más corto en términos de número de capas/saltos entre  $s$  y  $t$ . Este algoritmo fue descubierto por Jack *Edmonds* y Richard Manning *Karp*, por lo que lleva el nombre de algoritmo de Edmonds Karp [14]. Se ejecuta en  $O(VE^2)$ , ya que se puede desmostrar que tras  $O(VE)$  iteraciones de BFS, todos los aumentos de camino han quedado agotados. Los lectores curiosos pueden consultar referencias como [14, 7] para aprender más sobre esta demostración. Una BFS se ejecuta, en un grafo de flujo, en  $O(E)$ , para una complejidad de tiempo total de  $O(VE^2)$ . El algoritmo de Edmonds Karp solo necesita, en la figura 4.25, dos caminos  $s-t$ :  $s \rightarrow a \rightarrow t$

<sup>14</sup>Después de enviar el flujo  $s \rightarrow a \rightarrow b \rightarrow t$ , la arista adelante  $a \rightarrow b$  es sustituida por la inversa  $b \rightarrow a$ , y así sucesivamente. Si esto no sucede, el valor del flujo máximo será solo  $1 + 99 + 99 = 199$ , en vez de 200, lo que es erróneo.

<sup>15</sup>El número de aristas en un grafo de flujo debe ser  $E \geq V - 1$ , para asegurar que el flujo  $\exists \geq 1$   $s-t$ . Esto implica que tanto DFS como BFS, utilizando una lista de adyacencia, se ejecutan en  $O(E)$ , en vez de  $O(V+E)$ .

(2 saltos, envía un flujo de 100 unidades) y  $s \rightarrow b \rightarrow t$  (2 saltos, otras 100 unidades). Así, no pierde tiempo enviando el flujo a través de los caminos más largos (3 saltos):  $s \rightarrow a \rightarrow b \rightarrow t$  (o  $s \rightarrow b \rightarrow a \rightarrow t$ ).

Implementar por primera vez el algoritmo de Edmonds Karp puede resultar un desafío a los nuevos programadores. En esta sección incluimos nuestro código más sencillo de Edmonds Karp, que utiliza *solo* una matriz de adyacencia llamada `res`, con tamaño  $O(V^2)$ , para almacenar la capacidad residual de cada arista. Esta versión, que se ejecuta en  $O(VE)$  iteraciones de la  $BFS \times O(V^2)$  por  $BFS$ , debido a la matriz de adyacencia =  $O(V^3E)$ , es lo suficientemente rápida como para resolver *algunos* problemas (pequeños) de flujo máximo.

```

1 int res[MAX_V][MAX_V], mf, f, s, t; // variables globales
2 vi p; // p guarda el árbol de expansión de BFS desde s
3
4 void augment(int v, int minEdge) { //recorrer árbol expansión BFS desde s->t
5 if (v == s) { f = minEdge; return; } // guardar minEdge en var global f
6 else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v]));
7 res[p[v]][v] -= f; res[v][p[v]] += f; }
8
9 // dentro de int main(): fijar 'res', 's' y 't' con valores apropiados
10 mf = 0; // mf significa flujo máximo
11 while (1) { // algoritmo de Edmonds Karp en $O(VE^2)$ (en realidad $O(V^3 E)$)
12 f = 0;
13 // ejecutar BFS, comparar con BFS original mostrada en sección 4.2.2
14 vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
15 p.assign(MAX_V, -1); // guardar árbol de expansión BFS, desde s a t
16 while (!q.empty()) {
17 int u = q.front(); q.pop();
18 if (u == t) break; // detener BFS si hemos llegado al desagüe t
19 for (int v = 0; v < MAX_V; v++) // nota: esta parte es lenta
20 if (res[u][v] > 0 && dist[v] == INF)
21 dist[v] = dist[u]+1, q.push(v), p[v] = u; // 3 líneas en 1
22 }
23 augment(t, INF); // buscar el peso de arista mínimo 'f' en este camino
24 if (f == 0) break; // no podemos enviar más flujo ('f' = 0), terminar
25 mf += f; // podemos enviar un flujo, aumentar flujo máximo
26 }
27 printf("%d\n", mf); // este es el valor del flujo máximo

```



[visualgo.net/maxflow](http://visualgo.net/maxflow)



ch4\_08\_edmonds\_karp.cpp



ch4\_08\_edmonds\_karp.java

### Ejercicio 4.6.3.1

Antes de continuar, responde a la pregunta de la figura 4.26.

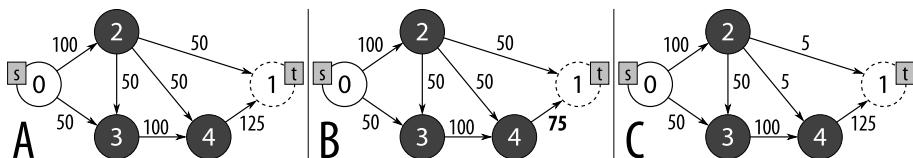


Figura 4.26: ¿Cuál es el valor del flujo máximo de estos tres grafos residuales?

### Ejercicio 4.6.3.2

La debilidad principal del sencillo código que hemos incluido en esta sección, radica en que la enumeración de los vecinos de un vértice consume  $O(V)$ , en vez de  $O(k)$  (donde  $k$  es el número de vecinos de ese vértice). La otra debilidad (aunque no es significativa), se encuentra en que no necesitamos `vi.dist`, ya que `bitset` (para etiquetar si se ha visitado un vértice) es suficiente. Modifica el código de Edmonds Karp mostrado para lograr una complejidad de tiempo de  $O(VE^2)$ .

### Ejercicio 4.6.3.3\*

Una implementación todavía mejor del algoritmo de Edmonds Karp, consiste en evitar utilizar una matriz de adyacencia  $O(V^2)$  para almacenar la capacidad residual de cada arista. La mejor opción para almacenar, tanto la capacidad actual como el flujo real (no solo el residual) de cada arista, es una lista de adyacencia y aristas modificada, en  $O(V+E)$ . De esta forma, tenemos 3 datos de cada arista: su capacidad original, su flujo actual  $y$ , y, de ahí, podemos deducir el flujo residual, restando el flujo actual de la capacidad original. Vuelve a modificar la implementación. ¿Cómo gestionamos el flujo inverso de forma eficiente?

## 4.6.4 Modelado de grafos de flujo - parte 1

Con el código ya visto para el algoritmo de Edmonds Karp, resolver un problema de flujo de red (básico/estándar) es, ahora, más sencillo. Es cuestión de:

1. Reconocer que el problema es, efectivamente, de flujo de red (esta habilidad irá mejorando a medida que vayas resolviendo más problemas del mismo estilo).
2. Construir el grafo de flujo adecuado (es decir, si utilizas el código que hemos mostrado, inicializar la matriz residual `res` y establecer los valores apropiados para  $s$  y  $t$ ).
3. Ejecutar el código de Edmonds Karp en ese grafo de flujo.

En esta subsección, hemos mostrado un ejemplo de *modelado* del grafo de flujo (residual) del problema UVa 259 - Software Allocation<sup>16</sup>. El enunciado resumido de este problema es el siguiente: tienes 26 aplicaciones (etiquetadas de la ‘A’ a la ‘Z’), hasta 10 ordenadores (numerados del 0 al 9), el número de personas que han venido a utilizar cada aplicación ese día (un entero positivo de un dígito, o [1..9]), la lista de ordenadores en los que se puede ejecutar una aplicación en particular y el hecho de que cada ordenador solo puede ejecutar una aplicación ese día. La tarea consiste en determinar si se puede realizar una asignación (es decir, un *emparejamiento*) de aplicaciones a ordenadores válidos y, si es así, generar esa posible asignación. En caso contrario, basta con escribir un signo de exclamación ‘!’.

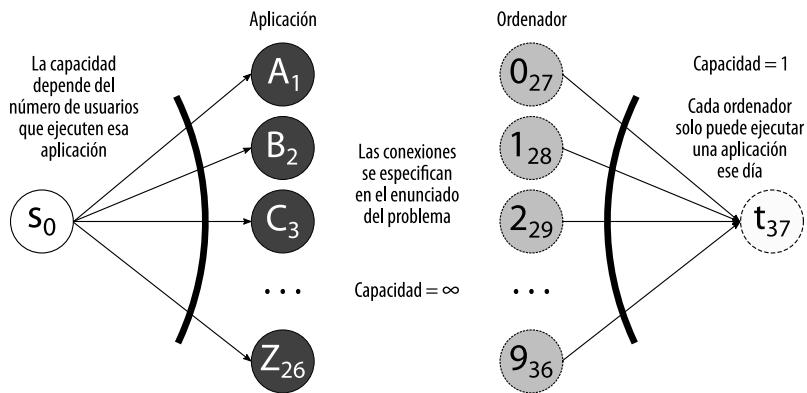


Figura 4.27: Grafo residual de UVa 259 [49]

En la figura 4.27 se muestra una formulación de un grafo de flujo (bipartito). Indexamos los vértices como [0..37], ya que hay  $26+10+2$  vértices especiales = 38 vértices. Se asigna el índice 0 al origen  $s$ , las 26 posibles aplicaciones son [1..26], los 10 posibles ordenadores son [27..36] y, finalmente, el desagüe  $t$  obtiene el índice 37.

Después, vinculamos las aplicaciones a los ordenadores válidos, como se indica en la descripción del problema. Conectamos el origen  $s$  a todas las aplicaciones y conectamos todos los ordenadores al desagüe  $t$ . Todas las aristas de este grafo de flujo son *dirigidas*. El problema indica que puede haber *más de un* usuario (digamos que  $X$ ) que puede utilizar una aplicación concreta  $A$  en un día dado. Por lo tanto, establecemos el peso de la arista dirigida (capacidad) del origen  $s$  a una aplicación en particular  $A$  como  $X$ . El problema también indica que cada ordenador solo se puede usar una vez. Así, fijamos el peso de la arista dirigida de cada ordenador  $B$  al desagüe  $t$  como 1. El peso de la arista entre aplicaciones y ordenadores válidos será  $\infty$ . Con esta disposición, si existe un flujo desde una aplicación  $A$  a un ordenador  $B$  y, de ahí, al desagüe  $t$ , habrá *una asignación* (*un emparejamiento*) entre esa aplicación  $A$  y el ordenador  $B$ .

Una vez que tengamos este grafo de flujo, podemos ejecutar sobre él nuestra implementación del algoritmo de Edmonds Karp (vista antes), para obtener el flujo máximo  $mf$ . Si  $mf$  es igual al número de aplicaciones ejecutadas ese día, significa que tenemos una solución, es decir, si hay  $X$  usuarios ejecutando la aplicación  $A$ , entonces el algoritmo de Edmonds Karp debe encontrar  $X$  caminos diferentes (emparejamientos) desde  $A$  hasta el desagüe  $t$  (igual para el resto).

<sup>16</sup>En realidad, este problema tiene un tamaño de entrada pequeño (solo hay  $26+10 = 36$  vértices, más otros 2: el origen y el desagüe), lo que hace que se pueda resolver mediante *backtracking* recursivo (ver la sección 3.2). Si el grafo dado implica entorno a [100..200] vértices, el flujo máximo es la solución buscada. El nombre de este problema es ‘problema de asignación’, o emparejamiento bipartito (especial) con capacidad.

Las asignaciones aplicación → ordenador se pueden encontrar fácilmente, comprobando las aristas inversas entre los ordenadores (vértices 27-36) y las aplicaciones (vértices 1-26). Una arista inversa (ordenador → aplicación) en la matriz residual  $\text{res}$ , contendrá el valor +1, si la arista directa correspondiente (aplicación → ordenador) está seleccionada en los caminos que aportan al flujo máximo  $\text{mf}$ . Esta es también la razón por la que comenzamos el grafo de flujo utilizando solo aristas *dirigidas* de las aplicaciones a los ordenadores.

### Ejercicio 4.6.4.1

¿Por qué utilizamos  $\infty$  para los pesos (capacidades) de las aristas dirigidas de las aplicaciones a los ordenadores? ¿Podríamos utilizar la capacidad 1 en vez de  $\infty$ ?

### Ejercicio 4.6.4.2\*

Se puede resolver el problema general de asignación (emparejamiento bipartito con capacidad, no solo este problema de ejemplo UVa 259) con el algoritmo estándar de emparejamiento bipartito de cardinalidad máxima (MCBM), que veremos en la sección 4.7.4? Si es posible, indica cuál es la mejor solución. Si no es posible, explica el motivo.

## 4.6.5 Otras aplicaciones

Hay otras variantes interesantes de los problemas que implican flujo de red. A continuación, veremos tres más, mientras que dejaremos otras para las secciones 4.7.4 (grafo bipartito), 9.22 y 9.23. Algunos de los trucos que veremos aquí son aplicables a otros problemas de grafos.

### Corte mínimo

Vamos a definir un corte  $s - t$   $C = (\text{componente-}S, \text{componente-}T)$  como una partición de  $V \in G$ , de forma que el origen  $s \in \text{componente-}S$  y el desagüe  $t \in \text{componente-}T$ . Definimos también un *conjunto de cortes* de  $C$  como el conjunto  $\{(u, v) \in E \mid u \in \text{componente-}S, v \in \text{componente-}T\}$ , de forma que, si se eliminan todas las aristas del conjunto de cortes de  $C$ , el flujo máximo de  $s$  a  $t$  sea 0 (es decir,  $s$  y  $t$  estén desconectados). El coste de un corte  $s - t$   $C$ , se define como la suma de la capacidad de las aristas en el conjunto de cortes de  $C$ . El problema del corte mínimo consiste en minimizar la capacidad de un corte  $s - t$ . Este problema es más generalista que la búsqueda de puentes (ver la sección 4.2.1), es decir, en este caso podemos cortar *más* de una arista, y queremos hacerlo de la forma menos costosa. Como con los puentes, el corte mínimo tiene aplicaciones en el ‘sabotaje’ de redes, por ejemplo, un problema de corte mínimo puro es UVa 10480 - Sabotage.

La solución es sencilla: el subproducto del cálculo del flujo máximo es el corte mínimo. Veamos, nuevamente, la figura 4.24.D. Una vez ha finalizado el algoritmo del flujo máximo, ejecutamos un nuevo recorrido del grafo (DFS/BFS) desde el origen  $s$ . Todos los vértices alcanzables por el origen  $s$ , utilizando aristas de peso positivo en el grafo residual, pertenecen al componente- $S$  (es decir, vértices 0 y 2). Todos los vértices no alcanzables pertenecen al componente- $T$  (vértices 1 y

3). Todas las aristas que conectan el componente- $S$  con el componente- $T$ , pertenecen al conjunto de cortes de  $C$  (aristas 0-3 (capacidad 30/flujo 30/residual 0), 2-3 (5/5/0) y 2-1 (25/25/0), en este caso). El valor del corte mínimo es  $30+5+25 = 60 =$  valor del flujo máximo mf. Este es el mínimo de todos los cortes  $s - t$  posibles.

### Multiorigen/multidesagüe

En ocasiones, podemos tener más de un origen y/o más de un desagüe. Sin embargo, esta variante no es más difícil que el problema de flujo de red original, con origen y desagüe únicos. Creamos un super origen  $ss$  y un super desagüe  $st$ . Conectamos  $ss$  con todos los  $s$  con capacidad infinita, y todos los  $t$  con  $st$ , también con capacidad infinita, y ejecutamos el algoritmo de Edmonds Karp con normalidad. Hemos visto esta variante en el **ejercicio 4.4.2.1**.

### Capacidad de los vértices

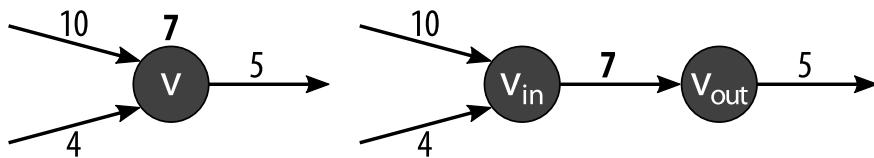


Figura 4.28: Técnica de división de vértices

También existe una variante de flujo de red en la que las capacidades no vienen determinadas solo por las aristas, sino *también por los vértices*. Para resolverla, podemos utilizar la técnica de la *división de vértices* que, por desgracia, *duplica* el número de vértices del grafo de flujo. Un grafo ponderado con peso en los vértices se puede convertir en otro más cómodo, *sin* peso en los vértices, dividiendo cada vértice ponderado  $v$  en  $v_{in}$  y  $v_{out}$ , reasignando las aristas entrantes y salientes a  $v_{in}/v_{out}$ , respectivamente, y, por último, estableciendo el peso del vértice original  $v$  como peso de la arista  $v_{in} \rightarrow v_{out}$ . En la figura 4.28 hay una ilustración de ello. Con todos los pesos ya asignados a las aristas, se ejecuta el algoritmo de Edmonds Karp con normalidad.

### 4.6.6 Modelado de grafos de flujo - parte 2

La parte más difícil de tratar con el problema del flujo de red, es modelar el grafo de flujo (asumiendo que tenemos un buen código para el flujo máximo ya escrito). En la sección 4.6.4, hemos visto un modelado de ejemplo que sirve para enfrentarse al problema de asignación (o emparejamiento bipartito con capacidad). Ahora veremos otro modelado de grafo de flujo (más complicado), para el problema UVa 11380 - Down Went The Titanic. Un consejo antes de continuar leyendo: no te limites a memorizar la solución, trata de entender los pasos clave necesarios para obtener el grafo de flujo requerido.

En la figura 4.29, encontramos cuatro pequeños casos de prueba para UVa 11380. Tenemos una pequeña rejilla bidimensional, que contiene los cinco caracteres de la tabla 4.5. Queremos poner la mayor cantidad de '\*' (gente) en los diferentes lugares seguros: las '#' (tablas grandes). Las flechas sólidas y punteadas de la figura 4.29 indican la respuesta.

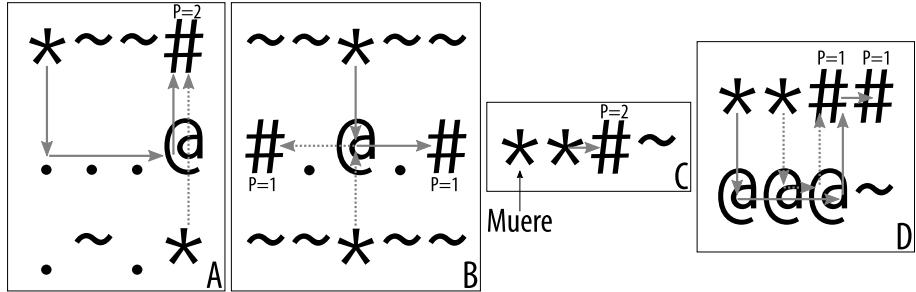


Figura 4.29: Algunos casos de prueba de UVa 11380

| Símbolo | Significado                | Veces usado | Capacidad |
|---------|----------------------------|-------------|-----------|
| *       | Gente en el hielo flotante | 1           | 1         |
| ~       | Agua congelada             | 0           | 0         |
| .       | Hielo flotante             | 1           | 1         |
| @       | Iceberg grande             | $\infty$    | 1         |
| #       | Tabla grande               | $\infty$    | $P$       |

Tabla 4.5: Caracteres utilizados en UVa 11380

Para modelar el grafo de flujo, podemos utilizar el siguiente proceso mental. En la figura 4.30.A, comenzamos conectando las celdas que no son ‘~’, con una capacidad grande (1000 es suficiente para este problema). Esto describe los posibles movimientos en la rejilla. En la figura 4.30.B, establecemos la capacidad de los vértices de las casillas ‘\*’ y ‘.’ a 1, para indicar que solo se pueden utilizar *una vez*. Después, establecemos la capacidad de los vértices de ‘@’ y ‘#’ a un valor grande (nuevamente 1000 es suficiente en este caso) para indicar que se pueden utilizar *varias veces*. En la figura 4.30.C, creamos los vértices de origen  $s$  y desagüe  $t$ . El origen  $s$  está conectado a todas las casillas ‘\*’, con capacidad 1, de la rejilla, para indicar que hay una persona a la que salvar. Todas las casillas ‘#’ de la rejilla están conectadas al desagüe  $t$ , con capacidad  $P$ , para indicar que la tabla grande se puede utilizar  $P$  veces. En este punto, la respuesta solicitada (el número de supervivientes), equivale al valor del flujo máximo entre el origen  $s$  y el desagüe  $t$  en este grafo de flujo. Como el grafo de flujo utiliza las capacidades de los vértices, tenemos que utilizar la técnica de *división de vértices* que hemos visto antes.

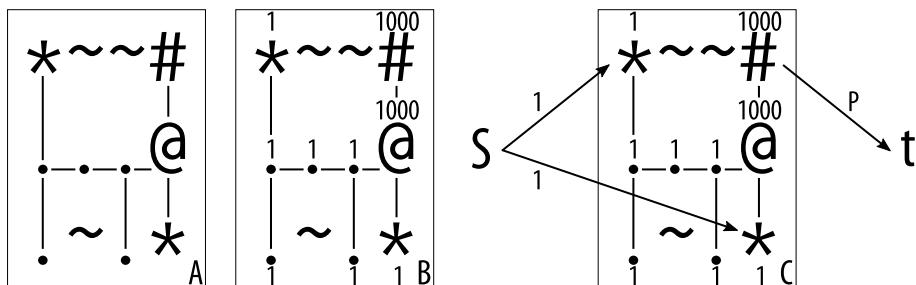


Figura 4.30: Modelado del grafo de flujo

### Ejercicio 4.6.6.1\*

¿Es el algoritmo de Edmonds Karp en  $O(VE^2)$ , lo suficientemente rápido para calcular el valor del flujo máximo en el grafo de flujo más grande posible en el problema UVa 11380: rejilla de  $30 \times 30$  y  $P = 10$ ? ¿Por qué?

### Comentarios sobre flujo de red en concursos de programación

En el momento de publicación de este libro, cuando un problema de flujo de red, normalmente máximo, aparece en un concurso de programación, *suele* ser uno de los problemas ‘decisivos’. Muchos de los problemas de grafos interesantes del ICPC están escritos de forma que, a primera vista, no parezcan de flujo de red. Lo más difícil para el concursante, es darse cuenta de que el problema subyacente es, de hecho, de flujo de red y, a partir de ahí, pueda modelar el grafo de flujo correctamente. Esta habilidad clave se desarrolla mediante la práctica.

Para ahorrar tiempo, siempre escaso, en la programación (y, especialmente, en la depuración) del código, relativamente largo, del flujo máximo, sugerimos que, en un equipo del ICPC, uno de los miembros dedique un esfuerzo significativo en preparar un buen código de flujo máximo (quizá la implementación del algoritmo de Dinic, en la sección 9.7) y pruebe a resolver varios problemas de flujo de red, de entre los disponibles en muchos jueces en línea, para mejorar su confianza con los flujos de red y sus variantes. En la lista de ejercicios de programación de esta sección, hemos incluido algunos sencillos de flujo de red máximo, emparejamiento bipartito con capacidad (problema de asignación), corte mínimo y flujo de red con capacidades en los vértices. Intenta resolver todos los que puedas.

En la sección 4.7.4, veremos que el problema clásico del emparejamiento bipartito de cardinalidad máxima (MCBM) se puede resolver también mediante flujo máximo, aunque existe un algoritmo más sencillo y específico. Más adelante, en el capítulo 9, trataremos algunos problemas más difíciles, relacionados con flujo de red como, por ejemplo, un algoritmo más rápido para el flujo máximo (sección 9.7), problemas de caminos independientes y de aristas disjuntas (sección 9.13), el problema del conjunto independiente *ponderado* máximo en grafos bipartitos (sección 9.22) y el problema del flujo (máximo) de coste mínimo (sección 9.23).

En la IOI, el flujo de red (y sus variantes) no se encuentra incluido en el temario de 2009 [20]. Por lo tanto, los concursantes de la IOI podrían ignorar esta sección. Sin embargo, creemos que es una buena idea que se adelanten a las materias más avanzadas, para mejorar sus capacidades con problemas de grafos.

### Ejercicios de programación

#### Ejercicios de programación relativos a flujo de red:

##### Problema de flujo de red máximo estándar (Edmonds Karp)

1. [UVa 00259 - Software Allocation \\*](#) (tratado en esta sección)
2. [UVa 00820 - Internet Bandwidth \\*](#) (LA 5220 - final Orlando00; flujo máximo básico; visto en esta sección)
3. UVa 10092 - The Problem with the ... (asignación; emparejamiento con capacidad; similar a UVa 259)
4. UVa 10511 - Councilling (emparejamiento; flujo máximo; mostrar la asignación)

|                                                 |                                                                                                                                                                                                                                                                                                    |
|-------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5. UVa 10779 - Collectors Problem               | (modelado de flujo máximo no directo; la idea es construir un grafo de flujo en el que cada aumento de camino corresponda a una serie de pegatinas repetidas, comenzando con Bob regalando una de ellas, y finalizando al recibir otra nueva; repetir hasta que no sea posible)                    |
| 6. UVa 11045 - My T-Shirt Suits Me              | (problema de asignación; emparejamiento con capacidad; similar a UVa 259, 10092 y 12873; en realidad la restricción de la entrada es suficientemente pequeña como para usar <i>backtracking</i> recursivo)                                                                                         |
| 7. <u>UVa 11167 - Monkeys in the Emei ... *</u> | (modelado de flujo máximo; hay muchas aristas en el grafo de flujo; por lo tanto, es mejor comprimir las aristas de capacidad 1, siempre que sea posible; usar el algoritmo de flujo máximo de Dinic en $O(V^2 E)$ para que la gran cantidad de aristas no penalice el rendimiento de la solución) |
| 8. UVa 11418 - Clever Naming Patterns           | (dos capas de emparejamiento de grafos (en realidad no es emparejamiento bipartito); usar solución de flujo máximo)                                                                                                                                                                                |

### Variantes

|                                               |                                                                      |
|-----------------------------------------------|----------------------------------------------------------------------|
| 1. UVa 10330 - Power Transmission             | (flujo máximo; capacidades de vértices)                              |
| 2. UVa 10480 - Sabotage                       | (problema directo de corte mínimo)                                   |
| 3. <u>UVa 11380 - Down Went The Titanic *</u> | (modelado flujo máximo con capacidad de vértices; similar UVa 12125) |
| 4. <u>UVa 11506 - Angry Programmer *</u>      | (corte mínimo con capacidades de vértices)                           |
| 5. <u>UVa 12125 - March of the Penguins *</u> | (modelado flujo máximo con capacidad de vértices; similar UVa 11380) |

## 4.7 Grafos especiales

Existen algoritmos polinómicos más sencillos y rápidos para algunos problemas de grafos básicos, si el grafo dado es *especial*. Con nuestra experiencia, hemos identificados los siguientes grafos especiales, que suelen aparecer en concursos de programación: **grafo acíclico dirigido (DAG)**, **árbol**, **grafo euleriano** y **grafo bipartito**. Los autores de los problemas pueden obligar a los concursantes a utilizar algoritmos específicos para estos grafos especiales, al proporcionar tamaños de entrada grandes, en los que un algoritmo generalista, aunque correcto, obtendría un veredicto de tiempo límite superado (TLE) (ver la información sobre esto contenida en [21]). En esta sección, trataremos algunos problemas de grafos populares que utilizan estos grafos especiales (ver la figura 4.31), muchos de los cuales ya han sido tratados como grafos generalistas. Por el momento, los grafos bipartitos (sección 4.7.4) siguen excluidos del temario de la IOI [20].

### 4.7.1 Grafo acíclico dirigido

Un grafo acíclico dirigido (DAG) es un grafo especial, que cuenta con las características de que es dirigido y no tiene ciclos. Un DAG garantiza la ausencia de ciclos *por definición*. Esto hace que los problemas que se pueden modelar como un DAG sean muy apropiados para una solución con técnicas de programación dinámica (ver la sección 3.5). Después de todo, la recurrencia en la DP debe ser *acíclica*. Podemos ver los estados de la DP como vértices de un DAG implícito, y las

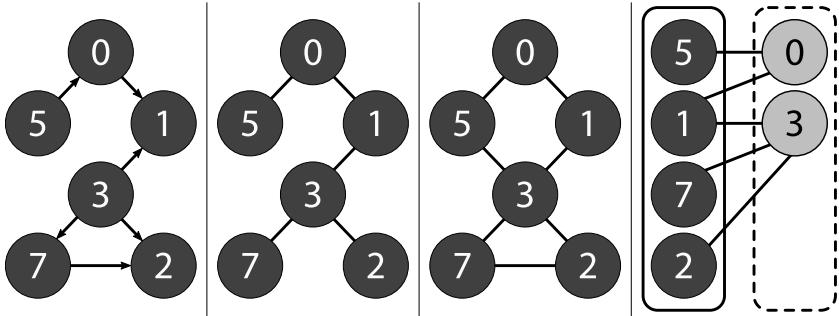


Figura 4.31: Grafos especiales (izquierda a derecha): DAG, árbol, euleriano, bipartito

transiciones acíclicas entre los estados de DP como aristas dirigidas del mismo. La ordenación topológica de ese DAG (ver la sección 4.2.5), permite que cada problema superpuesto (subgrafo del DAG) sea procesado una sola vez.

#### Caminos más cortos/largos (de origen único) en un DAG

El problema de los caminos más cortos de origen único (SSSP) se vuelve mucho más sencillo si el grafo dado es un DAG. Esto se debe a que, para un DAG, existe, al menos, un orden topológico. Podemos utilizar un algoritmo de ordenación topológica en  $O(V+E)$ , como el de la sección 4.2.1, para encontrar ese orden y, después, relajar las aristas salientes de los vértices, de acuerdo a su orden. El orden topológico nos asegura que, si tenemos un vértice  $b$  que tiene una arista entrante desde el vértice  $a$ , el vértice  $b$  será relajado *después* de que el  $a$  haya obtenido el valor de distancia más corta correcto. De esta forma, la propagación de los valores de distancia más corta será correcta con una sola pasada lineal en  $O(V+E)$ . Esta es, también, la esencia del principio de la programación dinámica, de evitar volver a calcular los problemas superpuestos, como ya hemos visto en la sección 3.5. Cuando calculamos la DP de abajo hacia arriba, lo que hacemos, en esencia, es llenar la tabla de DP utilizando el orden topológico del DAG implícito subyacente de recurrencias de DP.

El problema de los *caminos más largos* (de origen único)<sup>17</sup>, es un problema de búsqueda de los caminos más largos (sencillos<sup>18</sup>) desde un vértice de inicio  $s$  hasta otros vértices. La versión de decisión de este problema es NP-completa en un grafo general<sup>19</sup>. Sin embargo, el problema vuelve a ser fácil si el grafo no tiene ciclos, lo que es cierto en un DAG. La solución para los caminos más largos en un DAG<sup>20</sup>, consiste en un pequeño cambio sobre la solución de programación dinámica para los SSSP en un DAG, que hemos visto antes. Un truco aplicable consiste en multiplicar todos los pesos de las aristas por -1 y ejecutar la solución SSSP anterior.

<sup>17</sup>En realidad, podrían ser orígenes múltiples, ya que podemos empezar desde cualquier vértice con un grado de entrada 0.

<sup>18</sup>En un grafo general, con aristas de peso positivo, el problema del camino más largo está mal definido, pues se puede tomar un ciclo positivo y utilizarlo para crear un camino más largo de distancia infinita. Ocurre lo mismo en el caso de ciclos negativos en el problema del camino más corto. Esa es la razón por la que, en un grafo general, utilizamos el término ‘problema del camino más largo *sencillo*’. Todos los caminos de un DAG son sencillos por definición, por lo que podemos abreviar con el término ‘problema del camino más largo’.

<sup>19</sup>La versión de decisión de este problema pide si el grafo general tiene un camino sencillo de peso total  $\geq k$ .

<sup>20</sup>El problema LIS de la sección 3.5.2 también se puede modelar como la búsqueda de caminos más largos en un DAG implícito.

Por último, debemos volver a multiplicar por -1 los valores resultantes, para obtener la respuesta.

Los caminos más largos en un DAG tienen aplicaciones en la organización de proyectos, como por ejemplo en UVa 452 - Project Scheduling, sobre técnicas de evaluación y revisión de proyectos (PERT). Podemos modelar la dependencia entre subproyectos como un DAG y el tiempo necesario para completar un subproyecto como el *peso de un vértice*. El tiempo mínimo imprescindible para completar el proyecto completo, viene determinado por el camino más largo de este DAG (también llamado el camino *crítico*), que comienza desde cualquier vértice (subproyecto) con grado de entrada 0. En la figura 4.32 hay un ejemplo con 6 subproyectos, sus unidades de tiempo de finalización estimada y sus dependencias. El camino más largo  $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ , con 16 unidades de tiempo, determina el tiempo más corto posible para finalizar el proyecto completo. Para lograrlo, todos los subproyectos a lo largo del camino más largo (crítico) deben finalizar a tiempo.

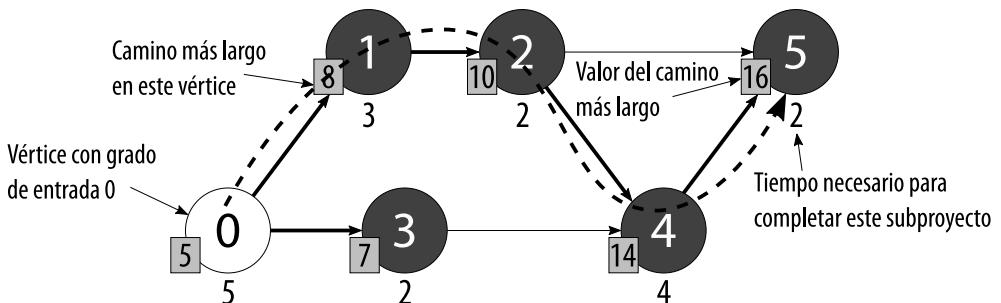


Figura 4.32: El camino más largo en este DAG

### Conteo de caminos en un DAG

Problema de referencia (UVa 988 - Many paths, one destination): en la vida uno tiene que elegir entre muchos caminos y cada uno nos dirige a muchas vidas diferentes. Enumera cuántas vidas diferentes se pueden vivir, dado un conjunto específico de elecciones en cada momento. Recibimos una lista de eventos y el número de elecciones que se pueden hacer para cada uno de ellos. El objetivo es contar cuántas formas hay de ir desde el evento que lo comenzó todo (el nacimiento, índice 0) hasta el que ya no deje más posibilidades (es decir, la muerte, índice  $n$ ).

Es evidente que el grafo subyacente del problema anterior es un DAG, ya que podemos movernos hacia adelante en el tiempo, pero no hacia atrás. El número de caminos se encuentra fácilmente al calcular un (cualquier) orden topológico en  $O(V+E)$  (en este problema, el vértice 0/nacimiento, siempre será el primero en el orden topológico, y el vértice  $n$ /muerte, siempre será el último). Comenzamos estableciendo `num_caminos[0] = 1`. Después, procesamos el resto de vértices, de uno en uno, de acuerdo al orden topológico. Cuando procesamos un vértice  $u$ , actualizamos cada vecino  $v$  de  $u$ , estableciendo `num_caminos[v] += num_caminos[u]`. Después de esos  $O(V+E)$  pasos, sabremos el número de caminos que hay en `num_caminos[n]`. La figura 4.33 muestra un ejemplo con 9 eventos y 6 recorridos vitales posibles.

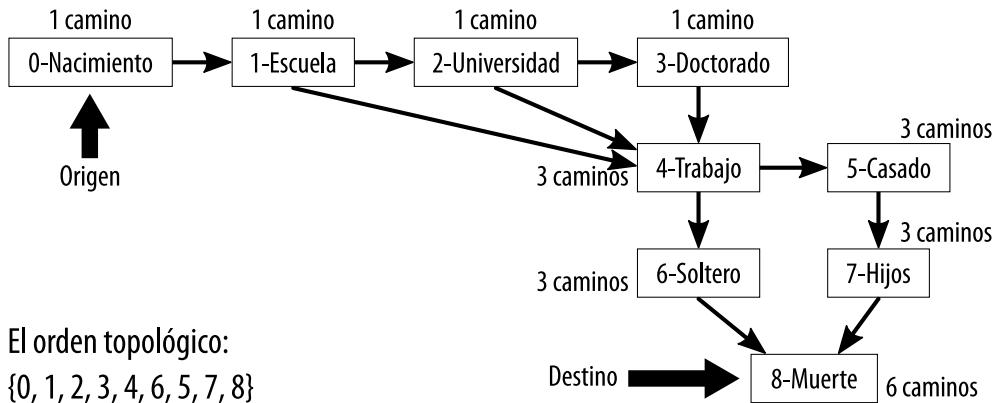


Figura 4.33: Ejemplo de conteo de caminos en un DAG, de abajo a arriba

#### Implementaciones de abajo a arriba frente a de arriba a abajo

Antes de continuar, nos gustaría incidir en que las tres soluciones a caminos más cortos/más largos/contados en el DAG anterior, son soluciones de DP de abajo a arriba. Comenzamos en casos base conocidos (los vértices de origen) y utilizamos el orden topológico del DAG para propagar la información correcta a los vértices vecinos, sin necesidad de volver hacia atrás.

En la sección 3.5, vimos que la DP también se puede escribir de arriba a abajo. Usando el problema UVa 988 como ilustración, también podemos escribir la solución de DP de la siguiente manera: digamos que  $\text{numPaths}(i)$  es el número de caminos que comienzan en el vértice  $i$  hasta el destino  $n$ . Escribimos la solución con estas relaciones de recurrencia de búsqueda completa:

1.  $\text{numCaminos}(n) = 1 // \text{en el destino } n, \text{ solo hay un camino posible}$
2.  $\text{numCaminos}(i) = \sum_j \text{numCaminos}(j), \forall j \text{ adyacente a } i$

Para evitar reiterar cálculos, *memoizamos* el número de caminos de cada vértice  $i$ . Hay  $O(V)$  vértices (estados) distintos, y cada vértice solo se procesa una vez. También hay  $O(E)$  aristas que, igualmente, se visitan una única vez. Por lo tanto, la complejidad de tiempo de esta técnica de arriba a abajo es también de  $O(V+E)$ , idéntica a la de abajo a arriba vista antes. La figura 4.34 muestra el DAG similar, pero los valores están calculados de destino a origen (siguiendo las flechas punteadas). Compara esta figura 4.34 con la anterior 4.33, donde los valores estaban calculados de origen a destino.

#### Conversión de un grafo general en un DAG

En los problemas más desafiantes de los concursos, el grafo dado en el enunciado del problema no es un *DAG explícito*. Sin embargo, tras un análisis más profundo, dicho grafo se puede modelar como un DAG, si añadimos uno o más parámetros. Una vez que tengamos el DAG, el siguiente paso será aplicar la técnica de programación dinámica (de arriba a abajo o de abajo a arriba). Podemos ilustrar este concepto mediante dos ejemplos.

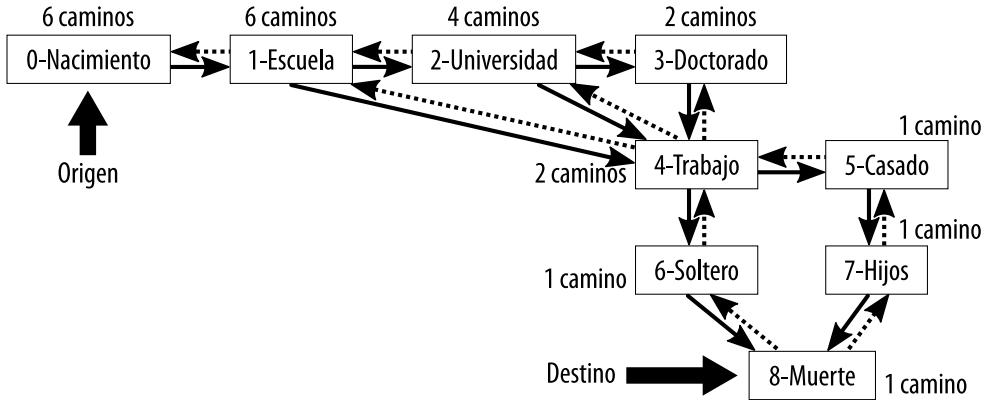


Figura 4.34: Ejemplo de conteo de caminos en un DAG, de arriba a abajo

### 1. SPOJ 0101: Fishmonger

Enunciado resumido del problema: dado el número de ciudades  $3 \leq n \leq 50$ , el tiempo disponible  $1 \leq t \leq 1000$  y dos matrices  $n \times n$  (una indica los tiempos de viaje y la otra los peajes entre dos ciudades), elegir una ruta desde la ciudad portuaria (vértice 0), de forma que el pescadero tenga que pagar el menor peaje posible para llegar a la ciudad del mercado (vértice  $n - 1$ ), dentro de un cierto tiempo  $t$ . El pescadero *no tiene* que visitar todas las ciudades. Devolver dos datos: el número total de peajes utilizados y el tiempo de viaje consumido. En la parte izquierda de la figura 4.35, está el grafo de entrada original de este problema.

Hay *dos* requisitos potencialmente conflictivos en este problema. El primero es *minimizar* los peajes de la ruta. El segundo es *asegurar* que el pescadero llega a la ciudad del mercado dentro del tiempo establecido, lo que puede provocar que tenga que pagar peajes más caros en algún punto del recorrido. El segundo requisito es un límite *inalterable* de este problema. Es decir, debemos satisfacerlo o no tendremos una solución.

Un algoritmo voraz de SSSP, como el de Dijkstra (ver la sección 4.4.3), en su forma canónica, no sirve para este problema. Elegir un camino con el tiempo de viaje más corto para ayudar al pescadero a llegar a la ciudad del mercado  $n - 1$ , utilizando un tiempo  $\leq t$ , no implica el peaje más barato posible. Y elegir el peaje más barato posible, no garantiza que el pescadero llegue a la ciudad del mercado  $n - 1$  en un tiempo  $\leq t$ . Los dos requisitos no son independientes.

Sin embargo, si añadimos un parámetro  $t\_left$  (tiempo restante) a cada vértice, entonces el grafo se convierte en un DAG, como se muestra en la parte derecha de la figura 4.35. Comenzamos con un vértice  $(puerto, t)$  en el DAG, a través de la arista con peso  $toll[cur][X]$ . Como el tiempo se va reduciendo, nunca nos encontraremos en una situación cíclica. Después, podemos utilizar la recursión de DP (de arriba a abajo)  $go(cur, t\_left)$ , para encontrar el camino más corto (en términos de peajes totales) en el DAG. La respuesta se encuentra llamando a  $go(0, t)$ . A continuación, incluimos el código en C++ de  $go(cur, t\_left)$ :

```

1 ii go(int cur, int t_left) { // devuelve par (peajepagado, tiemponecesario)
2 if (t_left < 0) return ii(INF, INF); // estado no válido, podar
3 if (cur == n-1) return ii(0, 0); // en el mercado, peaje=0, tiempo=0

```

```

4 if (memo[cur][t_left] != ii(-1, -1)) return memo[cur][t_left];
5 ii ans = ii(INF, INF);
6 for (int X = 0; X < n; X++) if (cur != X) { // ir a otra ciudad
7 ii nextCity = go(X, t_left - travelTime[cur][X]); // paso recursivo
8 if (nextCity.first + toll[cur][X] < ans.first) { // tomar coste mínimo
9 ans.first = nextCity.first + toll[cur][X];
10 ans.second = nextCity.second + travelTime[cur][X];
11 }
12 } }
13 return memo[cur][t_left] = ans; } // respuesta a tabla recordatoria

```

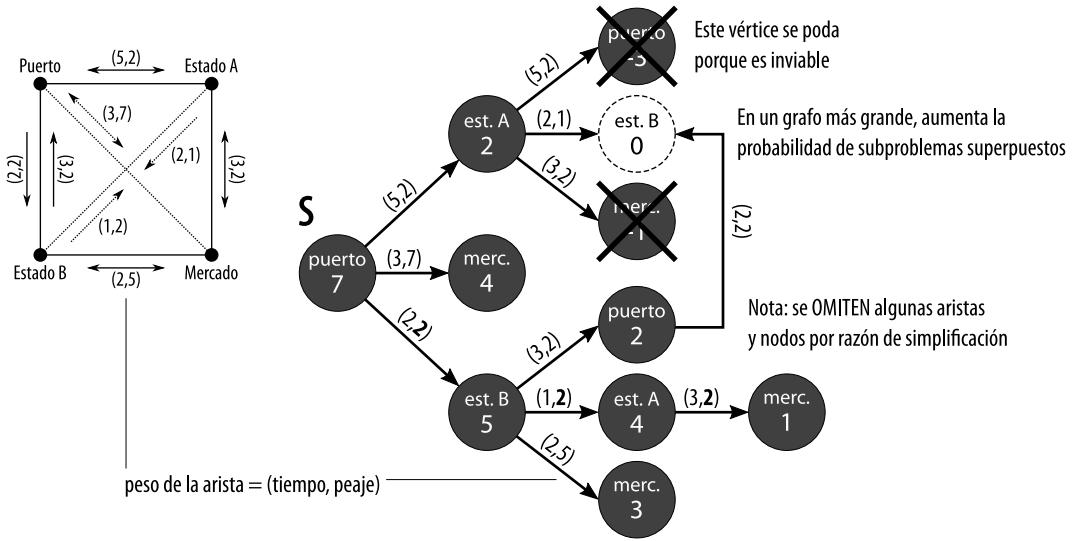


Figura 4.35: El grafo general dado (a la izquierda) se convierte en un DAG

Al utilizar DP de arriba a abajo, no tenemos que construir el DAG de forma explícita y calcular el orden topológico requerido. La recursión lo hará por nosotros. Solo hay  $O(nt)$  estados distintos (hay que notar que la tabla recordatoria es un objeto de pares). Cada estado se puede calcular en  $O(n)$ . La complejidad de tiempo total es, por lo tanto,  $O(n^2t)$ , lo que es factible.

## 2. Cobertura de vértices mínima (en un árbol)

La estructura de datos de árbol es también una estructura de datos acíclica. Pero, a diferencia del DAG, un árbol no tiene subárboles superpuestos. Por lo tanto, no tiene ningún sentido utilizar programación dinámica en un árbol estándar. Sin embargo, al igual que en el ejemplo anterior, algunos árboles se pueden convertir en DAG, en concursos de programación, si añadimos parámetros a cada uno de sus vértices. A partir de ahí, la solución consiste, normalmente, en ejecutar DP sobre el DAG resultante. Estos problemas se denominan (incorrectamente<sup>21</sup>) como ‘DP en un árbol’, dentro de la terminología de la programación competitiva.

<sup>21</sup>Ya hemos mencionado que no tiene sentido utilizar DP en un árbol. Pero el término ‘DP en un árbol’ que, en realidad, se refiere a ‘DP en un DAG implícito’, es una expresión muy conocida en la programación competitiva.

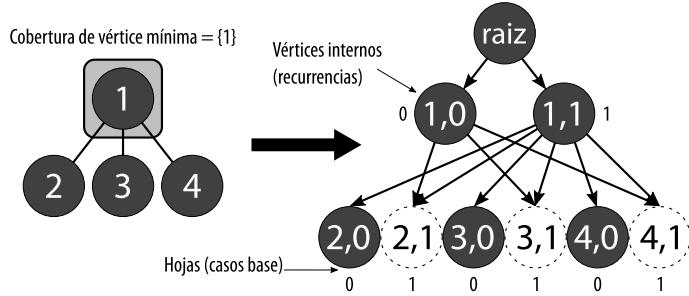


Figura 4.36: El grafo/árbol general dado (a la izquierda) se convierte en un DAG

Un ejemplo de esta DP en un árbol, es el problema de la búsqueda de la cobertura de vértices mínima (MVC) en un árbol. En este problema, debemos seleccionar el conjunto más pequeño posible de vértices  $C \in V$ , de forma que cada arista del árbol incida en, al menos, un vértice del conjunto  $C$ . En el árbol de ejemplo, a la izquierda de la figura 4.36, la solución está en tomar solo el vértice 1, porque todas las aristas 1-2, 1-3 y 1-4, son incidentes al vértice 1.

Ahora, tenemos solo dos posibilidades para cada vértice. O lo tomamos, o no. Al añadir este estado ‘tomado o no tomado’ a cada vértice, podemos convertir el árbol en un DAG (ver la parte derecha de la figura 4.36). Cada vértice tendrá la información del número de vértice y la etiqueta booleana de si está tomado, o no. Las aristas implícitas vienen determinadas por las siguientes reglas: primera, si el vértice actual no ha sido tomado, debemos tomar a todos sus hijos para encontrar una solución válida y, segunda, si el vértice actual ha sido tomado, obtenemos la mejor opción entre tomar o no tomar a sus hijos. Podemos escribir esta recurrencia de DP de arriba a abajo:  $\text{MVC}(v, \text{etiqueta})$ . La respuesta se encuentra llamando a  $\min(\text{MVC}(\text{raiz}, \text{falso}), \text{MVC}(\text{raiz}, \text{verdadero}))$ . Hay que señalar la presencia de subproblemas superpuestos (círculos punteados) en el DAG. Sin embargo, como solo hay  $2 \times V$  estados y cada vértice tiene un máximo de dos aristas entrantes, esta solución de programación dinámica se ejecuta en  $O(V)$ .

```

1 int MVC(int v, int flag) { // cobertura de vértices mínima
2 int ans = 0;
3 if (memo[v][flag] != -1) return memo[v][flag]; // DP de arriba a abajo
4 else if (leaf[v]) // leaf[v] es verdadero si v es una hoja, si no, falso
5 ans = flag; // 1/0 = tomado/no tomado
6 else if (flag == 0) { // si v no está tomado, debemos tomar a sus hijos
7 ans = 0; // Nota: 'hijos' es una lista de adyacencia que contiene la
8 // versión dirigida del árbol (puntos padre a sus hijos; pero los hijos
9 // no apuntan a los padres)
10 for (int j = 0; j < (int)Children[v].size(); j++)
11 ans += MVC(Children[v][j], 1);
12 }
13 else if (flag == 1) { // si v está tomado, obtenemos el mínimo entre
14 ans = 1; // tomar o no a sus hijos
15 for (int j = 0; j < (int)Children[v].size(); j++)
16 ans += min(MVC(Children[v][j], 1), MVC(Children[v][j], 0));
17 }
18 return memo[v][flag] = ans;
19 }
```

## Sección 3.5 - revisitada

Queremos seguir destacando al lector la fuerte conexión existente entre las técnicas de programación dinámica vistas en la sección 3.5 y los algoritmos sobre los DAG. Todos los ejercicios de programación sobre caminos más cortos/más largos/contados en un DAG (o en un grafo general convertido a un DAG, mediante modelado/transformación de grafos) se pueden clasificar también dentro de la categoría de DP. Es habitual, cuando tenemos un problema con solución de DP que ‘minimiza esto’, ‘maximiza aquello’ o ‘cuenta algo’, que dicha solución, en realidad, calcule los caminos más cortos, más largos o su número en la recurrencia de DP del DAG (normalmente implícito) de ese problema.

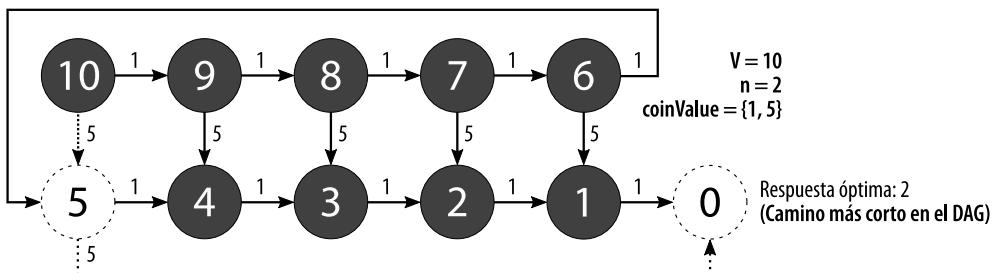


Figura 4.37: Cambio de monedas como caminos más cortos en un DAG

En este momento, invitamos al lector a que vuelva a leer, desde este nuevo punto de vista, algunos de los problemas de DP que hemos visto en la sección 3.5 (el tratar la DP como algoritmos sobre DAG no es un concepto habitual en otros libros de texto sobre ciencias de la computación). Para empezar, podemos revisitar el problema clásico del cambio de monedas. La figura 4.37 muestra el mismo caso de prueba utilizado en el primer ejemplo de la subsección del cambio de monedas de la sección 3.5.2. Tenemos  $n = 2$  denominaciones de monedas:  $\{1, 5\}$ . La cantidad objetivo es  $V = 10$ . Podemos modelar cada vértice como el valor actual. En este caso concreto, cada vértice  $v$  tiene  $n = 2$  aristas no ponderadas, que van a los vértices  $v-1$  y  $v-5$ , salvo que eso provoque que el índice sea negativo. Podemos ver que el grafo es un DAG y que algunos estados (marcados con círculos punteados) están superpuestos (tienen más de una arista entrante). En este punto, podemos resolver el problema encontrando el *camino más corto* en este DAG, desde el origen  $V = 10$  al destino  $V = 0$ . El ordenamiento topológico más sencillo consiste en procesar los vértices en orden inverso, es decir,  $\{10, 9, 8, \dots, 1, 0\}$ , que es un orden topológico válido. Sin ninguna duda, podemos utilizar los caminos más cortos en  $O(V+E)$  en una solución DAG. Sin embargo, como el grafo no es ponderado, también podemos utilizar BFS en  $O(V+E)$  para resolverlo (incluso podríamos utilizar Dijkstra, pero resultaría excesivo). El camino  $10 \rightarrow 5 \rightarrow 0$  es el más corto, con un peso total = 2 (son necesarias dos monedas). En este caso de prueba concreto, una solución voraz para el cambio de monedas también elegiría la misma ruta  $10 \rightarrow 5 \rightarrow 0$ .

A continuación, volvamos sobre el problema clásico de la mochila 0-1. Esta vez utilizaremos el siguiente caso de prueba:  $n = 5, V = \{4, 2, 10, 1, 2\}, W = \{12, 1, 4, 1, 2\}, S = 15$ . Podemos modelar cada vértice como un par de valores  $(id, remW)$ . Todo vértice tiene, al menos, una arista de  $(id, remW)$  a  $(id+1, remW)$ , que corresponde a no tomar un objeto determinado  $id$ . Algunos vértices tienen otra arista  $(id, remW)$  a  $(id+1, remW-W[id])$ , si  $W[id] \leq remW$ , que corresponde a tomar determinado objeto  $id$ . La figura 4.38 muestra algunas secciones del DAG de cálculo del problema de la mochila 0-1 estándar, utilizando el caso de prueba mencionado.

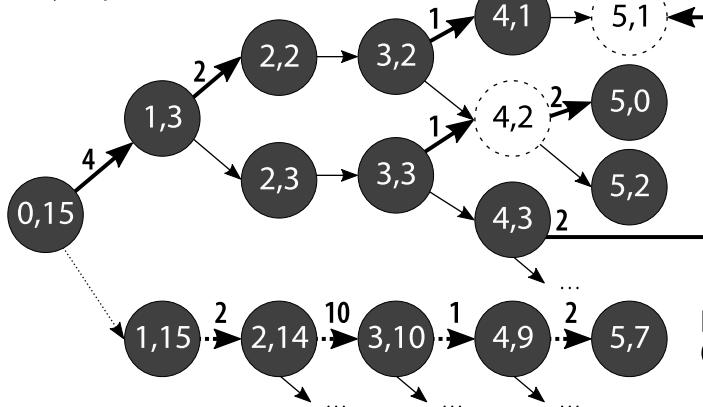
Se puede visitar alguno de los estados con más de un camino (un subproblema superpuesto se indica con un círculo punteado). A partir de aquí, podemos resolver el problema encontrando el *camino más largo* de este DAG desde el origen  $(0, 15)$  al destino  $(5, \text{cualquiera})$ . La respuesta es el camino  $(0, 15) \rightarrow (1, 15) \rightarrow (2, 14) \rightarrow (3, 10) \rightarrow (4, 9) \rightarrow (5, 7)$ , con peso  $0+2+10+1+2 = 15$ .

**Arista gruesa ponderada** = tomar el objeto,  
el valor muestra el peso de la arista

**Arista normal no ponderada** = no tomar  
el objeto, peso de la arista = 0

$$n = 5, S = 15$$

| id | v  | w  |
|----|----|----|
| 0  | 4  | 12 |
| 1  | 2  | 1  |
| 2  | 10 | 4  |
| 3  | 1  | 1  |
| 4  | 2  | 2  |



Respuesta óptima: 15  
Camino más largo en el DAG

Figura 4.38: Mochila 0-1 como caminos más largos en un DAG

Veamos un ejemplo más: la solución para UVa 10943 - How do you add?, tratada en la sección 3.5.3. Si dibujamos el DAG del caso de prueba  $n = 3, K = 4$ , tendremos el resultado que se muestra en la figura 4.39. Hay subproblemas superpuestos, que se indican con círculos punteados. Si contamos el número de caminos de este DAG, tendremos la respuesta = 20 caminos.

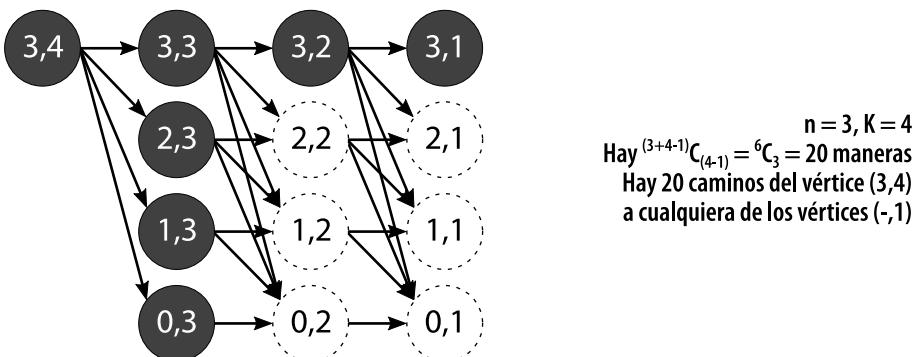


Figura 4.39: UVa 10943 como conteo de caminos en un DAG

### Ejercicio 4.7.1.1\*

Dibuja el DAG de algunos de los casos de prueba de otros problemas clásicos de DP de la sección 3.5, que no hayamos mencionado: el problema del viajante (TSP)  $\approx$  caminos más cortos en el DAG implícito, subsecuencia creciente máxima (LIS)  $\approx$  caminos más largos en el DAG implícito, variante del cambio de monedas (el que cuenta el número de formas posibles de obtener el valor  $V$  céntimos, utilizando una lista de denominaciones de  $N$  monedas)  $\approx$  conteo de caminos en el DAG, etc.

## 4.7.2 Árbol

Un árbol es un grafo especial con las siguientes características: tiene  $E = V - 1$  (cualquier algoritmo  $O(V+E)$  es  $O(V)$  en un árbol), no tiene ciclos, es conexo y existe un camino único para cualquier par de vértices.

### Recorrido de un árbol

En las secciones 4.2.1 y 4.2.2, hemos visto algoritmos de DFS y BFS en  $O(V + E)$ , para recorrer un grafo generalista. Si el grafo dado es un *árbol binario con raíz*, existen algoritmos de recorrido *más sencillos*, como preorden, inorden y postorden (nota: los recorridos nivel-orden son, esencialmente, BFS). No hay una mejora de velocidad significativa, ya que estos tres algoritmos de recorrido se ejecutan en  $O(V)$ , pero el código es más sencillo. El pseudocódigo a continuación:

```
preorden(v) inorden(v) postorden(v)
 visita(v); inorden(izquierda(v)); postorden(izquierda(v));
 preorden(izquierda(v)); visita(v); postorden(derecha(v));
 preorden(derecha(v)); inorden(derecha(v)); visita(v);
```

### Búsqueda de puntos de articulación y puentes en un árbol

En la sección 4.2.1, hemos visto el algoritmo de DFS de Tarjan en  $O(V+E)$ , para encontrar puntos de articulación y puentes en un grafo. Sin embargo, si el grafo dado es un árbol, el problema se vuelve más sencillo, pues todas las aristas de un árbol son puentes, y todos los vértices internos ( $\text{grado} > 1$ ), son puntos de articulación. Sigue necesitando  $O(V)$ , pues tenemos que realizar un barrido del árbol para contar el número de vértices internos, pero el código es *más sencillo*.

### Caminos más cortos de origen único en un árbol ponderado

En las secciones 4.4.3 y 4.4.4, hemos visto dos algoritmos que tienen un uso general (Dijkstra en  $O((V+E) \log V)$  y Bellman-Ford  $O(VE)$ ), para resolver el problema de los SSSP en un grafo ponderado. Pero si el grafo dado es un árbol ponderado, el problema de los SSSP se vuelve *más sencillo*, pues cualquier algoritmo de recorrido de un grafo en  $O(V)$ , es decir, BFS o DFS, se puede utilizar para resolverlo. Hay un único camino entre dos vértices cualesquiera de un árbol,

así que, simplemente, lo recorremos para encontrar el camino único que los conecta. El peso de camino más corto entre estos dos vértices es, básicamente, la suma de los pesos de las aristas de ese camino único (por ejemplo, para ir del vértice 5 al 3 en la figura 4.40.A, el camino único es  $5 \rightarrow 0 \rightarrow 1 \rightarrow 3$ , con peso  $4 + 2 + 9 = 15$ ).

### Caminos más cortos entre todos los pares en un árbol ponderado

En la sección 4.5, hemos visto un algoritmo de uso general (Floyd Warshall en  $O(V^3)$ ), para resolver el problema de los APSP sobre un grafo ponderado. Sin embargo, si el grafo dado es un árbol ponderado, el problema de los APSP se hace *más sencillo*, al repetir el SSSP  $V$  veces en el árbol ponderado, estableciendo cada vértice como el origen, de uno en uno. La complejidad de tiempo total es  $O(V^2)$ .

### Diámetro de un árbol ponderado

En un grafo general, necesitamos el algoritmo de Floyd Warshall en  $O(V^3)$ , tratado en la sección 4.5, junto a otra comprobación de todos los pares en  $O(V^2)$ , para calcular el diámetro. Sin embargo, si el grafo dado es un árbol ponderado, el problema se vuelve *más sencillo*. Solo necesitamos dos recorridos en  $O(V)$ . Realizamos DFS/BFS desde *cualquier* vértice  $s$ , para encontrar el vértice más alejado  $x$  (por ejemplo, desde el vértice  $s = 1$  hasta el  $s = 2$ , en la figura 4.40.B1), entonces hacemos otra DFS/BFS desde el vértice  $x$ , para obtener el vértice realmente más alejado  $y$  desde  $x$ . La longitud del único camino entre el vértice  $x$  y el vértice  $y$  es el diámetro de ese árbol (por ejemplo, el camino  $x = 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow y = 5$ , con longitud 20, en la figura 4.40.B2).

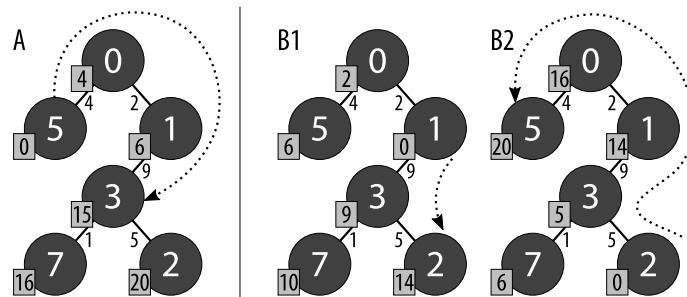


Figura 4.40: A: SSSP (parte de APSP); B1-B2: diámetro del árbol

### Ejercicio 4.7.2.1\*

Dados los recorridos inorden y preorden de un árbol de búsqueda binaria con raíz (BST)  $T$ , que contenga  $n$  vértices, escribe un pseudocódigo recursivo que muestre el recorrido postorden de ese BST. ¿Cuál es la complejidad de tiempo de tu mejor algoritmo?

### Ejercicio 4.7.2.2\*

Hay una solución más rápida que  $O(V^2)$  para el problema de caminos más cortos entre todos los pares en un árbol ponderado. Utiliza LCA. ¿Cómo?

### 4.7.3 Grafo euleriano

Un *camino euleriano* se define como el camino de un grafo que visita *cada vértice* del grafo *exactamente una vez*. Igualmente, un *ciclo euleriano* es un camino euleriano que comienza y termina en el mismo vértice. Un grafo que contenga un camino o ciclo eulerianos, se denomina grafo euleriano<sup>22</sup>.

Este tipo de grafo fue estudiado por primera vez por Leonhard Euler, mientras resolvía el problemas de los siete puentes de Königsberg, en 1736. El descubrimiento de Euler fue el ‘comienzo’ del campo de la teoría de grafos.

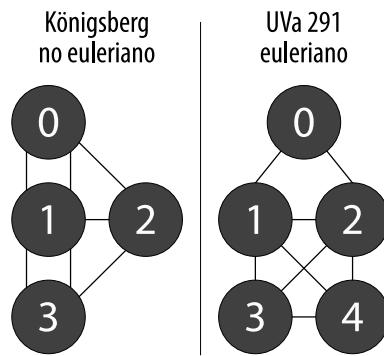


Figura 4.41: Euleriano

#### Comprobación de grafo euleriano

Comprobar si un grafo no dirigido conexo tiene un ciclo euleriano es sencillo. Basta con comprobar si todos los vértices tienen grados pares. La mecánica es similar para un camino euleriano, es decir, un grafo no dirigido contiene un camino euleriano si todos los vértices, menos dos, tienen grados pares. Este camino euleriano comenzará en uno de los vértices de grado impar y terminará en el otro<sup>23</sup>. Esta comprobación de grados puede hacerse en  $O(V+E)$ , normalmente de forma simultánea a la lectura del grafo de entrada. Puedes probar la comprobación en los dos grafos de la figura 4.41.

#### Escribir un ciclo euleriano

Aunque comprobar si un grafo es euleriano es sencillo, encontrar el ciclo/camino euleriano puede resultar más complicado. El siguiente código muestra el ciclo euleriano buscado cuando recibe un grafo euleriano no ponderado, almacenado como una lista de adyacencia, donde el segundo atributo del par de datos de la arista es el booleano 1 (la arista se puede utilizar) o 0 (la arista ya no se puede utilizar).

```
1 list<int> cyc; // necesitamos list para inserción rápida en el medio
2
3 void EulerTour(list<int>::iterator i, int u) {
4 for (int j = 0; j < (int)AdjList[u].size(); j++) {
5 ii v = AdjList[u][j];
6 if (v.second) { // si la arista se puede utilizar/no eliminada
7 v.second = 0; // hacer que el peso de la arista sea 0 ('eliminada')
8 for (int k = 0; k < (int)AdjList[v.first].size(); k++) {
9 ii uu = AdjList[v.first][k]; // eliminar arista bidireccional
10 if (uu.first == u && uu.second) {
11 uu.second = 0;
12 break;
```

<sup>22</sup>Comparar esta propiedad con la de *ciclo hamiltoniano* en el TSP (ver la sección 3.5.2).

<sup>23</sup>También es posible determinar un camino euleriano en un *grafo dirigido*: el grafo debe ser conexo, tener iguales vértices con grado de entrada y grado de salida, como mucho un vértice con grado de entrada - grado de salida = 1 y, como mucho, un vértice con grado de salida - grado de entrada = 1.

```

13 } }
14 EulerTour(cyc.insert(i, u), v.first);
15 } } }

16
17 // dentro de int main()
18 cyc.clear();
19 EulerTour(cyc.begin(), A); // cyc contiene camino euleriano empezado en A
20 for (list<int>::iterator it = cyc.begin(); it != cyc.end(); it++)
21 printf("%d\n", *it); // el camino euleriano

```

#### 4.7.4 Grafo bipartito

Un grafo bipartito es un grafo especial con las siguientes características: el conjunto de vértices  $V$  se puede dividir en dos conjuntos disjuntos  $V_1$  y  $V_2$ , y todas las aristas en  $(u, v) \in E$  tienen la propiedad de que  $u \in V_1$  y  $v \in V_2$ . Esto hace que un grafo bipartito no tenga ciclos de longitud impar (ver el [ejercicio 4.2.6.3](#)). Piensa que un árbol también es un grafo bipartito.

##### Emparejamiento bipartito de cardinalidad máxima (MCBM) y su solución de flujo máximo

Problema de referencia (de la ronda de calificación 1 de TopCoder Open 2009 [31]): dada una lista de números  $N$ , devolver una lista de todos los elementos de  $N$  que puedan ser emparejados con éxito con  $N[0]$ , como parte de un *emparejamiento primo completo*, en orden ascendente. El emparejamiento primo completo significa que cada elemento  $a$  en  $N$  se empareja a otro elemento único  $b$  en  $N$ , de forma que  $a + b$  sea primo.

Por ejemplo, dada la lista de números  $N = \{1, 4, 7, 10, 11, 12\}$ , la respuesta es  $\{4, 10\}$ . Esto se debe al emparejamiento de  $N[0] = 1$  con 4 resultados en emparejamientos primos y los otros cuatro elementos también pueden formar dos emparejamientos primos ( $7 + 10 = 17$  y  $11 + 12 = 23$ ). Encontramos una situación similar al emparejar  $N[0] = 1$  con 10, es decir,  $1 + 10 = 11$  es un emparejamiento primo y tenemos otros dos ( $4 + 7 = 11$  y  $11 + 12 = 23$ ). No podemos emparejar  $N[0] = 1$  con ningún otro elemento de  $N$ . Por ejemplo, si emparejamos  $N[0] = 1$  con 12, tenemos un emparejamiento primo, pero no hay forma de combinar los 4 números restantes para formar otros dos.

Restricciones: la lista  $N$  contiene un número par de elementos ( $[2..50]$ ). Cada elemento de  $N$  estará en el rango  $[1..1000]$  y será diferente.

Aunque este problema implica números primos, no es un problema puro de matemáticas, ya que los elementos de  $N$  no son más de 1000, y no hay muchos primos por debajo de esa cifra (solo son 168). El problema es que no podemos hacer emparejamientos por búsqueda completa, ya que hay  ${}_{50}C_2$  posibilidades para la primera pareja,  ${}_{48}C_2$  para la segunda, ..., hasta  ${}_2C_2$  para la última. Tampoco es viable la técnica de DP con máscara de bits (sección 8.3.1), porque  $2^{50}$  es demasiado grande.

La clave para resolver este problema se encuentra en identificar que este emparejamiento se realiza en un *grafo bipartito*. Para obtener un número primo, debemos sumar un número impar más un número par, porque dos números impares (o dos pares) dan como resultado un número par (que no es primo). Por lo tanto, podemos dividir los números impares y pares en dos conjuntos `set1/set2` y añadir una arista  $i \rightarrow j$  si `set1[i] + set2[j]` es primo.

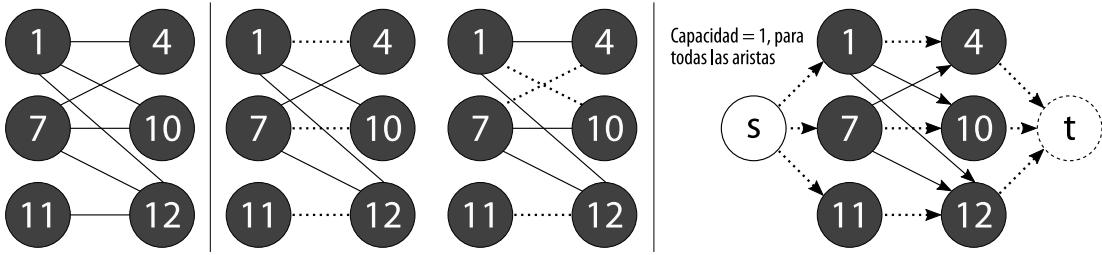


Figura 4.42: El problema de emparejamiento bipartito se puede reducir a uno de flujo máximo

Una vez construido este grafo bipartito, la solución es trivial pues, si los tamaños de `set1` y `set2` son diferentes, no es posible realizar un emparejamiento completo. Por otro lado, si el tamaño de ambos conjuntos es  $n/2$ , intentaremos conectar `set1[0]` con `set2[k]` para  $k = [0..n/2-1]$ , y realizar emparejamiento bipartito de cardinalidad máxima (MCBM) para el resto (la MCBM es una de las aplicaciones más comunes con grafos bipartitos). Si obtenemos  $n/2 - 1$  emparejamientos más, añadimos `set2[k]` a la respuesta. En este caso de prueba, la respuesta es  $\{4, 10\}$  (ver la parte central de la figura 4.42).

El problema de MCBM se puede reducir al de flujo máximo, si añadimos un vértice de origen `s` conectado a todos los vértices de `set1` y todos los vértices de `set2` se conectan a otro vértice adicional de desagüe `t`. Las aristas son dirigidas ( $s \rightarrow u$ ,  $u \rightarrow v$ ,  $v \rightarrow t$  donde  $u \in \text{set1}$  y  $v \in \text{set2}$ ). Al establecer las capacidades de todas las aristas de este grafo de flujo a 1, obligamos a que cada vértice de `set1` se empareje con, como máximo, un vértice de `set2`. El flujo máximo será igual al número máximo de emparejamientos del grafo original (ver la parte derecha de la figura 4.42 como ejemplo).

#### Conjunto independiente máximo y cobertura de vértices mínima en un grafo bipartito

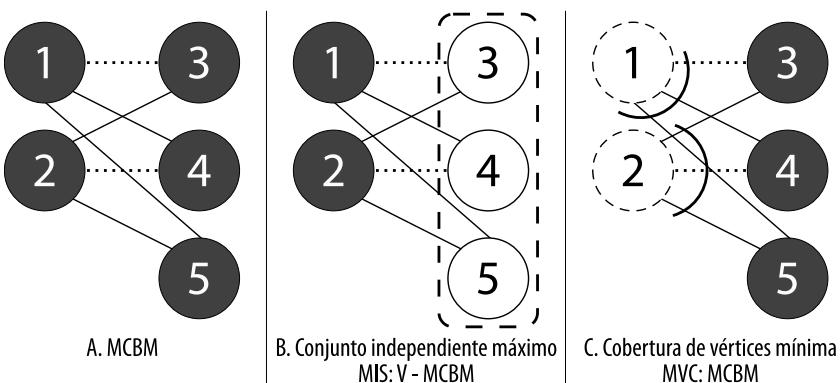


Figura 4.43: Variantes de MCBM

Un conjunto independiente (IS) de un grafo  $G$ , es un subconjunto de los vértices, de forma que no haya dos vértices en el subconjunto que representen una arista de  $G$ . Un IS máximo (MIS), es un IS en el que el añadir cualquier otro vértice al conjunto haga que este contenga una arista. En un grafo bipartito, el tamaño del MIS + el tamaño de la MCBM =  $|V|$ . O, en

otras palabras, el tamaño del MIS =  $|V|$  - el tamaño de la MCBM. En la figura 4.43.B, tenemos un grafo bipartito con 2 vértices a la izquierda y 3 a la derecha. El tamaño de la MCBM es 2 (dos líneas punteadas), y el tamaño del MIS es  $5-2 = 3$ . De hecho,  $\{3, 4, 5\}$  son los miembros del MIS de este grafo bipartito. Otra denominación del MIS es *conjunto dominante*.

La cobertura de vértices de un grafo  $G$  es un conjunto de  $C$  vértices, de forma que cada vértice de  $G$  sea incidente a, al menos, un vértice de  $C$ . En un grafo bipartito, el número de emparejamientos en una MCBM es igual al número de vértices en una cobertura de vértices mínima (MVC) (esto es un teorema del matemático húngaro Dénés König). En la figura 4.43.C, tenemos el mismo grafo bipartito de antes, con MCBM = 2. El MVC también es 2. De hecho,  $\{1, 2\}$  son los miembros del MVC de este grafo bipartito.

Es relevante que, aunque los valores de MCBM/MIS/MVC son únicos, las soluciones pueden no serlo. Por ejemplo, en la figura 4.43.A, también podemos conectar  $\{1, 4\}$  y  $\{2, 5\}$ , con la misma cardinalidad máxima de 2.

#### Aplicación de ejemplo: UVa 12083 - Guardian of Decency

Enunciado resumido del problema: dados  $N \leq 500$  estudiantes (clasificados por su estatura, sexo, gusto musical y deporte favorito), determinar cuántos son aptos para realizar una excursión, si el profesor quiere que cualquier pareja de estudiantes cumpla con, al menos, uno de estos cuatro criterios, para que ninguna sea compatible: 1) su estatura difiere en más de 40 cm.; 2) son del mismo sexo; 3) su música preferida es diferente; 4) su deporte favorito es el mismo (seguramente será aficionados a distintos equipos y no se pondrán de acuerdo).

En primer lugar, nos fijamos en que el problema trata la búsqueda del conjunto independiente máximo, es decir, los estudiantes elegidos no deben tener la posibilidad de ser compatibles. El conjunto independiente es un problema difícil en un grafo general, así que vamos a comprobar si el grafo es especial. Después, hay que darse cuenta de que hay un grafo bipartito evidente en el enunciado del problema: el sexo de los estudiantes (que está limitado a dos). Podemos poner a los estudiantes masculinos a la izquierda y a los femeninos a la derecha. En este punto, nos debemos preguntar: ¿cuáles son las aristas de este grafo bipartito? La respuesta está relacionada con el problema del conjunto independiente: trazamos una arista entre un estudiante masculino  $i$  y una estudiante femenina  $j$ , si existe la posibilidad de que  $(i, j)$  sean compatibles.

En el contexto de este problema: si  $i$  y  $j$  son de sexo *DISTINTO* y su estatura difiere por *NO MÁS* de 40 cm. y su música preferida es *LA MISMA* y su deporte favorito es *DIFERENTE*, entonces esta pareja, formada por un estudiante masculino  $i$  y uno femenino  $j$ , tiene una alta probabilidad de ser compatible. El profesor solo podrá elegir a uno de los dos.

Obtenido este grafo bipartito, ejecutamos el algoritmo MCBM e informamos del resultado:  $N - MCBM$ . Con este ejemplo, volvemos a poner de relieve la importancia de tener buenas habilidades para el *modelado de grafos*. No tiene sentido conocer el algoritmo del MCBM y su código, si, para empezar, el concursante no puede identificar el grafo bipartito en el enunciado.

#### Algoritmo de aumento de camino para emparejamiento bipartito de cardinalidad máxima

Hay un método mejor para resolver el problema del MCBM en un concurso de programación (en términos de tiempo de implementación), en vez de tomar la ‘vía del flujo máximo’. Podemos utilizar el algoritmo del *aumento de camino* en  $O(VE)$ , que es específico y fácil de implementar.

Con esta implementación a mano, es posible resolver fácilmente todos los problemas de MCBM, incluyendo otros problemas de grafos que lo necesitan, como el conjunto independiente máximo en un grafo bipartito, la cobertura de vértices mínima en un grafo bipartito y la cobertura de caminos mínima en un DAG (ver la sección 9.24).

El aumento de camino forma un camino que se inicia en un vértice *libre* (*no emparejado*) en el conjunto izquierdo del grafo bipartito, alterna entre una arista libre (ahora en el conjunto derecho), una arista emparejada (otra vez en el conjunto izquierdo), ..., una arista libre (ahora en el conjunto derecho), hasta que el camino llega, finalmente, a un vértice *libre* en el conjunto derecho del grafo bipartito. Claude Berge dijo en 1957 que un emparejamiento  $M$  en un grafo  $G$  es máximo (tiene el mayor número posible de aristas) si, y solo si, no se puede seguir realizando un aumento de camino en  $G$ . Este algoritmo del aumento de camino, es la implementación directa de la afirmación de Berge: buscar y eliminar los *aumentos de camino*.

Ahora vamos a echar un vistazo a un grafo bipartito simple en la figura 4.44, con  $n$  y  $m$  vértices en los conjuntos izquierdo y derecho, respectivamente. Los vértices de la izquierda están numerados desde  $[1..n]$ , y los de la derecha lo están desde  $[n+1..n+m]$ . Este algoritmo trata de encontrar y, posteriormente, eliminar los aumentos de camino comenzando desde los vértices libres del conjunto izquierdo.

Comenzamos con un vértice libre 1. En la figura 4.44.A, podemos ver que este algoritmo conectará ‘erróneamente’<sup>24</sup>, los vértices 1 y 3 (en vez de 1 y 4), ya que el camino 1-3 es un aumento de camino sencillo. Ambos vértices, 1 y 3, están vértices libres. Al conectar los vértices 1 y 3, tendremos nuestro primer emparejamiento. Después de conectar los vértices 1 y 3, no podremos encontrar más emparejamientos.

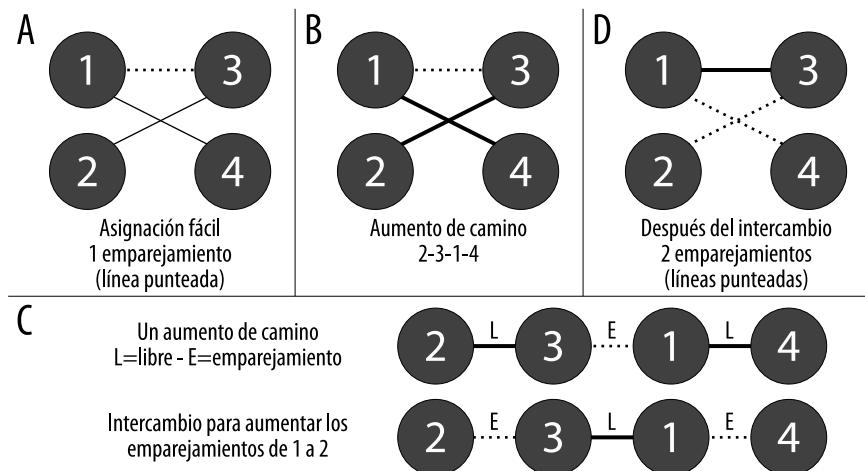


Figura 4.44: Algoritmo de aumento de camino

En la siguiente iteración (cuando nos encontramos en el vértice libre 2), este algoritmo mostrará toda su potencia, al encontrar el siguiente aumento de camino, que comienza desde el vértice libre 2 a la izquierda, va al vértice 3 utilizando una arista libre (2-3), va al vértice 1 mediante una arista ya utilizada (3-1) y, finalmente, vuelve al vértice 4 a través de una arista libre (1-4).

<sup>24</sup> Asumimos que los vecinos de un vértice están ordenados por su número de vértice creciente, es decir, desde el vértice 1 visitamos el vértice 3 *antes* que el 4.

Ambos vértices, 2 y 4, están libres. Por lo tanto, el aumento de camino es 2-3-1-4, como se ve en las figuras 4.44.B y 4.44.C.

Si intercambiamos el estado de las aristas en este aumento de camino, es decir, de ‘libres a utilizadas’ y de ‘utilizadas a libres’, obtendremos *un emparejamiento más*. En la figura 4.44.C se puede ver dónde se intercambia el estado de las aristas en el aumento de camino 2-3-1-4. Los emparejamientos actualizados se reflejan en la figura 4.44.D.

Este algoritmo seguirá realizando el proceso de encontrar aumentos de camino y eliminarlos, hasta que ya no queden más. Como el algoritmo repite un código  $O(E)$  similar a DFS<sup>25</sup>  $V$  veces, se ejecuta en  $O(VE)$ . El código aparece a continuación. Hay que decir que este no es el mejor algoritmo para encontrar el MCBM pero es, normalmente, suficiente. Más adelante, en la sección 9.12, aprenderemos el algoritmo de Hopcroft Karp, que resuelve el problema del MCBM en  $O(\sqrt{V}E)$  [28].

#### Ejercicio 4.7.4.1\*

En la parte derecha de la figura 4.42, hemos visto un método para reducir un problema de MCBM (los pesos de todas las aristas son 1) a un problema de flujo máximo. La pregunta es: ¿las aristas del grafo de flujo tienen que ser dirigidas? ¿Es correcto utilizar aristas no dirigidas en el grafo de flujo?

#### Ejercicio 4.7.4.2\*

Haz una lista con las palabras clave más comunes que pueden servir para ayudar a los concursantes a detectar un grafo bipartito en el enunciado de un problema. Por ejemplo, impar-par, masculino-femenino, etc. Toma también nota de qué problemas de concursos de programación incluyen esas palabras clave.

#### Ejercicio 4.7.4.3\*

Sugiere una mejora sencilla al algoritmo de aumento de camino que pueda evitar su complejidad de tiempo de  $O(VE)$ , en el peor caso en un grafo bipartito (casi) completo. Consejo: no necesitas el algoritmo de Hopcroft Karp, mencionado en la sección 9.12.

<sup>25</sup>Para simplificar el análisis, asumimos que  $E > V$  en estos grafos bipartitos.

```

1 vi match, vis; // variables globales
2
3 int Aug(int l) { // devolver 1 si se encuentra un aumento de camino
4 if (vis[l]) return 0; // devolver 0 en caso contrario
5 vis[l] = 1;
6 for (int j = 0; j < (int)AdjList[l].size(); j++) {
7 int r = AdjList[l][j]; // peso arista no necesario -> vector<vi> AdjList
8 if (match[r] == -1 || Aug(match[r])) {
9 match[r] = l; return 1; // encontrada 1 coincidencia
10 }
11 }
12 } // dentro de int main()
13 // construir grafo bipartito no ponderado con arista dirigida i->d fijada
14 int MCBM = 0;
15 match.assign(V, -1); // V = número de vértices en el grafo bipartito
16 for (int l = 0; l < n; l++) { // n = tamaño del conjunto izquierdo
17 vis.assign(n, 0); // reiniciar antes de cada recursión
18 MCBM += Aug(l);
19 }
20 printf("Found %d matchings\n", MCBM);

```

 [visualgo.net/matching](http://visualgo.net/matching)



ch4\_09\_mcbm.cpp



ch4\_09\_mcbm.java

## Comentarios sobre grafos especiales en concursos de programación

De los cuatro tipos de grafos especiales mencionados en esta sección 4.7, los DAG y los árboles son los más populares, especialmente entre los concursantes de la IOI. *No es extraño* que una tarea de la IOI contenga programación dinámica sobre un DAG o un árbol. Como estas variantes de la DP tienen, habitualmente, soluciones eficientes, el tamaño de la entrada suele ser grande. El siguiente grafo especial más popular es el bipartito. Este grafo especial es adecuado para problemas de flujo de red y de emparejamiento bipartito. Insistimos en que los concursantes deben dominar el uso del algoritmo de aumento de camino, más sencillo, para resolver el problema de la emparejamiento bipartito de cardinalidad máxima (MCBM). En esta sección, hemos visto que muchos problemas de grafos se pueden reducir, de alguna forma, a un problema MCBM. Los concursantes del ICPC deben familiarizarse con el grafo bipartito, además del DAG y el árbol. Los concursantes de la IOI no tiene por qué preocuparse del grafo bipartito, pues no está incluido en el temario de la IOI de 2009 [20]. El otro grafo especial que hemos tratado en este capítulo, el euleriano, no aparece hoy en día en muchos problemas de concursos. Existen otros grafos especiales, pero raramente los encontraremos. Son, por ejemplo, el grafo planar, el grafo completo  $K_n$ , el bosque de caminos, el grafo en estrella, etc. Cuando aparezcan, intenta utilizar sus propiedades especiales para acelerar tus algoritmos.

## Perfiles de los inventores de algoritmos

**Dénes König** (1884-1944) fue un matemático húngaro, que trabajó en el campo de la teoría de grafos y escribió el primer libro de texto sobre la materia. En 1931, König describió la equivalencia entre el problema del emparejamiento máximo y el de la cobertura de vértice mínima, en el contexto de los grafos bipartitos, es decir, demostró que el tamaño de la MCBM = el tamaño de la MVC en un grafo bipartito.

**Claude Berge** (1926-2002) fue un matemático francés, reconocido como uno de los fundadores de la combinatoria y la teoría de grafos modernas. Su mayor aportación incluida en este libro es el lema de Berge, que afirma que una  $M$  emparejada en un grafo  $G$  es máxima si, y solo si, no hay más aumentos de camino en  $G$  con respecto a  $M$ .

## Ejercicios de programación

### Ejercicios de programación relativos a grafos especiales:

#### Caminos más cortos/largos de origen único en un DAG

1. UVa 00103 - Stacking Boxes (caminos más largos en un DAG; sirve el *backtracking*)
2. **UVa 00452 - Project Scheduling \*** (PERT; caminos más largos en un DAG)
3. UVa 10000 - Longest Paths (caminos más largos en un DAG; sirve el *backtracking*)
4. UVa 10051 - Tower of Cubes (caminos más largos en un DAG; DP)
5. UVa 10259 - Hippity Hopscotch (caminos más largos en un DAG implícito; DP)
6. **UVa 10285 - Longest Run ... \*** (caminos más largos en un DAG implícito; sin embargo, el grafo es pequeño y se puede usar una solución de *backtracking* recursivo)
7. **UVa 10350 - Liftless Eme \*** (caminos más cortos; DAG implícito; DP)

Ver también subsecuencia creciente máxima (sección 3.5.3)

#### Conteo de caminos en un DAG

1. UVa 00825 - Walking on the Safe Side (conteo de caminos en DAG implícito; DP)
2. UVa 00926 - Walking Around Wisely (similar a UVa 825)
3. UVa 00986 - How Many? (conteo de caminos en un DAG; DP; s: x, y, último\_movimiento, picos\_encontrados; t: probar NE/SE)
4. **UVa 00988 - Many paths, one ... \*** (conteo de caminos en un DAG; DP)
5. **UVa 10401 - Injured Queen Problem \*** (conteo de caminos en un DAG implícito; DP; s: (columna, fila); t: siguiente columna, evitar 2 o 3 filas adyacentes)
6. UVa 10926 - How Many Dependencies? (conteo de caminos en un DAG; DP)
7. UVa 11067 - Little Red Riding Hood (similar a UVa 825)
8. UVa 11655 - Waterland (conteo de caminos en un DAG y otra tarea similar: contar el número de vértices implicados en los caminos)
9. **UVa 11957 - Checkers \*** (conteo de caminos en un DAG implícito; DP)

## Conversión de grafo general a un DAG

1. UVa 00590 - Always on the Run
2. **UVa 00907 - Winterim Backpack... \***
  - (s: (pos, **resto\_de\_día**))
  - (s: (pos, **resto\_de\_noche**))
  - (s: (pos, **movimientos\_restantes**))
  - (s: (pos, **combustible\_restante**))
  - (s: (pos, **discurso\_dado**))
  - (s: (pos, **resto\_de\_día**))
  - (s: (pos, **T\_restante**))
  - (s: (fila, **izquierda/derecha**); t: ir izquierda/derecha)
  - (s: (r, c, **neg\_restante\_estado**); t: abajo/(izquierda/derecha))  
(suma cromática mínima; máximo 6 colores)
  - (s: (r, c, **comida\_actual\_longitud**); t: 4 direcciones)
  - (s: (posAct, **tiempoAct**, **tiempoEsperaAct**); t: moverse adelante/resto)
  - (s: (id, **K\_restantes**); t: tomar raíz/intentar subárbol izquierda-derecha)  
(tratado en esta sección)
3. UVa 00910 - TV Game
4. UVa 10201 - Adventures in Moving ...
5. UVa 10543 - Traveling Politician
6. UVa 10681 - Teobaldo's Trip
7. UVa 10702 - Traveling Salesman
8. UVa 10874 - Segments
9. **UVa 10913 - Walking ... \***
10. UVa 11307 - Alternative Arborescence
11. **UVa 11487 - Gathering Food \***
12. UVa 11545 - Avoiding ...
13. UVa 11782 - Optimal Cut
14. SPOJ 0101 - Fishmonger

## Árbol

1. UVa 00112 - Tree Summing
  2. UVa 00115 - Climbing Trees
  3. UVa 00122 - Trees on the level
  4. UVa 00536 - Tree Recovery
  5. **UVa 00548 - Tree**
  6. UVa 00615 - Is It A Tree?
  7. UVa 00699 - The Falling Leaves
  8. UVa 00712 - S-Trees
  9. UVa 00839 - Not so Mobile
  10. UVa 10308 - Roads in the North
  11. **UVa 10459 - The Tree Root \***
  12. UVa 10701 - Pre, in and post
  13. **UVa 10805 - Cockroach Escape ... \***
  14. **UVa 11131 - Close Relatives**
  15. **UVa 11234 - Expressions**
  16. UVa 11615 - Family Tree
  17. **UVa 11695 - Flight Planning \***
  18. **UVa 12186 - Another Crisis**
  19. **UVa 12347 - Binary Search Tree**
- (*backtracking*)  
(recorrido de árbol; ancestro común mínimo)  
(recorrido de árbol)  
(reconstrucción desde pre + inorden)  
(reconstruir el árbol desde in + postorden)  
(comprobación de propiedades del grafo)  
(recorrido preorden)  
(variante de recorrido de árbol binario sencillo)  
(se puede ver como un problema recursivo en un árbol)  
(diámetro del árbol; tratado en esta sección)  
(identificar el diámetro de este árbol)  
(reconstruir el árbol desde pre + inorden)  
(implica diámetro del árbol)  
(leer el árbol; generar dos recorridos postorden)  
(convertir postorden a nivel-orden; árbol binario)  
(contar el tamaño de los subárboles)  
(cortar la peor arista en el diámetro del árbol; enlazar dos centros)  
(el grafo de entrada es un árbol)  
(dado el recorrido preorden de un BST, utilizar la propiedad de BST para obtener el BST; mostrar el recorrido postorden de esa BST)

## Grafo euleriano

1. UVa 00117 - The Postal Worker ...
  2. UVa 00291 - The House of Santa ...
  3. **UVa 10054 - The Necklace \***
  4. UVa 10129 - Play on Words
  5. **UVa 10203 - Snow Clearing \***
  6. **UVa 10596 - Morning Walk \***
- (camino euleriano; tomar el coste del camino)  
(camino euleriano en un grafo pequeño; basta con *backtracking*)  
(mostrar el camino euleriano)  
(comprobación de propiedades de grafo euleriano)  
(el grafo subyacente es euleriano)  
(comprobación de propiedades de grafo euleriano)

## Grafo bipartito

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"><li>1. UVa 00663 - Sorting Slides</li><li>2. UVa 00670 - The Dog Task</li><li>3. UVa 00753 - A Plug for Unix</li><li>4. UVa 01194 - Machine Schedule</li><li>5. UVa 10080 - Gopher II</li><li>6. <b>UVa 10349 - Antenna Placement *</b></li><li>7. <b>UVa 11138 - Nuts and Bolts *</b></li><li>8. <b>UVa 11159 - Factors and Multiples *</b></li><li>9. UVa 11419 - SAM I AM</li><li>10. UVa 12083 - Guardian of Decency</li><li>11. UVa 12168 - Cat vs. Dog</li><li>12. Top Coder Open 2009: Prime Pairs</li></ol> | <p>(probar a desactivar una arista para ver si cambia el MCBM; lo que implica que la arista no se utiliza)<br/>(MCBM)</p> <p>(inicialmente un problema de emparejamiento no estándar, pero se puede reducir a un problema MCBM simple)<br/>(LA 2523 - Beijing02; cobertura de vértices mínima/MVC)</p> <p>(MCBM)</p> <p>(MIS: <math>V</math> - MCBM)</p> <p>(problema MCBM puro; si comienzas con el MCBM, este problema es un buen punto de partida)</p> <p>(MIS; pero la respuesta es el MCBM)</p> <p>(MVC; teorema de König)</p> <p>(LA 3415 - NorthwesternEurope05; MIS)</p> <p>(LA 4288 - NorthwesternEurope08; MIS)</p> <p>(tratado en esta sección)</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 4.8 Soluciones a los ejercicios no resaltados

**Ejercicio 4.2.2.1:** basta sustituir  $\text{dfs}(\emptyset)$  con  $\text{bfs}$  desde el origen  $s = 0$ .

**Ejercicio 4.2.2.2:** la matriz de adyacencia, y las listas de adyacencia y aristas necesitan  $O(V)$ ,  $O(k)$  y  $O(E)$ , respectivamente, para enumerar la lista de vecinos de un vértice (nota:  $k$  es el número real de vecinos de un vértice). Como DFS y BFS exploran todas las aristas salientes de cada vértice, su tiempo de ejecución depende de la velocidad de la estructura de datos del grafo subyacente, al enumerar los vecinos. Por lo tanto, las complejidades de tiempo tanto de DFS como de BFS son  $O(V \times V = V^2)$ ,  $O(\max(V, V \sum_{i=0}^{V-1} k_i) = V+E)$  y  $O(V \times E = VE)$  para recorrer un grafo almacenado como matriz de adyacencia, lista de adyacencia y lista de aristas, respectivamente. Como la lista de adyacencia es la estructura de datos más eficiente para el recorrido de un grafo, puede ser beneficioso convertir primero una matriz de adyacencia o una lista de aristas a una lista de adyacencia (ver el **ejercicio 2.4.1.2\***), antes de recorrer el grafo.

**Ejercicio 4.2.3.1:** comenzar con vértices disjuntos. Para cada  $\text{edge}(u, v)$ , hacer  $\text{unionSet}(u, v)$ . El estado de los conjuntos disjuntos después de procesar todas las aristas, representa a los componentes conexos. La solución con BFS es ‘trivial’: basta con cambiar  $\text{dfs}(i)$  a  $\text{bfs}(i)$  desde el vértice de origen  $i$ . Ambos se ejecutan en  $O(V+E)$ , ya que asumimos que las operaciones UFDS son constantes.

**Ejercicio 4.2.5.1:** es un tipo de ‘recorrido postorden’, en terminología de recorrido de árboles binarios. La función  $\text{dfs2}$  visita a todos los hijos de  $u$ , antes de añadir el vértice  $u$  al final del vector  $\text{ts}$ . Así se satisface la propiedad de ordenación topológica.

**Ejercicio 4.2.5.2:** la respuesta es utilizar una lista enlazada. Sin embargo, hemos dicho en el capítulo 2 que conviene evitar las listas enlazadas, así que hemos decidido utilizar  $\text{vi ts}$ .

**Ejercicio 4.2.5.3:** el algoritmo terminará igualmente, pero la salida será irrelevante, pues los grafos no DAG no tienen orden topológico.

**Ejercicio 4.2.5.4:** debemos utilizar *backtracking* recursivo para hacerlo.

**Ejercicio 4.2.6.3:** demostración por contradicción. Asumimos que un grafo bipartito tiene un ciclo (de longitud) impar. Digamos que el ciclo impar contiene  $2k + 1$  vértices, para un cierto entero  $k$ , que forma este camino:  $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{2k-1} \rightarrow v_{2k} \rightarrow v_0$ . Ahora, podemos poner  $v_0$  en el conjunto izquierdo,  $v_1$  en el derecho, ...,  $v_{2k}$  de nuevo en el izquierdo, pero entonces tendremos una arista  $(v_{2k}, v_0)$  que dará problemas, ya que antes hemos puesto  $v_0$  en el izquierdo  $\rightarrow$  contradicción. Por lo tanto, un grafo bipartito no tiene ciclos impares. Esta propiedad puede resultar importante para resolver algunos problemas que implican grafos bipartitos.

**Exercice 4.2.7.1:** dos aristas inversas:  $2 \rightarrow 1$  y  $6 \rightarrow 4$ .

**Ejercicio 4.2.8.1:** puntos de articulación: 1, 3 y 6; puentes: 0-1, 3-4, 6-7 y 6-8.

**Ejercicio 4.2.9.1:** demostración por contradicción. Asumumos que existe un camino del vértice  $u$  al  $w$  y del  $w$  al  $v$ , donde  $w$  está fuera del SCC. A partir de aquí, podemos concluir que es posible viajar desde el vértice  $w$  a cualquier vértice del SCC y desde cualquier vértice del SCC a  $w$ . Por lo tanto, el vértice  $w$  debe estar en el SCC  $\rightarrow$  contradicción. Así pues, no hay ningún camino entre dos vértices en un SCC que abandonen el SCC.

**Ejercicio 4.3.2.1:** la razón de que esta finalización temprana sea correcta es que el algoritmo de Kruskal solo toma las aristas que serán parte del MST final. Si Kruskal ha tomado  $V - 1$  aristas, podemos estar seguros de que esas  $V - 1$  aristas no formarán un ciclo y, por definición, deben formar un árbol que ya recubra el grafo  $G$ . Otra implementación posible, utilizando la estructura de conjuntos disjuntos para unión-buscar, es detener el bucle del algoritmo de Kruskal cuando el número de conjuntos disjuntos sea uno. Recuerda que iniciamos el bucle del algoritmo de Kruskal con  $V$  conjuntos disjuntos iniciales y que cada arista tomada por el algoritmo reduce en uno el número de conjuntos disjuntos. Podemos hacerlo  $V-1$  veces antes de que solo quede un conjunto.

**Ejercicio 4.3.4.1:** encontramos que los problemas del ‘bosque’ MS y del segundo mejor ST son más difíciles de resolver con el algoritmo de Prim.

**Ejercicio 4.4.2.1:** para esta variante, la solución es sencilla. Basta con añadir a la cola todos los orígenes y establecer  $dist[s] = \infty$  para todos ellos, antes de ejecutar el bucle de la BFS. Como esto solo supone una llamada a la BFS, se ejecuta en  $O(V+E)$ .

**Ejercicio 4.4.2.2:** al principio del bucle `while`, cuando tomamos el primer vértice de la cola, comprobamos si ese vértice es el destino. Si lo es, detenemos el bucle. La complejidad de tiempo en el peor caso sigue siendo  $O(V+E)$ , pero nuestra BFS se detendrá antes, si el vértice de destino está cercano al de origen. Como la BFS se realiza en un grafo no ponderado, esta estrategia es correcta (ver también el [ejercicio 4.4.3.5\\*](#)).

**Ejercicio 4.4.2.3:** es posible transformar el grafo de peso constante en un grafo no ponderado, sustituyendo todos los pesos de las aristas por unos. La información SSSP obtenida por la BFS se multiplica, entonces, por esa constante, para obtener las respuestas correctas.

**Ejercicio 4.4.3.1:** en un grafo ponderado positivo, sí. Cada vértice solo se procesará una vez. Cada vez que se procesa un vértice, intentamos relajar a sus vecinos. Debido a la eliminación perezosa, podemos tener un máximo de  $O(E)$  elementos en la cola de prioridad en un momento dado, pero esto sigue siendo  $O(\log E) = O(\log V^2) = O(2 \times \log V) = O(\log V)$  por cada operación de añadir o eliminar de la cola. Por lo tanto, la complejidad de tiempo permanece en  $O((V+E) \log V)$ . En un grafo con (unas pocas) aristas de peso negativo, pero sin ciclos negativos, se ejecuta más despacio, debido a la necesidad de volver a procesar vértices ya procesados,

pero los valores de caminos más cortos son correctos (a diferencia de la implementación de Dijkstra mostrada en [7]). Esto puede verse en un ejemplo de la sección 4.4.4. En algunos casos atípicos, esta implementación de Dijkstra puede ser extraordinariamente lenta en determinados grafos con aristas de pesos negativos, aunque el grafo no tenga ciclos negativos (ver el **ejercicio 4.4.3.2\***). Si el grafo tiene ciclos negativos, esta variante de implementación de Dijkstra se verá atrapada en un bucle infinito.

**Ejercicio 4.4.3.3:** utilizar `set<ii>`. Este conjunto almacena pares ordenados de datos sobre los vértices, como se muestra en la sección 4.4.3. El vértice con la distancia mínima es el primer elemento del conjunto (ordenado). Para actualizar la distancia desde el origen de un vértice determinado, buscamos y eliminamos el par de valores antiguo. Después, insertamos el nuevo par. Como procesamos una vez cada vértice y cada arista  $y$ , para ello, accedemos cada vez a `set<ii>` en  $O(\log V)$ , la complejidad de tiempo total de la variante de implementación de Dijkstra, usando `set<ii>`, sigue siendo de  $O((V+E) \log V)$ .

**Ejercicio 4.4.3.4:** en la sección 2.3, hemos mostrado la forma de invertir el montículo máximo predeterminado de la `priority_queue` de la STL de C++, y convertirlo en un montículo mínimo, multiplicando las claves de ordenación por -1.

**Ejercicio 4.4.3.5:** una respuesta similar a la del **ejercicio 4.4.2.2**, si el grafo ponderado dado no tiene aristas de peso negativo. Hay un riesgo potencial de dar una respuesta incorrecta si el grafo ponderado dado tiene alguna arista de peso negativo.

**Ejercicio 4.4.3.6:** no, no podemos utilizar DP. El modelado de estados y transiciones mencionado en la sección 4.4.3 genera un grafo estado-espacio que *no es* un DAG. Por ejemplo, podemos comenzar desde el estado  $(s, 0)$ , añadir una unidad de combustible en el vértice  $s$  para llegar al estado  $(s, 1)$ , ir al vértice vecino  $y$  (suponemos que se encuentra a 1 unidad de distancia), para llegar al estado  $(y, 0)$ , añadir otra unidad de combustible en el vértice  $y$  para llegar al estado  $(y, 1)$  y, entonces, volver al estado  $(s, 0)$  (un ciclo). Esto es un problema de camino más corto en un grafo ponderado general. Necesitamos utilizar el algoritmo de Dijkstra.

**Ejercicio 4.4.4.1:** esto es debido a que, inicialmente, solo el vértice origen tiene la información de distancia correcta. Después, cada vez que relajamos todas las  $E$  aristas, garantizamos que, al menos, un vértice más con un salto más (en términos de las aristas usadas en el camino más corto desde el origen) tiene la información de distancia correcta. En el **ejercicio 4.4.1.1**, hemos visto que el camino más corto debe ser un camino sencillo (tiene un máximo de  $E = V-1$  aristas). Después de  $V-1$  pasadas del algoritmo de Bellman Ford, incluso el vértice con el número más grande de saltos tendrá la información de distancia correcta.

**Ejercicio 4.4.4.2:** poner una etiqueta booleana `modificado = falso` en el bucle más exterior (el que repite la relajación de todas las  $E$  aristas  $V-1$  veces). Si se hace, al menos, una operación de relajación en los bucles interiores (el que explora las  $E$  aristas), establecer `modificado = verdadero`. Salir inmediatamente del bucle más exterior si `modificado` sigue siendo falso después de que se hayan examinado todas las  $E$  aristas. Si esta no relajación se produce en la iteración  $i$  del bucle más exterior, tampoco habrá más relajaciones en las iteraciones  $i+1, i+2, \dots, i = V-1$ .

**Ejercicio 4.5.1.1:** esto es debido a que sumaremos  $\text{AdjMat}[i][k] + \text{AdjMat}[k][j]$ , lo que provocará un *desbordamiento*, si tanto  $\text{AdjMat}[i][k]$  como  $\text{AdjMat}[k][j]$  están cerca del rango `MAX_INT`, lo que nos dará una respuesta incorrecta que es bastante difícil de depurar.

**Ejercicio 4.5.1.2:** el algoritmo de Floyd Warshall funciona en un grafo con aristas de peso negativo. Para el caso de grafos con ciclos negativos, consultar la sección 4.5.3.

**Ejercicio 4.5.3.1:** ejecutar el algoritmo de Warshall directamente sobre un grafo con  $V \leq 1000$ , dará como resultado TLE. Como el número de consultas es bajo, podemos permitirnos ejecutar una DFS en  $O(V+E)$  por consulta, para comprobar si los vértices  $u$  y  $v$  están conectados por un camino. Si el grafo de entrada es dirigido, podemos encontrar primero los SCC en  $O(V+E)$ . Si  $u$  y  $v$  pertenecen al mismo SCC, entonces es seguro que  $u$  podrá llegar a  $v$ . Esto se puede comprobar sin incurrir en un coste adicional. Si el SCC que contiene  $u$  tiene una arista dirigida hacia el SCC que contiene  $v$ , entonces  $u$  también podrá llegar a  $v$ . Pero la comprobación de conectividad entre diferentes SCC es mucho más difícil de verificar y podríamos, simplemente, utilizar la DFS normal para obtener la respuesta.

**Ejercicio 4.5.3.3:** en el algoritmo de Floyd Warshall, sustituir la suma por multiplicación y establecer la diagonal principal a 1,0. Ejecutar Floyd Warshall y comprobar si la diagonal principal  $> 1,0$ .

**Ejercicio 4.6.3.1:**  $A = 150$ ;  $B = 125$ ;  $C = 60$ .

**Ejercicio 4.6.3.2:** en el código actualizado que mostramos a continuación, utilizando *tanto* una lista de adyacencia (para una enumeración rápida de los vecinos, no olvides incluir las aristas inversas correspondientes al flujo inverso) como una matriz de adyacencia (para el acceso rápido a la capacidad residual) en el mismo grafo de flujo, es decir, nos concentraremos en mejorar esta línea: `for (int v = 0; v < MAX_V; v++)`. También sustituimos `vi dist(MAX_V, INF);` por `bitset<MAX_V> vis;`, para acelerar el código un poco más.

```

1 // dentro de int main(), asumir que tenemos res (AdjMatrix) y AdjList
2 mf = 0;
3 while (1) { // un auténtico algoritmo de Edmonds Karp en O(VE^2)
4 f = 0;
5 bitset<MAX_V> vis; vis[s] = true; // cambiamos vi dist a bitset
6 queue<int> q; q.push(s);
7 p.assign(MAX_V, -1);
8 while (!q.empty()) {
9 int u = q.front(); q.pop();
10 if (u == t) break;
11 for (int j = 0; j < (int)AdjList[u].size(); j++) { // aquí AdjList
12 int v = AdjList[u][j]; // usamos vector<vi> AdjList
13 if (res[u][v] > 0 && !vis[v])
14 vis[v] = true, q.push(v), p[v] = u;
15 }
16 }
17 augment(t, INF);
18 if (f == 0) break;
19 mf += f;
20 }
```

**Ejercicio 4.6.4.1:** utilizamos  $\infty$  como capacidad de las ‘aristas dirigidas centrales’ entre los conjuntos izquierdo y derecho del grafo bipartito, por la validez de esta técnica de modelado de grafos de flujo en otros problemas similares. Si las capacidades del conjunto derecho hasta el desagüe  $t$  no son 1, como en UVa 259, obtendremos un valor de flujo máximo incorrecto, si establecemos la capacidad de estas ‘aristas dirigidas centrales’ a 1.

## 4.9 Notas del capítulo

Finalizamos este capítulo, relativamente largo, incidiendo en que en él hemos presentado infinidad de algoritmos y a sus inventores, la mayor cantidad de todo el libro. Esta tendencia, con toda seguridad, crecerá en el futuro, es decir, habrá *más* algoritmos de grafos utilizados en concursos de programación. Sin embargo, debemos advertir a los concursantes que en los ICPC y las IOI recientes, no solo se pide, normalmente, la resolución de problemas que implican las formas canónicas de estos algoritmos de grafos. Los nuevos problemas suelen exigir a los concursantes la utilización de modelado de grafos creativo, el uso de las propiedades especiales del grafo de entrada, la combinación de dos o más algoritmos o la combinación de un algoritmo con estructuras de datos avanzadas (por ejemplo, la mezcla del camino más largo en un DAG con una estructura de datos de árbol de segmentos), el uso de contracción SCC de un grafo dirigido, para transformarlo en un DAG, antes de resolver el problema propiamente dicho, etc. Veremos algunas de estas modalidades más difíciles de problemas de grafos en la sección 8.4.

Este capítulo, aunque bastante extenso, omite muchos algoritmos y problemas de grafos conocidos, que se pueden presentar en el ICPC. Por citar algunos: caminos más cortos  $k$ -ésimos, el problema del viajante bitónico (ver la sección 9.3), el **algoritmo de Chu Liu Edmonds** para el problema de la arborescencia de coste mínimo, el algoritmo MCBM de **Hopcroft Karp** (ver la sección 9.12), el algoritmo MCBM ponderado de **Kuhn Munkres**, el algoritmo de **emparejamiento de Edmonds** para grafos generales, etc. Invitamos a los lectores a conocer algunos de estos algoritmos en el capítulo 9.

Si quieres mejoras tus posibilidades de ganar en el ICPC, dedica algo de tiempo a estudiar más algoritmos/problemas de grafos<sup>26</sup> que los tratados en este libro. Estos problemas más difíciles no suelen aparecer en los concursos *regionales* y, si lo hacen, serán los problemas *decisivos*. La final mundial del ICPC es el lugar en que es más probable encontrar algunos de los problemas de grafos más complejos.

Pero hay buenas noticias para los concursantes de la IOI. Creemos que la mayoría de la materia de grafos incluida en el temario de la IOI ha quedado cubierta en este capítulo, y debería servir para obtener una buena puntuación en las tareas de la IOI que impliquen grafos. Aun así, es necesario dominar los algoritmos básicos tratados aquí y, después, mejorar tus habilidades en la resolución de problemas, mediante la aplicación de estos algoritmos básicos en problemas de grafos *creativos*, que suelen aparecer en la IOI, para resolver completamente la tarea.

---

<sup>26</sup>Invitamos al lector interesado a leer el artículo de Felix [23] sobre el algoritmo de flujo máximo para grafos grandes, de 411 millones de vértices y 31 mil millones de aristas.



# Capítulo 5

---

## Matemáticas

*Utilizamos las matemáticas a diario: para predecir el tiempo, para saber la hora o para manejar dinero. Las matemáticas son más que fórmulas o ecuaciones: son lógica y racionalidad, es utilizar la mente para resolver los mayores misterios conocidos.*

— Serie de televisión NUMB3RS

### 5.1 Introducción y motivación

La presencia en concursos de programación de problemas relacionados con las matemáticas no debería sorprender, puesto que la informática encuentra sus raíces más profundas en las matemáticas. El término ‘computadora’ viene de la palabra ‘computar’, ya que los equipos de computación se han construido, principalmente, para ayudar a los humanos a realizar cálculos numéricos. Muchos problemas interesantes de la vida real se pueden presentar como problemas matemáticos, como vamos a comprobar en este capítulo.

Los conjuntos de problemas más recientes del ICPC (sobre todo en Asia) suelen incluir uno o dos problemas de base matemática. Por otro lado, las IOI más recientes no incluyen tareas *estrictamente* matemáticas, pero muchas requieren de conocimientos relacionados. Este capítulo pretende formar a los concursantes en la resolución de muchos de estos problemas.

Somos conscientes de que la preparación en matemáticas, en la educación anterior a la universitaria, varía sustancialmente de unos países a otros. Por lo tanto, algunos concursantes estarán familiarizados con los términos matemáticos de la tabla 5.1, mientras que, para otros, tales expresiones resultarán desconocidas. Quizá porque el concursante no lo haya aprendido antes, o quizás porque el término es diferente al que conoce. En este capítulo, queremos equilibrar las diferencias mencionadas, presentando a los lectores una lista de terminología, definiciones, problemas y algoritmos comunes, que aparecen frecuentemente en concursos de programación.

### 5.2 Problemas matemáticos *ad hoc*

Comenzamos el capítulo con algo ligero: los problemas matemáticos *ad hoc*. Son un tipo de problema que no suele requerir más que capacidades básicas de programación y un poco de matemáticas elementales. Como en esta categoría caben muchos problemas, los dividiremos en

|                            |                       |                         |
|----------------------------|-----------------------|-------------------------|
| Progresión aritmética      | Progresión geométrica | Polinomio               |
| Álgebra                    | Logaritmo/Potencia    | <code>BigInteger</code> |
| Combinatoria               | Fibonacci             | Razón áurea             |
| Fórmula de Binet           | Teorema de Zeckendorf | Números de Catalan      |
| Factorial                  | Desarreglo            | Coeficientes binomiales |
| Teoría de números          | Número primo          | Criba de Eratóstenes    |
| Criba modificada           | Miller-Rabin          | Función $f_i$ de Euler  |
| Máximo común divisor       | Mínimo común múltiplo | Euclides extendido      |
| Ecuación diofántica lineal | Búsqueda de ciclos    | Teoría de probabilidad  |
| Teoría de juegos           | Juego de suma cero    | Árbol de decisión       |
| Partida perfecta           | <i>Minimax</i>        | Juego del <i>nim</i>    |

Tabla 5.1: Lista de *algunos* términos matemáticos utilizados en este capítulo

las subcategorías que aparecen a continuación. Estos problemas no aparecen en la sección 1.4, ya que son *ad hoc* pero con un toque matemático. Puedes pasar directamente de la sección 1.4 a esta, si prefieres hacerlo así. Pero recuerda que muchos de estos problemas son los más fáciles. Para tener un buen rendimiento en los concursos de programación, los concursantes también deberían dominar *el resto de secciones* de este capítulo.

- Los más sencillos: unas pocas líneas de código por problema, para ganar confianza. Indicados para aquellos que nunca hayan resuelto problemas matemáticos.
- Simulación matemática (fuerza bruta): las soluciones a estos problemas se logran mediante la simulación del proceso matemático. Normalmente, la solución necesita algún tipo de bucle. Por ejemplo, dado un conjunto  $S$  de  $1M$  de números aleatorios y un entero  $X$ , ¿cuántos enteros de  $S$  son menores que  $X$ ? La respuesta es la fuerza bruta, comprobar el  $1M$  de enteros y contar cuántos de ellos son menores que  $X$ . Esta solución es un poco más rápida que ordenar primero el  $1M$  de enteros. En la sección 3.2, puedes revisar varias técnicas de búsqueda completa/fuerza bruta iterativas. También en la sección 3.2, se mencionan algunos problemas matemáticos cuya solución es de fuerza bruta.
- Buscar un patrón o fórmula: estos problemas requieren, para su solución, una cuidadosa lectura del enunciado del problema, que permita detectar un patrón o una fórmula simplificada. El ataque frontal resultará, casi siempre, en un veredicto de TLE. Las soluciones correctas suelen ser cortas y no requieren bucles ni llamadas recursivas. Por ejemplo, definamos  $S$  como un conjunto infinito de *enteros cuadrados*, ordenados de forma creciente:  $\{1, 4, 9, 16, 25, \dots\}$ . Dado un entero  $X$  ( $1 \leq X \leq 10^{17}$ ), determinar cuántos enteros de  $S$  son menores que  $X$ . La respuesta es  $\lfloor \sqrt{X} - 1 \rfloor$ .
- Rejilla: estos problemas implican la manipulación de rejillas. La rejilla puede ser compleja, pero sigue alguna reglas básicas. No incluimos aquí las rejillas ‘triviales’, de una o dos dimensiones. La solución depende, normalmente, de la creatividad de quien resuelve el problema para encontrar los patrones necesarios para manipular o explorar la rejilla, o para simplificarla.
- Sistemas numéricos o secuencias: algunos problemas matemáticos *ad hoc* implican la definición de sistemas numéricos o secuencias existentes (o ficticias), y nuestra tarea consiste en hallar bien el número (secuencia) dentro de un rango, bien el que ocupa la posición  $n$ -ésima, verificando que el número (secuencia) dado sea válido conforme a la definición.

Comprender adecuadamente el enunciado del problema es, normalmente, clave para resolverlo. Pero algunos de los problemas más difíciles nos obligan, en primer lugar, a simplificar la fórmula. Hay ejemplos muy conocidos, como:

1. Sucesión de Fibonacci (sección 5.4.1): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
  2. Factorial (sección 5.5.3): 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...
  3. Subfactorial (sección 9.8): 1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, ...
  4. Números de Catalan (sección 5.4.3): 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
  5. Series de progresiones aritméticas:  $a_1, (a_1 + d), (a_1 + 2 \times d), (a_1 + 3 \times d), \dots$ , por ejemplo, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ..., que comienza con  $a_1 = 1$ , y tiene una diferencia de  $d = 1$  entre términos consecutivos. La suma de los primeros  $n$  términos de estas series de progresiones aritméticas  $S_n = \frac{n}{2} \times (2 \times a_1 + (n - 1) \times d)$ .
  6. Series de progresiones geométricas como  $a_1, a_1 \times r, a_1 \times r^2, a_1 \times r^3, \dots$ , por ejemplo, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ..., que comienza con  $a_1 = 1$  y con una razón común de  $r = 2$ , entre términos consecutivos. La suma de los primeros  $n$  términos de estas series de progresiones geométricas  $S_n = a_1 \times \frac{1-r^n}{1-r}$ . Tiendo en cuenta que  $r > 1$ .
- Logaritmos, exponenciación, potencia. Estos problemas implican el uso (inteligente) de las funciones `log()` y/o `exp()`. En los ejercicios escritos de este capítulo, aparecen algunos de los más importantes.
  - Polinomios. Estos problemas implican evaluación, derivación, multiplicación o división polinómicas, entre otras. Podemos representar un polinomio almacenando los coeficientes de los términos polinómicos, ordenados por sus potencias (normalmente de forma descendente). Las operaciones con polinomios suelen implicar un uso cuidadoso de bucles.
  - Variaciones de bases numéricas. Son problemas matemáticos que implican bases numéricas, pero no los problemas de conversión *estándar*, que se pueden resolver fácilmente utilizando la clase `BigIntger` de Java (ver la sección 5.3).
  - Simplemente *ad hoc*. Hay otros problemas relacionados con las matemáticas que no se pueden clasificar en ninguna de las subcategorías anteriores.

Sugerimos al lector, especialmente a aquel recién llegado a los problemas matemáticos, que mejore su preparación en esta rama resolviendo, al menos, dos o tres problemas *de cada subcategoría*, sobre todo aquellos que resaltamos como obligatorio \*.

### Ejercicio 5.2.1

¿Qué deberíamos utilizar en C/C++/Java para calcular  $\log_b(a)$  (base  $b$ )?

### Ejercicio 5.2.2

¿Qué respuesta obtendremos de `(int)floor(1 + log10((double)a))`?

### Ejercicio 5.2.3

¿Cómo podemos calcular  $\sqrt[n]{a}$  (la  $n$ -ésima raíz de  $a$ ) en C/C++/Java?

### Ejercicio 5.2.4\*

Estudia el método de (Ruffini-)Horner para la búsqueda de las raíces de una ecuación polinómica  $f(x) = 0$ .

### Ejercicio 5.2.5\*

Dados  $1 < a < 10, 1 \leq n \leq 100000$ , muestra cómo calcular el valor de  $1 \times a + 2 \times a^2 + 3 \times a^3 + \dots + n \times a^n$ , de forma eficiente, es decir, en  $O(\log n)$ . Tanto  $a$  como  $n$  son enteros. La solución ingenua en  $O(n)$  no es aceptable.

## Ejercicios de programación

Ejercicios de programación relacionados con problemas matemáticos *ad hoc*:

### Los más fáciles

1. UVa 10055 - Hashmat the Brave Warrior (función absoluta; el único misterio es usar `long long`)
2. UVa 10071 - Back to High School ... (muy sencillo: mostrar  $2 \times v \times t$ )
3. UVa 10281 - Average Speed (distancia = velocidad  $\times$  tiempo transcurrido)
4. UVa 10469 - To Carry or not to Carry (muy sencillo si usas `xor`)
5. **UVa 10773 - Back to Intermediate ... \*** (varios casos complicados)
6. UVa 11614 - Etruscan Warriors Never ... (buscar las raíces de una ecuación de segundo grado)
7. **UVa 11723 - Numbering Road \*** (matemáticas sencillas)
8. UVa 11805 - Bafana Bafana (existe una fórmula muy sencilla en  $O(1)$ )
9. **UVa 11875 - Brick Game \*** (obtener la mediana de una entrada ordenada)
10. UVa 12149 - Feynman (buscar el patrón; números cuadrados)
11. UVa 12502 - Three Families (hay que entender el juego de palabras)

### Simulación matemática (fuerza bruta), fáciles

1. UVa 00100 - The 3n + 1 problem (hacer lo que se pide; la única trampa está en que  $j$  puede ser  $< i$ )
2. UVa 00371 - Ackermann Functions (similar a UVa 100)
3. **UVa 00382 - Perfection \*** (hacer división tentativa)
4. UVa 00834 - Continued Fractions (hacer lo que se pide)
5. UVa 00906 - Rational Neighbor (calcular  $c$  desde  $d = 1$  hasta  $\frac{a}{b} < \frac{c}{d}$ )
6. **UVa 01225 - Digit Counting \*** (LA 3996 - Danang07;  $N$  es pequeño)

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7. UVa 10035 - Primary Arithmetic<br>8. <b>UVa 10346 - Peter's Smoke *</b><br>9. UVa 10370 - Above Average<br>10. UVa 10783 - Odd Sum<br>11. UVa 10879 - Code Refactoring<br>12. UVa 11150 - Cola<br>13. UVa 11247 - Income Tax Hazard<br>14. UVa 11313 - Gourmet Games<br>15. UVa 11689 - Soda Surpler<br>16. UVa 11877 - The Coco-Cola Store<br>17. UVa 11934 - Magic Formula<br>18. UVa 12290 - Counting Game<br>19. UVa 12527 - Different Digits | (contar el número de operaciones de acarreo)<br>(interesante problema de simulación)<br>(calcular la media; ver cuántos están por encima)<br>(el rango de entrada es muy pequeño; se resuelve por fuerza bruta)<br>(basta con usar fuerza bruta)<br>(similar a UVa 10346; cuidado con los casos límite)<br>(fuerza bruta para estar seguros)<br>(similar a UVa 10346)<br>(similar a UVa 10346)<br>(similar a UVa 10346)<br>(fuerza bruta sin más)<br>(que no haya -1 en la respuesta)<br>(probar todos; comprobar dígitos repetidos) |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Simulación matemática (fuerza bruta), difíciles

- |                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00493 - Rational Spiral<br>2. UVa 00550 - Multiplying by Rotation<br><br><b>3. UVa 00616 - Coconuts, Revisited *</b><br>4. UVa 00697 - Jack and Jill<br>5. UVa 00846 - Steps<br>6. UVa 10025 - The ? 1 ? 2 ? ...<br>7. UVa 10257 - Dick and Jane<br><br>8. UVa 10624 - Super Number<br><b>9. UVa 11130 - Billiard bounces *</b><br><br><b>10. UVa 11254 - Consecutive Integers *</b><br><br>11. UVa 11968 - In The Airport | (simular el proceso espiral)<br>(propiedad de multiplicación por rotación; probar de uno en uno empezando con un dígito)<br>(fuerza bruta hasta $\sqrt{n}$ ; obtener el patrón)<br>(formato en la salida y física básica)<br>(usa la fórmula de suma de progresión aritmética)<br>(empezar simplificando la fórmula; iterativo)<br>(podemos obtener por fuerza fruta las edades enteras de Spot, Puffy Yertle; necesita conocimientos matemáticos)<br>(backtracking con comprobación de divisibilidad)<br>(reflejar la mesa de billar a la derecha (y/o arriba), para que solo haya que tratar con una línea recta, en vez de con líneas que rebotan)<br>(usar la suma de progresión aritmética; fuerza bruta para todos los valores de $r$ desde $\sqrt{2n}$ hasta 1; detener en el primer $a$ válido)<br>(media; fabs; en caso de empate, elegir el más pequeño) |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Ver también algunos problemas matemáticos en la sección 3.2.

### Búsqueda de patrón o fórmula, fáciles

- |                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 10014 - Simple calculations<br>2. UVa 10170 - The Hotel with Infinite ...<br>3. UVa 10499 - The Land of Justice<br>4. UVa 10696 - f91<br><b>5. UVa 10751 - Chessboard *</b><br><br><b>6. UVa 10940 - Throwing Cards Away II *</b><br><br>7. UVa 11202 - The least possible effort<br><b>8. UVa 12004 - Bubble Sort *</b><br>9. UVa 12027 - Very Big Perfect Square | (deducir la fórmula necesaria)<br>(existe una fórmula de una línea)<br>(existe una fórmula sencilla)<br>(simplificación de fórmula muy sencilla)<br>(trivial para $N = 1$ y $N = 2$ ; empezar deduciendo la fórmula para $N > 2$ ; pista: usar la diagonal todo lo posible)<br>(buscar el patrón con solución de fuerza bruta, después enviar la solución optimizada)<br>(considerar simetría y reflejo)<br>(probar con $n$ pequeño; obtener el patrón; usar long long)<br>(truco con la raíz cuadrada) |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Búsqueda de patrón o fórmula, difíciles

1. UVa 00651 - Deck  
(usar el caso de ejemplo para deducir la fórmula)
2. UVa 00913 - Joana and The Odd ...  
(deducir las fórmulas, que son cortas)
3. **UVa 10161 - Ant on a Chessboard \***  
(implica raíz cuadrada)  
(árbol; deducir la fórmula)  
(solo hay tres casos diferentes)  
(analizar la representación binaria de  $X$ )  
(deducir la breve fórmula física)
4. UVa 10493 - Cats, with or without Hats  
(fórmula un poco más difícil de deducir; modPow;  
ver las secciones 5.3 o 9.21)
5. UVa 10509 - R U Kidding Mr. ...  
(principio de inclusión-exclusión)
6. UVa 10666 - The Eurocup is here  
(existe una fórmula directa; también con DP)
7. UVa 10693 - Traffic Volume  
(simplificación de la fórmula)
8. UVa 10710 - Chinese Shuffle  
(existe una fórmula en  $O(1)$ )  
(deducir la fórmula)
9. UVa 10882 - Koerner's Pub  
(existe una fórmula sencilla)
10. UVa 10970 - Big Chocolate  
(matemática simple; empezar deduciendo el patrón)
11. UVa 10994 - Simple Addition  
(imposible si  $n$  es impar o 2; si  $n$  múltiplo de 4, considerar el grafo completo  $K_4$ ; si  $n = 6 + k \times 4$ , considerar un componente cúbico de 6 vértices y el resto son  $K_4$ , como en el caso anterior)
12. **UVa 11231 - Black and White Painting \***  
(dibujar varios  $K_n$  pequeños; deducir el patrón)  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)
13. UVa 11246 - K-Multiple Free Set  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)
14. UVa 11296 - Counting Solutions to an ...  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)
15. UVa 11298 - Dissecting a Hexagon  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)
16. UVa 11387 - The 3-Regular Graph  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)
17. UVa 11393 - Tri-Isomorphism  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)
18. **UVa 11718 - Fantasy of a Summation \***  
(convertir bucles a una fórmula cerrada; usar modPow para calcular el resultado; ver las secciones 5.3 o 9.21)

## Rejillas

1. **UVa 00264 - Count on Cantor \***  
(rejilla; patrón)
2. UVa 00808 - Bee Breeding  
(rejilla; similar a UVa 10182)
3. UVa 00880 - Cantor Fractions  
(rejilla; similar a UVa 264)
4. **UVa 10182 - Bee Maja \***  
(rejilla)
5. **UVa 10233 - Dermuba Triangle \***  
(el número de elementos en la fila forma series de progresiones aritméticas; usar hypo t)
6. UVa 10620 - A Flea on a Chessboard  
(basta simular los saltos)
7. UVa 10642 - Can You Solve It?  
(lo contrario de UVa 264)
8. UVa 10964 - Strange Planet  
(convertir las coordenadas a  $(x, y)$ ; después, el problema es sobre la búsqueda de la distancia euclídea entre dos coordenadas)
9. SPOJ 3944 - Bee Walk  
(un problema con una rejilla)

## Sistemas numéricos o secuencias

1. UVa 00136 - Ugly Numbers  
(usar una técnica similar a UVa 443)
2. UVa 00138 - Street Numbers  
(fórmula de progresión aritmética; cálculo previo)
3. UVa 00413 - Up and Down Sequences  
(simular; manipulación de arrays)
4. **UVa 00443 - Humble Numbers \***  
(probar todos los  $2^i \times 3^j \times 5^k \times 7^l$ ; ordenar)
5. UVa 00640 - Self Numbers  
(DP de abajo a arriba; generar los números; etiquetar una vez)
6. UVa 00694 - The Collatz Sequence  
(similar a UVa 100)

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>7. UVa 00962 - Taxicab Numbers<br/>     8. UVa 00974 - Kaprekar Numbers<br/>     9. UVa 10006 - Carmichael Numbers<br/> <b>10. UVa 10042 - Smith Numbers *</b><br/>     11. UVa 10049 - Self-describing Sequence</p> <p>12. UVa 10101 - Bangla Numbers<br/> <b>13. UVa 10408 - Farey Sequences *</b></p> <p>14. UVa 10930 - A-Sequence<br/>     15. UVa 11028 - Sum of Product<br/>     16. UVa 11063 - B2 Sequences<br/>     17. UVa 11461 - Square Numbers<br/>     18. UVa 11660 - Look-and-Say sequences<br/>     19. UVa 11970 - Lucky Numbers</p> | <p>(calcular previamente la respuesta)<br/>     (no hay tantos números de Kaprekar)<br/>     (no primo que tiene <math>\geq 3</math> factores primos)<br/>     (factorización de primos; sumar los dígitos)<br/>     (suficiente para pasar de 2G almacenando solo los primeros <math>700K</math> números de la secuencia)<br/>     (seguir el enunciado del problema cuidadosamente)<br/>     (primero, generar <math>(i, j)</math> pares de forma que <math>\gcd(i, j) = 1</math>; después ordenar)<br/> <math>(ad hoc;</math> seguir las reglas del enunciado)<br/>     (es una ‘secuencia de diana’)<br/>     (ver si se repite un número; cuidado con los negativos)<br/>     (la respuesta es <math>\sqrt{b} - \sqrt{a - 1}</math>)<br/>     (simular; parar después del carácter <math>j</math>-ésimo)<br/>     (números cuadrados; divisibilidad; fuerza bruta)</p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Logaritmos, exponentiación, potencia

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>1. UVa 00107 - The Cat in the Hat<br/>     2. UVa 00113 - Power Of Cryptography<br/>     3. UVa 00474 - Heads Tails Probability<br/>     4. UVa 00545 - Heads<br/> <b>5. UVa 00701 - Archaeologist's Dilemma *</b><br/>     6. UVa 01185 - BigNumber</p> <p>7. <b>UVa 10916 - Factstone Benchmark *</b><br/>     8. UVa 11384 - Help is needed for Dexter</p> <p>9. UVa 11556 - Best Compression Ever<br/>     10. UVa 11636 - Hello World<br/>     11. UVa 11666 - Logarithms<br/>     12. UVa 11714 - Blind Sorting</p> <p>13. <b>UVa 11847 - Cut the Silver Bar *</b><br/>     14. UVa 11986 - Save from Radiation<br/>     15. UVa 12416 - Excessive Space Remover</p> | <p>(usar logaritmos; potencia)<br/>     (usar <math>\exp(\ln(x) \times y))</math><br/>     (es solo un ejercicio de <math>\log</math> y <math>\text{pow}</math>)<br/>     (usar logaritmos; potencia; similar a UVa 474)<br/>     (usar logaritmos para contar el número de dígitos)<br/>     (número de dígitos de un factorial; usar logaritmos para resolverlo;<br/> <math>\log(n!) = \log(n \times (n - 1) \dots \times 1) =</math><br/> <math>\log(n) + \log(n - 1) + \dots + \log(1))</math><br/>     (usar logaritmos; potencia)<br/>     (buscar la potencia más pequeña de 2 que sea mayor que <math>n</math>; se puede resolver fácilmente con <math>\text{ceil}(eps + \log_2(n))</math>)<br/>     (relativo a potencias de 2; usar <code>long long</code>)<br/>     (usar logaritmos)<br/>     (encuentra la fórmula)<br/>     (usar un modelo de árbol de decisión para buscar los mínimos;<br/>     en algún momento la solución solo necesita logaritmos)<br/>     (existe una fórmula en <math>O(1)</math>: <math>\lfloor \log_2(n) \rfloor</math>)<br/> <math>(\log_2(N + 1);</math> comprobar manualmente la precisión)<br/> <math>(\log_2</math> del máximo de espacios consecutivos en una línea)</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Polinomios

- |                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>1. UVa 00126 - The Errant Physicist<br/>     2. UVa 00392 - Polynomial Showdown<br/> <b>3. UVa 00498 - Polly the Polynomial *</b><br/>     4. UVa 10215 - The Largest/Smallest Box<br/> <b>5. UVa 10268 - 498' *</b></p> | <p>(multiplicación polinómica y formato de salida complicado)<br/>     (seguir los órdenes; formato de salida)<br/>     (evaluación polinómica)<br/>     (dos casos triviales para la más pequeña; deducir la fórmula de la más grande, que implica una ecuación de segundo grado)<br/>     (derivación polinómica; regla de Horner)</p> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- |                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>6. UVa 10302 - Summation of Polynomials</li> <li>7. UVa 10326 - <i>The Polynomial Equation</i></li> <li>8. <b>UVa 10586 - Polynomial Remains *</b></li> <li>9. UVa 10719 - Quotient Polynomial</li> <li>10. UVa 11692 - <i>Rain Fall</i></li> </ol> | <p>(usar <code>long double</code>)</p> <p>(dadas sus raíces, reconstruir el polinomio; formato)</p> <p>(división; manipular coeficientes)</p> <p>(división polinómica y resto)</p> <p>(usar manipulación algebraica para deducir la ecuación de segundo grado; resolverlo; hay un caso especial cuando <math>H &lt; L</math>)</p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Variaciones de bases numéricas

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. <b>UVa 00377 - Cowculations *</b></li> <li>2. <b>UVa 00575 - Skew Binary *</b></li> <li>3. UVa 00636 - Squares</li> <br/> <li>4. UVa 10093 - An Easy Problem</li> <li>5. UVa 10677 - Base Equality</li> <li>6. <b>UVa 10931 - Parity *</b></li> <li>7. UVa 11005 - Cheapest Base</li> <li>8. UVa 11121 - Base -2</li> <li>9. <i>UVa 11398 - The Base-1 Number System</i></li> <li>10. <i>UVa 12602 - Nice Licence Plates</i></li> <li>11. <i>SPOJ 0739 - The Moronic Cowmpouter</i></li> <li>12. IOI 2011 - Alphabets</li> </ol> | <p>(operaciones en base 4)</p> <p>(modificación de base)</p> <p>(conversión de bases hasta base 99; no se puede usar BigInteger de Java porque su MAX_RADIX está limitada a 36)</p> <p>(probar todos)</p> <p>(probar todos desde r2 a r1)</p> <p>(conversión de decimal a binario; contar el número de unos)</p> <p>(probar todas las bases posibles desde 2 a 36)</p> <p>(busca el término 'negabinario')</p> <p>(basta seguir las nuevas reglas)</p> <p>(conversión de bases sencilla)</p> <p>(buscar la representación en base -2)</p> <p>(tarea de práctica; usar la base 26, más eficiente en espacio)</p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Simplemente *ad hoc*

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. UVa 00276 - Egyptian Multiplication</li> <li>2. UVa 00496 - Simply Subsets</li> <li>3. <i>UVa 00613 - Numbers That Count</i></li> <br/> <li>4. <b>UVa 10137 - The Trip *</b></li> <li>5. UVa 10190 - Divide, But Not Quite ...</li> <li>6. <i>UVa 11055 - Homogeneous Square</i></li> <li>7. <i>UVa 11241 - Humidex</i></li> <br/> <li>8. <b>UVa 11526 - H(n) *</b></li> <li>9. UVa 11715 - Car</li> <li>10. UVa 11816 - HST</li> <li>11. <b>UVa 12036 - Stable Grid *</b></li> </ol> | <p>(multiplicación de jeroglíficos egipcios)</p> <p>(manipulación de conjuntos)</p> <p>(analizar el número; determinar el tipo; concepto similar al problema de búsqueda de ciclos)</p> <p>(cuidado con el error de precisión)</p> <p>(simular el proceso)</p> <p>(no clásico; se necesita observación para evitar la solución de fuerza bruta)</p> <p>(el caso más difícil es calcular el punto de rocío a partir de la temperatura y el índice de humedad; deducirlo con álgebra)</p> <p>(fuerza bruta hasta <math>\sqrt{n}</math>; buscar el patrón; evitar TLE)</p> <p>(simulación de física)</p> <p>(matemáticas sencillas; se requiere precisión)</p> <p>(usar el principio del palomar)</p> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 5.3 Clase BigInteger de Java

### 5.3.1 Características básicas

Cuando los resultados intermedios o finales de un cálculo matemático con enteros no se pueden almacenar en el tipo de datos integrado más grande, y el problema no se puede resolver con técnicas de descomposición en factores primos (sección 5.5.5) o de aritmética modular (sección 5.5.8), no nos queda otra opción que recurrir a bibliotecas BigInteger (también conocidas como *bignum*). Por ejemplo, calcular el *valor exacto* de  $25!$  (el factorial de 25). El resultado es  $15.511.210.043.330.985.984.000.000$  (26 dígitos). Este número es, evidentemente, demasiado grande para almacenarlo en un `unsigned long long` de 64 bits en C/C++ (`long` en Java).

Un método para implementar la biblioteca BigInteger, consiste en almacenar el entero como una cadena (larga)<sup>1</sup>. Por ejemplo, podemos almacenar sin problema  $10^{21}$  como una cadena `num1 = "1.000.000.000.000.000.000"`, mientras que el mismo valor desbordaría el tipo `unsigned long long` de 64 bits de C/C++ (o el `long` de Java). Por lo tanto, en las operaciones matemáticas comunes, puedes utilizar un sistema de operaciones dígito a dígito para procesar los dos operandos BigInteger. Por ejemplo, si `num2 = "173"`, tenemos que `num1 + num2` resulta en:

$$\begin{array}{rcl} \text{num1} & = & 1.000.000.000.000.000.000 \\ \text{num2} & = & 173 \\ \hline & & + \\ \text{num1} + \text{num2} & = & 1.000.000.000.000.000.000.173 \end{array}$$

También podemos calcular `num1 * num2` como:

$$\begin{array}{rcl} \text{num1} & = & 1.000.000.000.000.000.000 \\ \text{num2} & = & 173 \\ \hline & & * \\ & & 3.000.000.000.000.000.000 \\ & & 70.000.000.000.000.000.000 \\ & & 100.000.000.000.000.000.000 \\ \hline & & + \\ \text{num1} * \text{num2} & = & 173.000.000.000.000.000.000 \end{array}$$

La suma y la resta son las dos operaciones más sencillas para realizar con BigInteger. La multiplicación requerirá un poco más de programación, como se deduce del ejemplo anterior. Implementar una división eficiente y elevar un entero a una potencia (ver sección 9.21), resulta más complicado. Programar estas funciones de biblioteca en C/C++ en un entorno de presión, como el de un concurso, puede ser un auténtico lío, incluso aunque tengamos una referencia escrita entre nuestras notas en ICPC<sup>2</sup>. Afortunadamente, Java cuenta con una clase BigInteger que podemos utilizar para este propósito. A día de hoy, la STL de C++ todavía no tiene una

<sup>1</sup>En realidad, los tipos de datos elementales también almacenan los números en memoria como *cadenas de bits limitadas*. Por ejemplo, el tipo de datos `int` de 32 bits, almacena un entero como 32 bits de una cadena binaria. La técnica de BigInteger no es más que una generalización de lo anterior, que utiliza un formato decimal (base 10) y una cadena de dígitos más larga. Nota: la clase BigInteger de Java utiliza, probablemente, un método más eficiente que el presentado en esta sección.

<sup>2</sup>Buenas noticias para los concursantes de la IOI, ya que las tareas no suelen necesitar el uso de BigInteger.

característica comparable, por lo que es una buena idea utilizar Java en los problemas que incluyan BigInteger.

La clase BigInteger de Java (o BI), cuenta con operaciones básicas con enteros: suma (`add(BI)`), resta (`subtract(BI)`), multiplicación (`multiply(BI)`), potencia (`pow(int exponente)`), división (`divide(BI)`), resto (`remainder(BI)`), módulo (`mod(BI)`), diferente a `remainder(BI)`, división y resto (`divideAndRemainder(BI)`) y otras funciones interesantes que veremos después. Todas se pueden expresar en una sola línea de código.

Debemos, sin embargo, remarcar que todas las operaciones con BigInteger son *significativamente más lentas* que sus equivalentes con tipos de datos enteros de 32/64 bits estándar. Como norma: si puedes utilizar un algoritmo que solo necesite tipos de datos de enteros tradicionales para resolver el problema, evita recurrir a BigInteger.

Para los recién llegados a la clase BigInteger de Java, incluimos el siguiente código breve, que resuelve el problema UVa 10925 - Krakovia. Este problema utiliza la suma de BigInteger (para agregar  $N$  facturas grandes) y la división (para distribuir el resultado de la suma entre  $F$  amigos). Se puede observar lo corto y limpio que resulta el código, en comparación al que sería necesario si se implementasen rutinas de BigInteger propias.

```
1 import java.util.Scanner; // en el paquete java.util
2 import java.math.BigInteger; // en el paquete java.math
3
4 class Main { // UVa 10925 - Krakovia
5 public static void main(String[] args) { // Nota: no podemos usar
6 Scanner sc = new Scanner(System.in); // BufferedReader, más rápido
7 int caseNo = 1;
8 while (true) {
9 int N = sc.nextInt(), F = sc.nextInt(); // N facturas, F amigos
10 if (N == 0 && F == 0) break;
11 BigInteger sum = BigInteger.ZERO; // BigInteger tiene esta constante
12 for (int i = 0; i < N; i++) { // sumar las N facturas grandes
13 BigInteger V = sc.nextBigInteger(); // y leer siguiente BigInteger
14 sum = sum.add(V); // esto es la suma con BigInteger
15 }
16 System.out.println("Bill #" + (caseNo++) + " costs " + sum +
17 ": each friend should pay " + sum.divide(BigInteger.valueOf(F)));
18 System.out.println(); // lo anterior es la división con BigInteger
19 } } }
```



ch5\_01\_UVa10925.java

### Ejercicio 5.3.1.1

Calcula el último dígito de  $25!$  que no sea un 0. ¿Se podría utilizar aquí un tipo de datos integrado en el lenguaje?

## Ejercicio 5.3.1.2

Comprueba si 25! es divisible entre 9317. ¿Podemos utilizar un tipo de datos integrado?

### 5.3.2 Características adicionales

La clase BigInteger de Java cuenta con algunas características adicionales, que pueden resultar útiles durante los concursos de programación, en términos de brevedad del código, en comparación a la posibilidad de escribir nosotros mismos las funciones<sup>3</sup>. Resulta que la clase BigInteger de Java incluye un conversor de bases: el constructor de la clase y la función `toString(int radix)`, una buena función de prueba de primalidad (probabilística): `isProbablePrime(int certainty)`, una rutina de máximo común divisor: `gcd(BI)` y una función de aritmética modular: `modPow(BI exponent, BI m)`. De todas estas características, el conversor de bases es la más útil, seguida del comprobador de primos. Mostramos las características adicionales utilizando cuatro problemas de ejemplo del Online Judge.

#### Conversión de bases

Veamos, a continuación, un ejemplo para UVa 10551 - Basic Remains. Dada una base  $b$  y dos enteros no negativos  $p$  y  $m$  (ambos en base  $b$ ), calcular  $p \% m$ , y mostrar el resultado como un entero en base  $b$ . La conversión de bases no es, en realidad, un problema matemático tan difícil<sup>4</sup>, pero se puede simplificar aún más con la clase BigInteger de Java. Podemos construir y mostrar una instancia de BigInteger de Java en cualquier base ( $radix$ ), como se muestra:

```
1 class Main { // UVa 10551 - Basic Remains
2 public static void main(String[] args) {
3 Scanner sc = new Scanner(System.in);
4 while (true) {
5 int b = sc.nextInt();
6 if (b == 0) break; // constructor de clase especial
7 BigInteger p = new BigInteger(sc.next(), b); // el segundo parámetro
8 BigInteger m = new BigInteger(sc.next(), b); // es la base
9 System.out.println((p.mod(m)).toString(b)); // posible cualquier base
10 } } }
```



ch5\_02\_UVa10551.java

<sup>3</sup>Nota para los programadores de C/C++: es bueno ser capaz de programar en varios lenguajes, cambiando a Java cuando resulte beneficioso.

<sup>4</sup>Por ejemplo, para convertir 132 en base 8 (octal) a base 2 (binario), podemos utilizar la base 10 (decimal) como paso intermedio:  $(132)_8$  es  $1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 64 + 24 + 2 = (90)_{10}$  y  $(90)_{10}$  es  $90 \rightarrow 45(0) \rightarrow 22(1) \rightarrow 11(0) \rightarrow 5(1) \rightarrow 2(1) \rightarrow 1(0) \rightarrow 0(1) = (1011010)_2$  (es decir, dividimos por 2 hasta llegar a 0, después leemos el resto desde atrás).

## Prueba de primalidad (probabilística)

Más adelante, en la sección 5.5.1, trataremos el algoritmo de la criba de Eratóstenes y un algoritmo de prueba de primalidad determinista, que resultará suficiente para la mayoría de concursos de programación. Sin embargo, será necesario escribir unas cuantas líneas de C/C++/Java para hacerlo funcionar. Si, únicamente, necesitas comprobar que un número entero (o, como mucho, unos pocos<sup>5</sup>) y, normalmente, bastante grande es primo, como por ejemplo en UVa 10235, existe la alternativa más cómoda de la función `isProbablePrime` en `BigInteger` de Java (una función de prueba de primalidad probabilística, basada en el algoritmo de Miller-Rabin [44, 55]). Hay un parámetro muy importante en esta función: `certainty`. Si la respuesta a la llamada es verdadera, la probabilidad de que el número `BigInteger` comprobado sea primo superará  $1 - \frac{1}{2}^{\text{certainty}}$ . En un problema típico de un concurso, `certainty = 10` debería ser suficiente, ya que  $1 - (\frac{1}{2})^{10} = 0,9990234375$  es  $\approx 1,0$ . Es relevante el hecho de que un valor mayor de `certainty` reduce, evidentemente, la probabilidad de un veredicto de WA, pero hacerlo ralentizará el programa y aumentará el riesgo de TLE. Puedes convencerte de ello resolviendo el **ejercicio 5.3.2.3\***.

```
1 class Main { // UVa 10235 - Simply Emirp
2 public static void main(String[] args) throws Exception {
3 BufferedReader br = new BufferedReader(
4 new InputStreamReader(System.in));
5 PrintWriter pw = new PrintWriter(
6 new BufferedWriter(new OutputStreamWriter(System.out)));
7 while (true) {
8 String N = br.readLine();
9 if (N == null) break;
10 BigInteger BN = new BigInteger(N);
11 BigInteger BRN = new BigInteger(
12 new StringBuffer(BN.toString()).reverse().toString());
13 pw.printf("%s is ", N);
14 if (!BN.isProbablePrime(10)) // certainty 10 suele ser suficiente
15 pw.printf("not prime.\n");
16 else if (!BN.equals(BRN) && BRN.isProbablePrime(10))
17 pw.printf("emirp.\n");
18 else
19 pw.printf("prime.\n");
20 }
21 pw.close();
22 }
}
```



ch5\_03\_UVa10235.java

<sup>5</sup>Tengamos en cuenta que, si el objetivo es generar una lista de los primeros millones de números primos, el algoritmo de la criba de Eratóstenes, que aparece en la sección 5.5.1, será más rápido que unos millones de llamadas a la función `isProbablePrime`.

## Máximo común divisor (GCD)

A continuación, podemos ver un ejemplo para el problema UVa 10814 - Simplifying Fractions. Se nos pide reducir una fracción grande a su forma más simple, dividiendo el numerador y el denominador por su GCD. En la sección 5.5.2 hay más detalles sobre el GCD.

```
1 class Main { /* UVa 10814 - Simplifying Fractions */
2 public static void main(String[] args) {
3 Scanner sc = new Scanner(System.in);
4 int N = sc.nextInt();
5 while (N-- > 0) { // distinto a C/C++, hay que usar > 0 en (N-- > 0)
6 BigInteger p = sc.nextBigInteger();
7 String ch = sc.next(); // ignoramos el signo de división en entrada
8 BigInteger q = sc.nextBigInteger();
9 BigInteger gcd_pq = p.gcd(q); // alucinante
10 System.out.println(p.divide(gcd_pq) + " / " + q.divide(gcd_pq));
11 } } }
```



ch5\_04\_UVa10814.java

## Aritmética modular

Incluimos un ejemplo de código para el problema UVa 1230 (LA 4104) - MODEX, que calcula  $x^y \pmod n$ . En las secciones 5.5.8 y 9.21 se puede ver el proceso de cálculo de la función `modPow`.

```
1 class Main { // UVa 1230 (LA 4104) - MODEX
2 public static void main(String[] args) {
3 Scanner sc = new Scanner(System.in);
4 int c = sc.nextInt();
5 while (c-- > 0) {
6 BigInteger x = BigInteger.valueOf(sc.nextInt()); // valueOf convierte
7 BigInteger y = BigInteger.valueOf(sc.nextInt()); // un entero simple
8 BigInteger n = BigInteger.valueOf(sc.nextInt()); // en un BigInteger
9 System.out.println(x.modPow(y, n)); // está en la biblioteca
10 } } }
```



ch5\_05\_UVa1230.java

### Ejercicio 5.3.2.1

Intenta resolver el problema UVa 389, utilizando la técnica de BigInteger de Java que hemos comentado. ¿Podrás entrar dentro del límite de tiempo? Si no, ¿hay otra técnica que sea (un poco) mejor?

### Ejercicio 5.3.2.2\*

A día de hoy, los problemas de concursos de programación que implican números decimales de *precisión arbitraria* (no necesariamente enteros) siguen siendo poco habituales. De hecho, solo hemos identificado dos en el juez en línea UVa que lo necesiten: UVa 10464 y UVa 11821. Intenta resolver estos problemas utilizando otra biblioteca: la clase BigDecimal de Java. Encontrarás información en: <http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>.

### Ejercicio 5.3.2.3\*

Escribe un programa en Java que determine *empíricamente* el valor mínimo del parámetro `certainty`, de forma que nuestro programa sea más rápido y no haya *números compuestos* entre [2..10M], un rango típico en un problema de un concurso, que sean detectados accidentalmente como números primos por `isProbablePrime(certainty)`. Como `isProbablePrime` utiliza un algoritmo probabilístico, deberás repetir el experimento varias veces para cada valor de `certainty`. ¿Será `certainty = 5` suficiente? ¿Y `certainty = 10`? ¿Qué pasa con `certainty = 1000`?

### Ejercicio 5.3.2.4\*

Estudia e implementa el algoritmo de Miller Rabin (ver [44, 55]), en caso de que tengas que hacerlo en C/C++.

## Ejercicios de programación

Ejercicios de programación relacionados con BigInteger <sup>NO<sup>6</sup></sup> mencionados en otra parte:

### Características básicas

1. UVa 00424 - Integer Inquiry  
(suma de BigInteger)
2. UVa 00465 - Overflow  
(suma/multiplicación de BigInteger; comparar con  $2^{31} - 1$ )
3. UVa 00619 - Numerically Speaking  
(BigInteger)
4. **UVa 00713 - Adding Reversed ... \***  
(BigInteger y reverse() de StringBuffer)  
(exponenciación de BigInteger)
5. UVa 00748 - Exponentiation  
(LA 3997 - Danang07; operación con módulo)
6. UVa 01226 - Numerical surprises  
(suma de BigInteger)
7. UVa 10013 - Super long sums  
(BigInteger y teoría de números)
8. UVa 10083 - Division  
(multiplicación de BigInteger)
9. UVa 10106 - Product  
(recurrencias; BigInteger)
10. UVa 10198 - Counting  
(BigInteger; deducir primero la fórmula)
11. *UVa 10430 - Dear GOD*  
(BigInteger: pow, subtract, mod)
12. *UVa 10433 - Automorphic Numbers*  
(división de BigInteger)
13. UVa 10494 - If We Were a Child Again  
(recurrencias; BigInteger)
14. UVa 10519 - Really Strange  
(suma, multiplicación y potencia de BigInteger)
15. **UVa 10523 - Very Easy \***  
(BigInteger es para  $3^n$ ; representación binaria del conjunto)
16. UVa 10669 - Three powers  
(suma y división de BigInteger)
17. UVa 10925 - Krakovia  
(tamaño de entrada de hasta 50 dígitos)
18. *UVa 10992 - The Ghost of Programmers*  
(resta de BigInteger)
19. UVa 11448 - Who said crisis?  
(simulación sencilla que implica BigInteger)
20. *UVa 11664 - Langton's Ant*  
(usar representación de cadenas con BigInteger)
21. UVa 11830 - Contract revision  
(BigInteger: mod, divide, subtract, equals)
22. **UVa 11879 - Multiple of 17 \***  
(LA 4209 - Dhaka08; lo difícil, simplificación de la fórmula; lo fácil, BigInteger)
23. *UVa 12143 - Stopping Doom's Day*  
(dibujar el árbol de ancestros para ver el patrón)
24. *UVa 12459 - Bees' ancestors*

### Característica adicional: conversión de bases numéricas

1. UVa 00290 - Palindroms  $\longleftrightarrow \dots$   
(también implica palíndromos)
2. **UVa 00343 - What Base Is This? \***  
(probar todas las parejas de bases posibles)
3. UVa 00355 - The Bases Are Loaded  
(conversión básica de base numérica)
4. **UVa 00389 - Basically Speaking \***  
(usar la clase Integer de Java)
5. UVa 00446 - Kibbles 'n' Bits 'n' Bits ...  
(conversión de base numérica)
6. UVa 10473 - Simple Base Conversion  
(decimal a hexadecimal y viceversa; si usas C/C++, te sirve strtol)
7. **UVa 10551 - Basic Remains \***  
(también módulo de BigInteger)
8. UVa 11185 - Ternary  
(decimal a base 3)
9. UVa 11952 - Arithmetic  
(comprobar bases de 2 a 18; caso especial para base 1)

<sup>6</sup>Hay muchos otros ejercicios de programación, en otras secciones de este capítulo (y también en los otros), que también utilizan BigInteger.

### Característica adicional: prueba de primalidad

1. [UVa 00960 - Gaussian Primes](#) (aquí hay teoría de números)
2. [UVa 01210 - Sum of Consecutive ... \\*](#) (LA 3399 - Tokyo05; sencillo)
3. [UVa 10235 - Simply Emirp \\*](#) (análisis de casos: primo/omirp/no primo; omirp es un número primo que, si se invierte, sigue siendo primo)
4. UVa 10924 - Prime Words (comprobar si la suma del valor de las letras es un primo)
5. [UVa 11287 - Pseudoprime Numbers \\*](#) (mostar'sí' si ! isPrime(p) + a.modPow(p, p) = a; usar BigInteger de Java) (LA 6149 - HatYai12; fuerza bruta; usar isProbablePrime para probar la primalidad)
6. [UVa 12542 - Prime Substring](#)

### Característica adicional: otros

1. [UVa 01230 - MODEX \\*](#) (LA 4104 - Singapore07; modPow)
2. [UVa 10023 - Square root](#) (programar el método de Newton con BigInteger)
3. UVa 10193 - All You Need Is Love (convertir dos cadenas binarias S1 y S2 a decimales y comprobar si  $gcd(s1, s2) > 1$ )
4. UVa 10464 - Big Big Real Numbers (clase BigDecimal de Java)
5. [UVa 10814 - Simplifying Fractions \\*](#) (GCD con BigInteger)
6. [UVa 11821 - High-Precision Number \\*](#) (clase BigDecimal de Java)

## Perfiles de los inventores de algoritmos

**Gary Lee Miller** es profesor de ciencias de la computación en la Carnegie Mellon University. Es el inventor inicial del algoritmo de prueba de primalidad de Miller-Rabin.

**Michael Oser Rabin** (nacido en 1931) es un científico de la computación israelí. Mejoró la idea de Miller e inventó el algoritmo de prueba de primalidad de Miller-Rabin. También inventó, junto a Richard Manning Karp, el algoritmo de coincidencia de cadenas de Rabin-Karp.

## 5.4 Combinatoria

La **combinatoria** es una rama de la *matemática discreta*<sup>7</sup> que trata el estudio de estructuras discretas **numerable**s. En los concursos de programación, aquellas cuestiones que implican combinatoria se suelen titular ‘¿Cuántos [objeto]?’ , ‘Contar [objeto]’ , etc., aunque algunos autores de problemas prefieren ocultar el hecho en los títulos. El código de la solución suele ser *corto*, pero encontrar la fórmula (normalmente recursiva) requiere algo de brillantez matemática y también paciencia.

En el ICPC<sup>8</sup>, si encontramos uno de estos problemas, deberemos asignar a un miembro del

<sup>7</sup>La matemática discreta consiste en el estudio de estructuras discretas (por ejemplo, enteros  $\{0, 1, 2, \dots\}$ , grafos/árboles (vértices y aristas), lógica (verdadero/falso)), en contraposición a aquellas que son continuas (por ejemplo, los números reales).

<sup>8</sup>En la IOI no es habitual encontrar problemas de combinatoria pura pero, en ocasiones, puede ser parte de una tarea mayor.

equipo que sea eficaz en matemáticas la búsqueda de la fórmula, mientras los otros dos se concentran en *otros* problemas. Una vez obtenida la fórmula, se puede programar rápidamente (interrumpiendo a quien esté utilizando el ordenador en ese momento). También es una buena idea memorizar/estudiar las fórmulas más comunes, como las relacionadas con Fibonacci (sección 5.4.1), coeficientes binomiales (sección 5.4.2) y números de Catalan (sección 5.4.3).

Algunas de estas fórmulas de combinatoria pueden implicar subproblemas superpuestos, que necesiten el uso de programación dinámica (sección 3.5). También podemos encontrarnos valores lo suficientemente grandes como para requerir técnicas de BigInteger (sección 5.3).

#### 5.4.1 Sucesión de Fibonacci

La sucesión de Leonardo *Fibonacci* se define como  $fib(0) = 0$ ,  $fib(1) = 1$  y, para  $n \geq 2$ ,  $fib(n) = fib(n-1) + fib(n-2)$ . Esto produce la siguiente secuencia, bastante conocida: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, etc. Esta sucesión aparece, en ocasiones, en problemas que no mencionan en absoluto el término ‘Fibonacci’, como en algunos de los ejercicios de programación de esta sección (UVa 900, 10334, 10450, 10497, 10862, etc.).

Normalmente, deducimos la sucesión de Fibonacci con una técnica ‘trivial’ de programación dinámica en  $O(n)$ , en vez de implementar la recurrencia directamente (ya que resulta muy lento). Sin embargo, la solución de DP  $O(n)$  *no* siempre es la más rápida. En la sección 9.21 veremos cómo calcular el  $n$ -ésimo término de la sucesión de Fibonacci (con un  $n$  grande) en tiempo  $O(\log n)$ , utilizando la eficiente potencia de matrices. Como apunte, existe una fórmula cerrada  $O(\log n)$  para obtener el término  $n$ -ésimo: calculamos el entero más cercano de  $\frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$  (fórmula de Binet) donde  $\phi$  (razón áurea) es  $\frac{1+\sqrt{5}}{2} \approx 1,618$ . Pero este método no es muy preciso para términos grandes, debido a la imprecisión de los cálculos de coma flotante.

La sucesión de Fibonacci crece muy rápidamente y algunos problemas que la incorporan deben resolverse utilizando la biblioteca BigInteger de Java (ver la sección 5.3).

Esta sucesión tiene muchas propiedades interesantes. Una de ellas se recoge en el **teorema de Zeckendorf**: todo entero positivo puede expresarse de forma única como la suma de uno o más números de Fibonacci diferentes, de forma que dicha suma no incluya dos números de Fibonacci consecutivos. Para cualquier entero positivo, se puede satisfacer el teorema de Zeckendorf utilizando un algoritmo *voraz*: tomando el número de Fibonacci más grande posible en cada paso. Por ejemplo:  $100 = 89 + 8 + 3$ ;  $77 = 55 + 21 + 1$ ,  $18 = 13 + 5$ , etc.

Otra propiedad es el **periodo de Pisano**, donde los uno/dos/tres/cuatro últimos dígitos de un número de Fibonacci se repite con un periodo de 60/300/1500/15000, respectivamente.

#### Ejercicio 5.4.1.1

Intenta  $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$  con un  $n$  pequeño, y comprueba si esta fórmula de Binet obtiene, en realidad,  $fib(7) = 13$ ,  $fib(9) = 34$ ,  $fib(11) = 89$ . Ahora, escribe un programa sencillo que busque el primer valor de  $n$  en el que el resultado de  $fib(n)$  sea diferente del de esta fórmula. ¿Es ese  $n$  lo suficientemente grande como para ser utilizado en concursos de programación?

## 5.4.2 Coeficientes binomiales

Otro problema clásico de combinatoria aparece en la búsqueda de *coeficientes* de la expansión algebraica de potencias de un binomio<sup>9</sup>. Estos coeficientes son también el número de maneras en las que  $n$  elementos se pueden tomar en cantidad  $k$  cada vez, normalmente expresado como  $C(n, k)$  o  ${}^nC_k$ . Por ejemplo,  $(x+y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$ .  $\{1, 3, 3, 1\}$  son los coeficientes binomiales de  $n = 3$  con  $k = \{0, 1, 2, 3\}$ , respectivamente. O, dicho en otras palabras, el número de formas en las que  $n = 3$  elementos se pueden obtener con  $k = \{0, 1, 2, 3\}$  elementos cada vez son  $\{1, 3, 3, 1\}$ , respectivamente.

Podemos calcular cada valor de  $C(n, k)$  con esta fórmula:  $C(n, k) = \frac{n!}{(n-k)! \times k!}$ . Sin embargo, calcular  $C(n, k)$  puede ser un desafío cuando  $n$  y/o  $k$  son grandes. Hay varios trucos como: hacer  $k$  más pequeño (si  $k > n-k$ , entonces  $k = n-k$ ) porque  ${}^nC_k = {}^nC_{n-k}$ ; durante los cálculos intermedios, dividimos los números antes de multiplicarlos por el siguiente; o utilizamos BigInteger (como último recurso, ya que las operaciones con BigInteger son lentas).

Si tenemos que calcular *muchos, pero no todos*, de los valores de  $C(n, k)$  para distintos  $n$  y  $k$ , es mejor utilizar programación dinámica de arriba a abajo. Podemos escribir  $C(n, k)$  como se muestra a continuación, utilizando una tabla bidimensional, para evitar repetir cálculos:

- $C(n, 0) = C(n, n) = 1$  // casos base.
- $C(n, k) = C(n-1, k-1) + C(n-1, k)$  // tomar o ignorar un elemento,  $n > k > 0$ .

Sin embargo, si tenemos que calcular *todos* los valores de  $C(n, k)$  desde  $n = 0$  hasta un valor determinado de  $n$ , puede resultar beneficioso construir el *triángulo de Pascal*, un *array* triangular de coeficientes binomiales. Las entradas a izquierda y derecha de cada fila siempre son 1. Los valores interiores son la suma de los dos valores inmediatamente superiores, como se puede ver, a continuación, en la fila  $n = 4$ . Esta es, en esencia, la versión de abajo a arriba de la solución de programación dinámica anterior.

|       |                                                        |
|-------|--------------------------------------------------------|
| n = 0 | 1                                                      |
| n = 1 | 1    1                                                 |
| n = 2 | 1    2    1                                            |
| n = 3 | 1    3    3    1    <- como hemos visto<br>\ / \ / \ / |
| n = 4 | 1    4    6    4    1 ... y sigue                      |

### Ejercicio 5.4.2.1

Una  $k$  utilizada frecuentemente para  $C(n, k)$  es  $k = 2$ . Demuestra que  $C(n, 2) = O(n^2)$ .

## 5.4.3 Números de Catalan

Comencemos definiendo el número de Catalan  $n$ -ésimo, utilizando la notación  ${}^nC_k$  de coeficientes binomiales ya vista, como:  $Cat(n) = \frac{(2 \times n)C_n}{n+1}$ ;  $Cat(0) = 1$ . Ahora veremos su sentido.

<sup>9</sup>Un binomio es un caso especial de un polinomio, que solo tiene dos términos.

Si nos piden calcular los valores de  $Cat(n)$  para *varios* valores de  $n$ , puede ser mejor calcularlos utilizando programación dinámica de abajo a arriba. Una vez conocemos  $Cat(n)$ , podemos calcular  $Cat(n+1)$  mediante la manipulación de la fórmula que veremos a continuación:

$$Cat(n) = \frac{(2n)!}{n! \times n! \times (n+1)},$$

$$Cat(n+1) = \frac{(2 \times (n+1))!}{(n+1)! \times (n+1)! \times ((n+1)+1)} = \frac{(2n+2) \times (2n+1) \times (2n)!}{(n+1) \times n! \times (n+1) \times n! \times (n+2)} = \frac{(2n+2) \times (2n+1) \times \dots \times [(2n)!]}{(n+2) \times (n+1) \times \dots \times [n! \times n! \times (n+1)]}.$$

Por lo tanto,

$$Cat(n+1) = \frac{(2n+2) \times (2n+1)}{(n+2) \times (n+1)} \times Cat(n).$$

Alternativamente, podemos establecer  $m = n+1$  de forma que tengamos:

$$Cat(m) = \frac{2m \times (2m-1)}{(m+1) \times m} \times Cat(m-1).$$

Los números de Catalan se encuentran (sorprendentemente) en varios problemas de combinatoria. Enumeramos ahora algunos de los más interesantes (hay varios más, ver el **ejercicio 5.4.4.8\***). Todos los ejemplos que aparecen a continuación utilizan  $n = 3$  y  $Cat(3) = \frac{(2 \times 3)C_3}{3+1} = \frac{^6C_3}{4} = \frac{20}{4} = 5$ .

1.  $Cat(n)$  cuenta el número de árboles binarios distintos con  $n$  vértices. Para  $n = 3$ :



2.  $Cat(n)$  cuenta el número de expresiones que contienen  $n$  pares de paréntesis correctamente emparejados, por ejemplo, para  $n = 3$ , tenemos:  $(\cdot)(\cdot)$ ,  $(\cdot)(\cdot)$ ,  $(\cdot)(\cdot)$ ,  $((\cdot))$  y  $(\cdot)(\cdot)$ . Para conocer más detalles sobre este problema, ver la sección 9.4.
3.  $Cat(n)$  cuenta el número de formas diferentes en que se pueden colocar  $n+1$  factores entre paréntesis, por ejemplo, para  $n = 3$  y  $3+1 = 4$  factores: {a, b, c, d}, tenemos:  $(ab)(cd)$ ,  $a(b(cd))$ ,  $((ab)c)d$ ,  $(a(bc))d$  y  $a((bc)d)$ .
4.  $Cat(n)$  cuenta el número de formas en que se puede triangular un polígono convexo (ver la sección 7.3) de  $n+2$  lados. Ver la parte izquierda de la figura 5.1.
5.  $Cat(n)$  cuenta el número de caminos monótonos que hay a lo largo de las aristas de una rejilla  $n \times n$ , que no pasan sobre la diagonal. Un camino monótono es aquel que comienza en la esquina inferior izquierda, termina en la esquina superior derecha y está formado completamente por aristas que apuntan a la derecha o hacia arriba. Ver la parte derecha de la figura 5.1 y la sección 4.7.1.

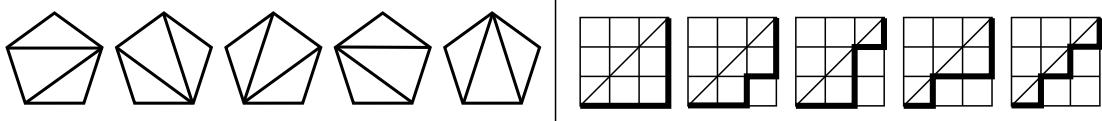


Figura 5.1: Izquierda: triangulación de un polígono convexo, derecha: caminos monótonos

#### 5.4.4 Notas sobre combinatoria en concursos de programación

Hay muchos otros problemas de combinatoria que pueden aparecer en concursos de programación, pero no serán tan frecuentes como la sucesión de Fibonacci, los coeficientes binomiales o los números de Catalan. Algunos de los más interesantes aparecen en la sección 9.8.

En los concursos de programación *en línea*, donde los concursantes tienen acceso a internet, hay otro truco que puede resultar útil. En primer lugar, generamos una salida de pequeñas instancias y, después, la buscamos en la OEIS (The On-Line Encyclopedia of Integer Sequences), que se encuentra en <http://oeis.org/>. Si tienes suerte, OEIS podrá indicarte el nombre de la secuencia y/o fórmula general necesaria para resolver las instancias más grandes.

Quedan todavía muchos sistemas de conteo y fórmulas, demasiados para incluirlos en este libro. Finalizamos la sección con algunos ejercicios escritos para verificar o mejorar tus conocimientos de combinatoria.

#### Ejercicio 5.4.4.1

Cuenta el número de posibles resultados si lanzas dos dados de 6 caras y dos monedas de 2 caras.

#### Ejercicio 5.4.4.2

¿De cuántas maneras se puede formar un número de tres dígitos de entre {0, 1, 2, ..., 9}, usando cada dígito una sola vez? No se permite utilizar el 0 como primer dígito.

#### Ejercicio 5.4.4.3

Supongamos que tienes la palabra de 6 letras ‘FACTOR’. Si tomamos tres letras de esta palabra, podemos formar otras válidas en inglés, como ‘ACT’, ‘CAT’, ‘ROT’, etc. ¿Qué número máximo de palabras de tres letras diferentes podemos formar con las letras de ‘FACTOR’? No tienes que preocuparte de si las palabras resultantes son correctas en inglés o no.

### Ejercicio 5.4.4.4

Dada la palabra de 5 letras ‘BOBBY’, reordena las letras para obtener otra, por ejemplo, ‘BBBOY’, ‘YO BBB’, etc. ¿Cuántas permutaciones *diferentes* son posibles?

### Ejercicio 5.4.4.5

Resuelve el problema UVa 11401 - Triangle Counting. Este problema tiene un enunciado corto: “Dadas  $n$  varillas de longitudes 1, 2, ...,  $n$ , elegir tres de ellas para formar un triángulo. ¿Cuántos triángulos distintos se pueden construir (considera la desigualdad de los triángulos, ver la sección 7.2)? ( $3 \leq n \leq 1M$ )”. Ten en cuenta que se considerará que dos triángulos son diferentes si tienen, al menos, un par de lados de distinta longitud. Si tienes suerte, podrás identificar el patrón en unos pocos minutos. Si no, este problema podría quedar sin resolver al finalizar el concurso, lo que no es bueno para tu equipo.

### Ejercicio 5.4.4.6\*

Estudia los siguientes términos: lema de Burnside, números de Stirling.

### Ejercicio 5.4.4.7\*

¿Cuál de los siguientes es más difícil de factorizar (sección 5.5.4), asumiendo que  $n$  es un entero grande arbitrario:  $\text{fib}(n)$ ,  $C(n, k)$  (asume que  $k = n/2$ ) o  $\text{Cat}(n)$ ? ¿Por qué?

### Ejercicio 5.4.4.8\*

Los números de Catalan  $\text{Cat}(n)$  aparecen en otros problemas interesantes que no hemos incluido en esta sección. Investiga sobre ello.

## Ejercicios de programación

Otros ejercicios de programación relacionados con la combinatoria:

### Sucesión de Fibonacci

1. UVa 00495 - Fibonacci Freeze
2. UVa 00580 - Critical Mass
3. **UVa 00763 - Fibinary Numbers \***  
(usar DP en  $O(n)$ ; BigInteger de Java)  
(relacionado con la sucesión de Tribonacci)
4. UVa 00900 - Brick Wall Patterns  
(representación Zeckendorf; voraz; usar BigInteger de Java)
5. UVa 00948 - Fibonaccimal Base  
(combinatoria; el patrón  $\approx$  Fibonacci)
6. UVa 01258 - Nowhere Money  
(representación de Zeckendorf; voraz)
7. UVa 10183 - How many Fibs?  
(LA 4721 - Phuket09; variante de Fibonacci; Zeckendorf; voraz)
8. **UVa 10334 - Ray Through Glasses \***  
(guardar el número de términos de Fibonacci al generarlos; BigInteger)
9. UVa 10450 - World Cup Noise  
(combinatoria; el patrón  $\approx$  Fibonacci)
10. UVa 10497 - Sweet Child Make Trouble  
(combinatoria; el patrón  $\approx$  Fibonacci)
11. UVa 10579 - Fibonacci Numbers  
(muy fácil con BigInteger de Java)
12. **UVa 10689 - Yet Another Number ... \***  
(fácil; periodo de Pisano)
13. UVa 10862 - Connect the Cable Wires  
(el patrón acaba  $\approx$  Fibonacci)
14. UVa 11000 - Bee  
(combinatoria; el patrón es similar a Fibonacci)
15. *UVa 11089 - Fi-binary Number*  
(la lista de los números Fi-binarios sigue el teorema de Zeckendorf)
16. UVa 11161 - Help My Brother (II)  
(Fibonacci y mediana)
17. UVa 11780 - Miles 2 Km  
(el fondo del problema es la sucesión de Fibonacci)

### Coeficientes binomiales

1. UVa 00326 - Extrapolation using a ...  
(tabla de diferencias)
2. UVa 00369 - Combinations  
(cuidado con el desbordamiento)
3. UVa 00485 - Pascal Triangle of Death  
(coeficientes binomiales y BigInteger)
4. UVa 00530 - Binomial Showdown  
(trabajar con double; optimizar cálculo)
5. *UVa 00911 - Multinomial Coefficients*  
(existe una fórmula:  $res = n! / (z_1! \times z_2! \times z_3! \times \dots \times z_k!)$ )
6. UVa 10105 - Polynomial Coefficients  
( $n! / (n_1! \times n_2! \times \dots \times n_k!)$ ; sin embargo, la deducción es compleja)
7. **UVa 10219 - Find the Ways \***  
(contar la longitud  $"C_k"$ ; BigInteger)
8. UVa 10375 - Choose and Divide  
(la tarea principal es evitar desbordamiento)
9. *UVa 10532 - Combination, Once Again*  
(coeficiente binomial modificado)
10. **UVa 10541 - Stripe \***  
(un buen problema de combinatoria)
11. **UVa 11955 - Binomial Theorem \***  
(aplicación pura; DP)

### Números de Catalan

1. **UVa 00991 - Safe Salutations \***  
(números de Catalan)
2. **UVa 10007 - Count the Trees \***  
(la respuesta es  $Cat(n) \times n!$ ; BigInteger)
3. UVa 10223 - How Many Nodes?  
(se puede calcular previamente la respuesta, ya que solo hay 19 números de Catalan  $< 2^{32}-1$ )
4. UVa 10303 - How Many Trees  
(generar  $Cat(n)$  como se muestra en esta sección; usar BigInteger)
5. **UVa 10312 - Expression Bracketing \***  
(número de paréntesis binarios =  $Cat(n)$ ; número de paréntesis = números Súper-Catalan)
6. *UVa 10643 - Facing Problems With ...*  
( $Cat(n)$  es parte de un problema mayor)

## Otros, más fáciles

1. UVa 11115 - Uncle Jack
2. UVa 11310 - Delivery Debacle \*
3. UVa 11401 - Triangle Counting \*
4. UVa 11480 - Jimmy's Balls
5. UVa 11597 - Spanning Subtree \*
6. UVa 11609 - Teams
7. UVa 12463 - Little Nephew

( $N^D$ ; usa BigInteger de Java)  
(necesita DP: digamos que  $dp[i]$  es el número de formas en las que las tartas se puede empaquetar para una caja  $2 \times i$ )  
(encuentra el patrón)  
(prueba todos los  $r$ ; pero existe una fórmula más sencilla)  
(utiliza conocimientos de teoría de grafos; la respuesta es trivial)  
( $N \times 2^{N-1}$ ; usa BigInteger de Java para la parte de modPow)  
(duplica los calcetines y los zapatos para simplificar el problema)

## Otros, difíciles

1. UVa 01224 - Tile Code
2. UVa 10079 - Pizza Cutting
3. UVa 10359 - Tiling
4. UVa 10733 - The Colored Cubes
5. UVa 10784 - Diagonal \*
6. UVa 10790 - How Many Points of ...
7. UVa 10918 - Tri Tiling
8. UVa 11069 - A Graph Problem \*
9. UVa 11204 - Musical Instruments
10. UVa 11270 - Tiling Dominoes
11. UVa 11538 - Chess Queen \*
12. UVa 11554 - Hapless Hedonism
13. UVa 12022 - Ordering T-shirts

(LA 3904 - Seoul07; deduce la fórmula observando instancias pequeñas)  
(deduce la fórmula de una línea)  
(deduce la fórmula; usa BigInteger de Java)  
(lema de Burnside)  
(el número de diagonales en  $n$ -gon =  $n * (n - 3) / 2$ ; úsallo para deducir la fórmula)  
(utiliza una fórmula de progresión aritmética)  
(hay dos recurrencias relacionadas)  
(usa programación dinámica)  
(solo importa la primera elección)  
(secuencia A004003 en OEIS)  
(contar filas, columnas y diagonales)  
(similar a UVa 11401)  
(el número de formas en que  $n$  competidores se pueden clasificar, permitiendo empates; ver <http://oeis.org/A000670>)

## Perfiles de los inventores de algoritmos

**Leonardo Fibonacci** (también conocido como **Leonardo Pisano**) (1170-1250) fue un matemático italiano. Publicó un libro titulado ‘Liber Abaci’ (Libro del ábaco/cálculo), en el que trató un problema sobre el crecimiento de una población de *conejos*, basado en hechos idealizados. La solución fue una secuencia de números, conocida hoy como la sucesión de Fibonacci.

**Edouard Zeckendorf** (1901-1983) fue un matemático belga. Es principalmente conocido por su trabajo con la sucesión de Fibonacci y, en particular, por demostrar el teorema de Zeckendorf.

**Jacques Philippe Marie Binet** (1786-1856) fue un matemático francés. Realizó aportaciones significativas a la teoría de números. La fórmula de Binet, que expresa números de Fibonacci de forma cerrada, se llama así en su honor, aunque el mismo resultado ya se conocía anteriormente.

**Blaise Pascal** (1623-1662) fue un matemático francés. Uno de sus famosos inventos, tratado en este libro, es el triángulo de Pascal de coeficientes binomiales.

**Eugène Charles Catalan** (1814-1894) fue un matemático francobelga. Introdujo los números de Catalan para resolver un problema de combinatoria.

**Eratóstenes de Cirene** ( $\approx$  300-200 a.C.) fue un matemático griego. Inventó la geografía, realizó mediciones de la circunferencia de la Tierra y diseñó un algoritmo sencillo, tratado en este libro, para generar números primos.

**Leonhard Euler** (1707-1783) fue un matemático suizo. Sus inventos mencionados en este libro son la función indicatriz de Euler ( $F_i$ ) y el camino euleriano (grafos).

**Christian Goldbach** (1690-1764) fue un matemático alemán. Es recordado por la conjetura de Goldbach, que discutió extensamente con Leonhard Euler.

**Diofanto de Alejandría** ( $\approx$  200-300 d.C.) fue un matemático grecoalejandrino. Realizó extensos estudios sobre álgebra. Uno de sus trabajos son las ecuaciones diofánticas lineales.

## 5.5 Teoría de números

Dominar todos los aspectos posibles en el campo de la *teoría de números* es importante, ya que algunos problemas matemáticos resultan fáciles (o más fáciles) si conoces la teoría que reside detrás de los problemas. En caso contrario, o la fuerza bruta llevará a una respuesta TLE o, sencillamente, no podrás trabajar con la entrada dada, pues será demasiado grande sin un procesamiento previo.

### 5.5.1 Números primos

Un número natural mayor o igual que 2:  $\{2, 3, 4, 5, 6, 7, \dots\}$ , se considera **primo** si solo es divisible por 1 y por sí mismo. El primer número primo, y el único que es par, es el 2. Los siguientes primos son: 3, 5, 7, 11, 13, 17, 19, 23, 29, ..., y hay un número infinito de números primos más (la prueba en [57]). Existen 25 primos en el espacio  $[0..100]$ , 168 en  $[0..1000]$ , 1000 en  $[0..7919]$ , 1229 en  $[0..10000]$ , etc. Algunos primos más grandes son<sup>10</sup> 104729, 1299709, 15485863, 179424673, 2147483647, 32416190071, 112272535095293, 48112959837082048697, etc. Los números primos conforman una materia importante en la teoría de números y son el origen de muchos problemas de programación<sup>11</sup>. En esta sección, trataremos algoritmos relacionados con números primos.

#### Función optimizada de prueba de primalidad

El primer algoritmo que presentamos en esta sección, se utiliza para comprobar si un número natural  $N$  dado es primo, por ejemplo `bool isPrime(N)`. La versión más ingenua consistiría en verificarlo mediante la propia definición de un número primo, es decir, comprobar si  $N$  es divisible por un divisor  $\in [2..N-1]$ . Es una opción válida, pero su coste, por el número de divisiones, es de  $O(N)$ . No es el mejor camino, pero podemos optimizarlo de varias formas.

La primera mejora importante es comprobar si  $N$  es divisible por un divisor  $\in [2..\sqrt{N}]$ , esto es, nos detenemos cuando el divisor es mayor que  $\sqrt{N}$ . El motivo es claro: si  $N$  es divisible por  $d$ , entonces  $N = d \times \frac{N}{d}$ . Si  $\frac{N}{d}$  es menor que  $d$ , entonces  $\frac{N}{d}$ , o un factor primo de  $\frac{N}{d}$ , habrían dividido  $N$  anteriormente. Por lo tanto,  $d$  y  $\frac{N}{d}$  no pueden ser, *los dos*, mayores que  $\sqrt{N}$ . Esta

<sup>10</sup>Puede resultar útil tener una lista de números primos grandes para realizar pruebas, ya que son los que resultan más difíciles para algoritmos como la comprobación o la factorización de números primos.

<sup>11</sup>En la vida real, los primos grandes se utilizan en el campo de la criptografía, porque es muy difícil factorizar un número  $xy$  en  $x$  y  $y$  cuando ambos son **primos relativos** (también conocidos como **coprimos**).

mejora nos lleva a  $O(\sqrt{N})$ , lo que ya resulta mucho más rápido que la versión anterior, pero todavía podemos doblar la velocidad.

La segunda mejora consiste en comprobar si  $N$  es divisible por un divisor  $\in [3, 5, 7, \dots, \sqrt{N}]$ , es decir, comprobamos únicamente los números impares hasta  $\sqrt{N}$ . Esto es debido a que hay un único número primo par, el 2, que podemos verificar por separado. Esto es  $O(\sqrt{N}/2)$ , que también es  $O(\sqrt{N})$ .

La tercera mejora<sup>12</sup>, que ya resulta suficientemente buena<sup>13</sup> para los problemas de los concursos es comprobar si  $N$  es divisible por divisores primos  $\leq \sqrt{N}$ . Esto responde a que, si un número primo  $X$  no es divisor de  $N$ , no tiene ningún sentido comprobar los múltiplos de  $X$  como posibles divisores de  $N$ . Resulta más rápido que  $O(\sqrt{N})$ , aproximadamente  $O(\#\text{primos} \leq \sqrt{N})$ . Por ejemplo, hay 500 números impares en  $[1.. \sqrt{10^6}]$ , pero sólo tenemos 168 números primos en el mismo rango. El teorema de números primos de [57] dice que el número de primos menor o igual a  $N$ , expresado como  $\pi(M)$ , está limitado por  $O(M/(\ln(M) - 1))$ . Por ello, la complejidad de esta función de prueba de primalidad se encuentra, aproximadamente, en  $O(\sqrt{N}/\ln(\sqrt{N}))$ . El código se incluye más adelante.

### Criba de Eratóstenes: generación de listas de números primos

Si queremos generar una lista de números primos en el rango  $[0..N]$ , existe un algoritmo mejor que comprobar si cada número del rango es primo o no. El algoritmo se llama ‘criba de Eratóstenes’, inventado por Eratóstenes de Alejandría.

Primero, este algoritmo de criba establece todos los números del rango como ‘posibles primos’, pero indica que el 0 y el 1 no lo son. Después, toma el 2 como primo y elimina todos los múltiplos<sup>14</sup> de 2, empezando desde  $2 \times 2 = 4, 6, 8, 10, \dots$ , hasta que el múltiplo sea mayor que  $N$ . A continuación, toma el siguiente número no eliminado, el 3, como primo y elimina todos los múltiplos del 3 comenzando con  $3 \times 3 = 9, 12, 15, \dots$ . Después es el turno del 5 y elimina todos sus múltiplos empezando por  $5 \times 5 = 25, 30, 35, \dots$ . Y así sucesivamente. Finalizado el proceso, cualquier número no eliminado en el rango  $[0..N]$  es primo. El algoritmo realiza, aproximadamente,  $(N \times (1/2 + 1/3 + 1/5 + 1/7 + \dots + 1/\text{último primo del rango} \leq N))$  operaciones. Utilizando la ‘suma de los reciprocos de los primos hasta  $n$ ’, terminamos con una complejidad de alrededor de  $O(N \log \log N)$ .

Como generar una lista de primos  $\leq 10000$ , utilizando la criba, es rápido (el código que incluimos a continuación llega hasta  $10^7$  en condiciones de concurso), optaremos por ella para los números pequeños y reservaremos la función de prueba de primalidad optimizada para los más grandes. El código es el siguiente:

```

1 #include <bitset> // STL compacto para la criba, mejor que vector<bool>
2 ll _sieve_size; // ll se define como: typedef long long ll;
3 bitset<10000010> bs; // 10^7 debería ser suficiente en la mayoría de casos
4 vi primes; // lista compacta de primos en forma de vector<int>
5

```

<sup>12</sup>Esta es recursiva, comprobar que un número es primo utilizando otro primo más pequeño, pero la razón resultará obvia después de leer la siguiente sección.

<sup>13</sup>Ver también la comprobación probabilística de primalidad de Miller-Rabin, utilizando la clase BigInteger de Java, en la sección 5.3.2.

<sup>14</sup>La implementación más común comienza desde  $2 \times i$ , en vez de  $i \times i$ , pero la diferencia no es muy apreciable.

```

6 void sieve(ll upperbound) { // crear lista de primos en [0..upperbound]
7 _sieve_size = upperbound+1; // añadir 1 para incluir upperbound
8 bs.set(); // establecer todos los bits a 1
9 bs[0] = bs[1] = 0; // excepto los índices 0 y 1
10 for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
11 // tachar los múltiplos de i a partir de i*i
12 for (ll j = i*i; j <= _sieve_size; j += i) bs[j] = 0;
13 primes.push_back((int)i); // añadir este primo a la lista de primos
14 } } // llamar a este método desde main
15
16 bool isPrime(ll N) { // una buena prueba de primalidad determinista
17 if (N <= _sieve_size) return bs[N]; // O(1) para primos pequeños
18 for (int i = 0; i < (int)primes.size(); i++)
19 if (N % primes[i] == 0) return false;
20 return true; // tarda más si N es un primo grande
21 } } // nota: solo funciona para N <= (último primo en vi primes)^2
22
23 // dentro de int main()
24 sieve(10000000); // puede ir hasta 10^7 (tarda unos segundos)
25 printf("%d\n", isPrime(2147483647)); // primo de 10 dígitos
26 printf("%d\n", isPrime(136117223861LL)); // no es primo, 104729*1299709

```



ch5\_06\_primes.cpp



ch5\_06\_primes.java

## 5.5.2 Máximo común divisor y mínimo común múltiplo

El máximo común divisor (GCD) de dos enteros  $a$  y  $b$ , expresado como  $\gcd(a, b)$ , es el entero positivo  $d$  más grande que  $d \mid a$  y  $d \mid b$ , donde  $x \mid y$  significa que  $x$  divide a  $y$ . Ejemplos de GCD:  $\gcd(4, 8) = 4$ ,  $\gcd(6, 9) = 3$ ,  $\gcd(20, 12) = 4$ . Un uso práctico del GCD se encuentra en la simplificación de fracciones (ver UVa 10814, sección 5.3.2), como en  $\frac{6}{9} = \frac{6/\gcd(6,9)}{9/\gcd(6,9)} = \frac{6/3}{9/3} = \frac{2}{3}$ .

Encontrar el GCD de dos enteros es una tarea sencilla mediante un algoritmo *euclídeo* eficiente de divide y vencerás [57, 7], que se puede implementar en una sola línea (ver a continuación). Así, la búsqueda del GCD de dos enteros no suele ser el enigma principal de los problemas de concursos relacionados con matemáticas, sino parte de una solución mayor.

El GCD está íntimamente relacionado con el mínimo común múltiplo (LCM). El LCM de dos enteros  $(a, b)$ , expresado como  $\text{lcm}(a, b)$ , se define como el entero positivo más pequeño  $l$  de forma que  $a \mid l$  y  $b \mid l$ . Ejemplos de LCM:  $\text{lcm}(4, 8) = 8$ ,  $\text{lcm}(6, 9) = 18$ ,  $\text{lcm}(20, 12) = 60$ . Se ha demostrado (ver [57]) que:  $\text{lcm}(a, b) = \frac{a \times b}{\gcd(a, b)}$ . También se puede implementar en una línea de código (ver a continuación).

Ambos algoritmos se ejecutan en  $O(\log_{10} n)$ , donde  $n = \max(a, b)$ .

```

1 int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
2 int lcm(int a, int b) { return a * (b / gcd(a, b)); }

```

Se puede encontrar el GCD de más de dos números, realizando varias llamadas a gcd para dos números, por ejemplo,  $\text{gcd}(a, b, c) = \text{gcd}(a, \text{gcd}(b, c))$ . La estrategia para la búsqueda del LCM de más de dos números es similar.

### Ejercicio 5.5.2.1

La fórmula para el LCM es  $\text{lcm}(a, b) = \frac{a \times b}{\text{gcd}(a, b)}$  pero, ¿por qué utilizamos  $a \times \frac{b}{\text{gcd}(a, b)}$  en su lugar? Consejo: prueba con  $a = 10^9$  y  $b = 8$ , utilizando enteros con signo de 32 bits.

### 5.5.3 Factorial

El factorial de  $n$ , también expresado como  $n!$  o  $\text{fac}(n)$ , se define como 1 si  $n = 0$  y como  $n \times \text{fac}(n-1)$  si  $n > 0$ . Sin embargo, normalmente resulta más cómodo trabajar con la versión iterativa, es decir,  $\text{fac}(n) = 2 \times 3 \times 4 \times 5 \times \dots \times (n-1) \times n$  (un bucle desde 2 hasta  $n$ , evitando el 1). El valor de  $\text{fac}(n)$  crece muy rápido. Solo podremos utilizar long long en C/C++ (long en Java) hasta  $\text{fac}(20)$ . Después, tendremos que utilizar la biblioteca BigInteger de Java para obtener un cálculo preciso, aunque lento (ver la sección 5.3), trabajar con los factores primos del factorial (ver la sección 5.5.5) u obtener los resultados intermedio y final módulo un número más pequeño (ver la sección 5.5.8).

### 5.5.4 Búsqueda de factores primos con división por tentativa optimizada

En la teoría de números, sabemos que un número primo  $N$  solo tiene el 1 y a sí mismo como factores, pero un número **compuesto**  $N$ , es decir, no primo, se puede escribir de forma única como la multiplicación de sus factores primos. Esto es, los números primos son ladrillos de construcción de enteros mediante la multiplicación (el teorema fundamental de la aritmética). Por ejemplo,  $N = 1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$  (la última expresión se denomina **factorización de potencias de primos**).

Un algoritmo ingenuo genera una lista de primos (por ejemplo, mediante criba) y comprueba cuáles de ellos pueden dividir el entero  $N$ , sin alterar  $N$ . Pero esto es mejorable.

Otro algoritmo mejor utiliza el espíritu del divide y vencerás. Un entero  $N$  se puede expresar como  $N = PF \times N'$ , donde  $PF$  es un factor primo y  $N'$  es otro número, concretamente  $\frac{N}{PF}$ , es decir, podemos reducir el tamaño de  $N$ , extrayendo su factor primo  $PF$ . Si seguimos haciendo esto, llegará un momento en que  $N' = 1$ . Para acelerar todavía más el proceso, utilizamos la propiedad de divisibilidad, que dice que no existe un divisor mayor que  $\sqrt{N}$ , así que solo repetiremos el proceso de búsqueda de factores primos hasta que  $PF > \sqrt{N}$ . Detenerse en  $\sqrt{N}$  provoca un caso especial: si (el actual)  $PF^2 > N$  y  $N$  todavía no es 1, entonces  $N$  es el último factor primo. El siguiente código toma un entero  $N$  y devuelve la lista de sus factores primos.

En el peor de los casos, cuando  $N$  ya es primo, el algoritmo de factorización en números primos mediante división por tentativa, necesita probar todos los primos más pequeños hasta  $\sqrt{N}$ , expresado matemáticamente como  $O(\pi(\sqrt{N})) = O(\sqrt{N}/\ln\sqrt{N})$ . En el siguiente código, se puede ver el ejemplo de factorización del número compuesto grande 136117223861, en dos factores primos grandes  $104729 \times 1299709$ . Sin embargo, si tenemos números compuestos con muchos factores primos pequeños, este algoritmo es razonablemente rápido. Ver 142391208960, que es  $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$ .

```

1 vi primeFactors(ll N) { // condición previa, N >= 1
2 vi factors; // recuerda: vi es vector<int>, ll es long long
3 ll PF_idx = 0, PF = primes[PF_idx]; // primes ha sido poblado por criba
4 while (PF*PF <= N) { // detenerse en sqrt(N); N puede ser menor
5 while (N%PF == 0) { N /= PF; factors.push_back(PF); } // eliminar PF
6 PF = primes[++PF_idx]; // considerar solo los primos
7 }
8 if (N != 1) factors.push_back(N); // caso especial si N es primo
9 return factors; // si N no cabe en un entero de 32 bits y es primo
10 } // entonces habrá que cambiar 'factors' a vector<ll>
11
12 // dentro de int main(), asumiendo que ya hemos llamado a sieve(1000000)
13 vi r = primeFactors(2147483647); // muy lento, 2147483647 es primo
14 for (vi::iterator i = r.begin(); i != r.end(); i++) printf("> %d\n", *i);
15
16 r = primeFactors(136117223861LL); // lento, 104729*1299709
17 for (vi::iterator i = r.begin(); i != r.end(); i++) printf("# %d\n", *i);
18
19 r = primeFactors(142391208960LL); // rápido, 2^10*3^4*5*7^4*11*13
20 for (vi::iterator i = r.begin(); i != r.end(); i++) printf("! %d\n", *i);

```

### Ejercicio 5.5.4.1

Examina el código anterior. ¿Cuáles son los valores de N que pueden hacer fallar este código? Puedes asumir que vi primes contiene una lista de números primos, siendo el mayor 9999991 (un poco menos de 10 millones).

### Ejercicio 5.5.4.2

John Pollard inventó un algoritmo mejor para factorizar enteros. Estudia e implementa el algoritmo rho de Pollard (el original y la mejora de Richard P. Brent) [52, 3].

## 5.5.5 Trabajo con factores primos

Además de utilizar la técnica de BigInteger de Java (sección 5.3), que resulta ‘lenta’, podemos trabajar con los *cálculos intermedios* de enteros largos, *con precisión*, si utilizamos los *factores primos* de los enteros, en vez de los propios enteros. Por lo tanto, para algunos problemas no triviales de teoría de números, tendremos que trabajar con los factores primos de los enteros de la entrada, aunque el problema principal no trate sobre números primos. Después de todo, los factores primos son los ladrillos que construyen los enteros. Veamos un caso de estudio.

El problema UVa 10139 - Factovisors se puede resumir así: “¿Es  $m$  un divisor de  $n!$ , es decir,  $\text{fac}(n)$ ? ( $0 \leq n, m \leq 2^{31}-1$ )”. En la sección 5.5.3, hemos visto que con los *tipos de datos integrados*, el factorial más largo que podemos calcular con precisión es  $20!$ . En la sección 5.3, mostramos que podemos calcular enteros grandes con la técnica BigInteger de Java. Sin embargo, es *muy lento* calcular con precisión el valor exacto de  $n!$ , si  $n$  es grande. La solución a este problema consiste en trabajar con los factores primos tanto de  $n!$  como de  $m$ . Factorizamos  $m$  y comprobamos si tiene ‘soporte’ en  $n!$ . Por ejemplo, cuando  $n = 6$ , tenemos  $6!$  expresado como factorización de potencias de primos:  $6! = 2 \times 3 \times 4 \times 5 \times 6 = 2 \times 3 \times (2^2) \times 5 \times (2 \times 3) = 2^4 \times 3^2 \times 5$ . Para  $6!$ ,  $m_1 = 9 = 3^2$  tiene soporte, pues vemos que  $3^2$  forma parte de  $6!$ , por lo que  $m_1 = 9$  divide a  $6!$ . Sin embargo,  $m_2 = 27 = 3^3$  no tiene soporte, puesto de relevancia por el hecho de que la potencia más grande de 3 en  $6!$  es solo  $3^2$ , por lo que  $m_2 = 27$  no divide a  $6!$ .

### Ejercicio 5.5.5.1

Determina el GCD y el LCM de  $(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2)$ .

## 5.5.6 Funciones que implican factores primos

Hay otras funciones de la teoría de números, bien conocidas, que implican factores primos y que veremos a continuación. Todas las variantes tienen una complejidad de tiempo similar, con la factorización en primos mediante división por tentativa que hemos visto antes. El lector interesado puede ilustrarse en el capítulo 7 (“Funciones multiplicativas”) de [57].

1. numPF( $N$ ): contar el número de *factores primos* de  $N$ .

Un sencillo cambio sobre el algoritmo de división por tentativa para encontrar factores primos que hemos visto antes.

```

1 11 numPF(11 N) {
2 11 PF_idx = 0, PF = primes[PF_idx], ans = 0;
3 while (PF*PF <= N) {
4 while (N%PF == 0) { N /= PF; ans++; }
5 PF = primes[++PF_idx];
6 }
7 if (N != 1) ans++;
8 return ans;
9 }
```

2. numDiffPF( $N$ ): contar el número de factores primos *diferentes* de  $N$ .
3. sumPF( $N$ ): *sumar* los factores primos de  $N$ .
4. numDiv( $N$ ): contar el número de *divisores* de  $N$ .

El divisor de un entero  $N$ , se define como un entero que divide a  $N$  sin dejar resto. Si un número  $N = a^i \times b^j \times \dots \times c^k$ , entonces  $N$  tiene  $(i+1) \times (j+1) \times \dots \times (k+1)$  divisores. Por ejemplo:  $N = 60 = 2^2 \times 3^1 \times 5^1$  tiene  $(2+1) \times (1+1) \times (1+1) = 3 \times 2 \times 2 = 12$

divisores. Los 12 divisores son:  $\{1, \underline{2}, \underline{3}, 4, \underline{5}, 6, 10, 12, 15, 20, 30, 60\}$ . Hemos destacado los factores primos de 60. Hay que notar que  $N$  tiene más divisores que factores primos.

```

1 ll numDiv(ll N) {
2 ll PF_idx = 0, PF = primes[PF_idx], ans = 1; // empezar en ans = 1
3 while (PF*PF <= N) {
4 ll power = 0; // contar la potencia
5 while (N%PF == 0) { N /= PF; power++; }
6 ans *= (power+1); // según la fórmula
7 PF = primes[++PF_idx];
8 }
9 if (N != 1) ans *= 2; // (último factor tiene pow = 1, le añadimos 1)
10 return ans;
11 }
```

##### 5. sumDiv(N): suma de los divisores de N.

En el ejemplo anterior,  $N = 60$  tenía 12 divisores. La suma de todos ellos es 168. Esto también se puede calcular mediante factores primos. Si un número  $N = a^i \times b^j \times \dots \times c^k$ , entonces la suma de los divisores de  $N$  es  $\frac{a^{i+1}-1}{a-1} \times \frac{b^{j+1}-1}{b-1} \times \dots \times \frac{c^{k+1}-1}{c-1}$ . Vamos a probar.  $N = 60 = 2^2 \times 3^1 \times 5^1$ ,  $\text{sumDiv}(60) = \frac{2^{2+1}-1}{2-1} \times \frac{3^{1+1}-1}{3-1} \times \frac{5^{1+1}-1}{5-1} = \frac{7 \times 8 \times 24}{1 \times 2 \times 4} = 168$ .

```

1 ll sumDiv(ll N) {
2 ll PF_idx = 0, PF = primes[PF_idx], ans = 1; // empezar en ans = 1
3 while (PF*PF <= N) {
4 ll power = 0;
5 while (N%PF == 0) { N /= PF; power++; }
6 ans *= ((ll)pow((double)PF, power+1.0) - 1) / (PF-1);
7 PF = primes[++PF_idx];
8 }
9 if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N-1); // último
10 return ans;
11 }
```

##### 6. EulerPhi(N): contar el número de enteros positivos $< N$ que son primos relativos a $N$ .

Recordemos: se dice que dos enteros  $a$  y  $b$  son primos relativos (o coprimos) si  $\gcd(a, b) = 1$ , por ejemplo, 25 y 42. Un algoritmo ingenuo para contar el número de enteros positivos  $< N$ , que sean primos relativos a  $N$ , comienza con `contador = 0`, itera en  $i \in [1..N-1]$ , e incrementa el `contador` si  $\gcd(i, N) = 1$ . Esto resulta lento para un  $N$  grande.

Un algoritmo mejor es la función Fi (indicatriz) de Euler  $\varphi(N) = N \times \prod_{PF} (1 - \frac{1}{PF})$ , donde  $PF$  es un factor primo de  $N$ .

Por ejemplo,  $N = 36 = 2^2 \times 3^2$ .  $\varphi(36) = 36 \times (1 - \frac{1}{2}) \times (1 - \frac{1}{3}) = 12$ . Esos 12 enteros positivos, que son primos relativos a 36, son  $\{1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35\}$ .

```

1 ll EulerPhi(ll N) {
2 ll PF_idx = 0, PF = primes[PF_idx], ans = N; // empezar en ans = N
3 while (PF*PF <= N) {
```

```

4 if (N%PF == 0) ans -= ans / PF; // contar solo el factor único
5 while (N%PF == 0) N /= PF;
6 PF = primes[++PF_idx];
7 }
8 if (N != 1) ans -= ans / N; // último factor
9 return ans;
10 }

```

### Ejercicio 5.5.6.1

Implementar `numDiffPF(N)` y `sumPF(N)`. Consejo: ambos son similares a `numPF(N)`.

## 5.5.7 Criba modificada

Si hay que determinar el número de factores primos para *muchos* (o un *rango* de) enteros, hay una solución mejor que llamar a `numDiffPF(N)`, como hemos visto en la sección 5.5.6, *muchas veces*. La mejor solución es el algoritmo de criba modificada. En vez de buscar los factores primos y, después, calcular los valores necesarios, comenzamos con los números primos y modificamos los valores de sus múltiplos. A continuación incluimos el código, corto, de la criba modificada:

```

1 memset(numDiffPF, 0, sizeof numDiffPF);
2 for (int i = 2; i < MAX_N; i++)
3 if (numDiffPF[i] == 0) // i es un número primo
4 for (int j = i; j < MAX_N; j += i)
5 numDiffPF[j]++; // aumentar valores de los múltiplos de i

```

Este algoritmo de criba modificada es preferible a llamadas individuales a `numDiffPF(N)`, si el rango es grande. Pero si solo necesitamos calcular el número de factores primos diferentes de un solo entero  $N$ , aunque sea grande, puede resultar más rápido usar `numDiffPF(N)`.

### Ejercicio 5.5.7.1

La función `EulerPhi(N)`, de la sección 5.5.6, se puede reescribir para que utilice la criba modificada. Haz los cambios necesarios.

### Ejercicio 5.5.7.2\*

¿Podemos escribir el código de la criba modificada para las otras funciones de la sección 5.5.6 (es decir, además de `numDiffPF(N)` y `EulerPhi(N)`), sin aumentar la complejidad de tiempo de la criba? Si es posible, escribe el código. Si no, explícalo.

## 5.5.8 Aritmética modular

Algunos cálculos matemáticos en problemas de programación terminan dando un positivo muy grande (o un negativo muy pequeño) en los resultados intermedios o finales, que queda fuera del ámbito de los tipos de datos de enteros más grandes integrados en los lenguajes de programación (a día de hoy, el `long long` de C++ o el `long` de Java, ambos de 64 bits). En la sección 5.5.5, hemos visto una forma de calcular enteros grandes con precisión. En la sección 5.5.5, hemos visto otra forma de trabajar con enteros grandes, a través de sus factores primos. En otros problemas, solo nos interesa el resultado *módulo* un número (normalmente primo), de forma que los resultados intermedios, o finales, siempre quedan dentro del tipo de datos de enteros integrado. En esta subsección veremos esos tipos de problemas.

Por ejemplo, en el problema UVa 10176 - Ocean Deep! Make it shallow!! se nos pide que convirtamos un número binario largo (de hasta 100 dígitos) en decimal. Un cálculo rápido nos indica que el número más grande posible es  $2^{100}-1$ , lo que queda fuera del alcance de un entero de 64 bits. Sin embargo, el problema solo pregunta si el resultado es divisible por 131071 (un número primo). Así, lo que tenemos que hacer es convertir el número binario a decimal dígito a dígito, mientras realizamos operaciones módulo 131071 en los resultados intermedios. Si el resultado final es 0, entonces el *número binario original* (que nunca hemos calculado por completo), es divisible por 131071.

### Ejercicio 5.5.8.1

¿Qué afirmaciones son válidas? El símbolo ‘%’ indica una operación de módulo.

1.  $(a + b - c) \% s = ((a \% s) + (b \% s) - (c \% s) + s) \% s$
2.  $(a * b) \% s = (a \% s) * (b \% s)$
3.  $(a * b) \% s = ((a \% s) * (b \% s)) \% s$
4.  $(a / b) \% s = ((a \% s) / (b \% s)) \% s$
5.  $(a^b) \% s = ((a^{b/2} \% s) * (a^{b/2} \% s)) \% s$ ; asume que  $b$  es par.

## 5.5.9 Euclídeo extendido: solución de la ecuación diofántica lineal

Problema de referencia: supongamos que un ama de casa compra manzanas y naranjas por un precio total de 8,39 SGD. Una manzana cuesta 25 céntimos y una naranja 18. ¿Cuántas ha comprado de cada clase?

Este problema se puede modelar como una ecuación lineal con dos variables:  $25x + 18y = 839$ . Como sabemos que tanto  $x$  como  $y$  deben ser enteros, esta ecuación se denomina ecuación *diofántica* lineal. Es posible resolver la ecuación diofántica lineal con dos variables aunque solo tengamos una ecuación.

Digamos que  $a$  y  $b$  son enteros con  $d = \gcd(a, b)$ . La ecuación  $ax + by = c$  no tiene soluciones integrales si  $d \nmid c$  no es cierto. Pero si  $d \mid c$ , entonces hay infinitas soluciones integrales. La primera solución  $(x_0, y_0)$  se encuentra usando el algoritmo *euclídeo extendido* que mostramos a continuación, y el resto se pueden derivar de  $x = x_0 + (b/d)n$ ,  $y = y_0 - (a/d)n$ , donde  $n$  es

un entero. Los problemas de los concursos de programación suelen tener restricciones añadidas, para garantizar una salida finita (y única).

```
1 // almacenar x, y, d como variables globales
2 void extendedEuclid(int a, int b) {
3 if (b == 0) { x = 1; y = 0; d = a; return; } // caso base
4 extendedEuclid(b, a % b); // similar al gcd original
5 int x1 = y;
6 int y1 = x - (a / b) * y;
7 x = x1;
8 y = y1;
9 }
```

Usando `extendedEuclid`, podemos resolver el problema de referencia anterior: la ecuación diofántica lineal con dos variables  $25x + 18y = 839$ .

$a = 25, b = 18$ , `extendedEuclid(25, 18)` resulta en  $x = -5, y = 7, d = 1$ , o  $25 \times (-5) + 18 \times 7 = 1$ .

Multiplicamos los componentes izquierdo y derecho de la ecuación anterior por  $839/gcd(25, 18) = 839$ :  $25 \times (-4195) + 18 \times 5873 = 839$ . Por lo tanto,  $x = -4195 + (18/1)n$  e  $y = 5873 - (25/1)n$ .

Como necesitamos un número no negativo de  $x$  e  $y$  (un número no negativo de manzanas y naranjas), tenemos dos restricciones adicionales:  $-4195 + 18n \geq 0$  y  $5873 - 25n \geq 0$ , o  $4195/18 \leq n \leq 5873/25$  o  $233,05 \leq n \leq 234,92$ .

El único entero posible para  $n$  ahora es 234. Por lo que la solución única resulta ser  $x = -4195 + 18 \times 234 = 17$  e  $y = 5873 - 25 \times 234 = 23$ , es decir, 17 manzanas (de 25 céntimos cada una) y 23 naranjas (de 18 céntimos cada uno), para un total de 8,39 SGD.

### 5.5.10 Comentarios sobre teoría de números en concursos de programación

Existen muchos otros problemas de teoría de números que no hemos incluido en este libro. Según nuestra experiencia, suelen aparecer problemas de teoría de números en los ICPC, sobre todo en Asia. Por lo tanto, es una buena idea que un miembro del equipo se prepare específicamente en la materia incluida en este libro y en otros.

## Ejercicios de programación

### Números primos

1. UVa 00406 - Prime Cuts
2. **UVa 00543 - Goldbach's Conjecture \*** (criba; tomar los centrales)  
(criba; búsqueda completa; conjectura de Christian Goldbach (actualizada por Leonhard Euler): todo número  $\geq 4$  se puede expresar como la suma de dos números primos; similar a UVa 686, 10311 y 10948)
3. UVa 00686 - Goldbach's Conjecture (II) (similar a UVa 543, 10311 y 10948)
4. UVa 00897 - Annagramatic Primes (criba; solo hay que comprobar las rotaciones de los dígitos)
5. UVa 00914 - Jumping Champion (criba; cuidado con  $L$  y  $U < 2$ )
6. **UVa 10140 - Prime Distance \*** (criba; barrido lineal)
7. UVa 10168 - Summation of Four Primes (*backtracking* con poda)

8. UVa 10311 - Goldbach and Euler
9. **UVa 10394 - Twin Primes \***  
 (análisis de casos; fuerza bruta; similar a UVa 543, 686 y 10948)  
 (criba; comprobar si  $p$  y  $p + 2$  son primos; si lo son, serán gemelos; cálculo previo del resultado)  
 (*ad hoc*; cálculo previo de las respuestas)
10. UVa 10490 - Mr. Azad and his Son
11. UVa 10650 - Determinate Prime
12. UVa 10852 - Less Prime
13. UVa 10948 - The Primary Problem
14. UVa 11752 - The Super ...  
 (3 primos consecutivos equidistantes)  
 (criba;  $p = 1$ , buscar el primer primo  $\geq \frac{n}{2} + 1$ )  
 (conjetura de Goldbach; similar a UVa 543, 686 y 10311)  
 (probar base 2 hasta  $2^{16}$ ; potencia compuesta; ordenar)

## GCD y/o LCM

1. UVa 00106 - Fermat vs. Phytagoras
2. UVa 00332 - Rational Numbers from ...  
 (fuerza bruta; usar GCD para obtener 3-tuplas de primos relativos)
3. UVa 00408 - Uniform Generator
4. UVa 00412 - Pi
5. **UVa 10407 - Simple Division \***  
 (usar GCD para simplificar la fracción)
6. **UVa 10892 - LCM Cardinality \***  
 (problema de búsqueda de ciclos con solución fácil: es una buena elección si  $step < mod$  y  $GCD(step, mod) == 1$ )  
 (fuerza bruta; GCD para encontrar elementos sin factor en común)  
 (restar el conjunto s con  $s[0]$ ; buscar GCD)
7. UVa 11388 - GCD LCM
8. UVa 11417 - GCD
9. *UVa 11774 - Doom's Day*
10. **UVa 11827 - Maximum GCD \***  
 (número de pares divisores de  $N$ :  $(m, n)$  tal que  $gcd(m, n) = 1$ )  
 (entender la relación entre GCD y LCM)  
 (usar fuerza bruta, pues la entrada es pequeña)  
 (buscar patrón que implica GCD con casos de prueba pequeños)
11. *UVa 12068 - Harmonic Mean*  
 (GCD de muchos números; entrada pequeña)  
 (implica fracciones; usar LCM y GCD)

## Factorial

1. **UVa 00324 - Factorial Frequencies \***  
 (contar dígitos de  $n!$  hasta 366!)
2. UVa 00568 - Just the Facts
3. **UVa 00623 - 500 (factorial) \***  
 (se puede usar BigInteger de Java; lento, pero AC)
4. UVa 10220 - I Love Big Numbers
5. UVa 10323 - Factorial. You Must ...  
 (fácil con BigInteger de Java)
6. **UVa 10338 - Mischievous Children \***  
 (usar BigInteger de Java; cálculo previo)  
 (desbordamiento:  $n > 13$  / -n impar; undimiento:  $n < 8$  / -n par; en realidad, el factorial de un número negativo no está definido)  
 (usar long long para almacenar hasta  $20!$ )

## Búsqueda de factores primos

1. **UVa 00516 - Prime Land \***  
 (problema que implica factorización de potencias de primos)
2. **UVa 00583 - Prime Factors \***  
 (problema de factorización básico)
3. UVa 10392 - Factoring Large Numbers  
 (enumerar los factores primos de la entrada)
4. **UVa 11466 - Largest Prime Divisor \***  
 (usar criba eficiente para obtener los factores primos más grandes)

## Trabajo con factores primos

1. UVa 00160 - Factors and Factorials  
 (cálculo previo de primos pequeños ya que los factores primos de  $100!$  son  $< 100$ )
2. UVa 00993 - Product of digits  
 (buscar divisores de 9 a 1; similar a UVa 10527)
3. UVa 10061 - How many zeros & how ...  
 (en decimal, '10'; con un cero, se factoriza  $2 \times 5$ )

4. **UVa 10139 - Factovisors \***  
 5. UVa 10484 - Divisibility of Factors  
 6. UVa 10527 - Persistent Numbers  
 7. UVa 10622 - Perfect P-th Power  
 8. **UVa 10680 - LCM \***  
 9. UVa 10780 - Again Prime? No time.  
 10. UVa 10791 - Minimum Sum LCM  
 11. UVa 11347 - Multifactorials  
 12. *UVa 11395 - Sigma Function*  
 13. **UVa 11889 - Benefit \***
- (comparar factores primos de  $n!$  y  $m$ )  
 (factores primos de factorial;  $D$  puede ser negativo)  
 (similar a UVa 993)  
 (GCD de todas las potencias primas; anotar si  $x$  es negativo)  
 (usar primefactors([1 .. N]) para obtener  $\text{LCM}(1, 2, \dots, N)$ )  
 (similar a UVa 10139)  
 (analizar los factores primos de  $N$ )  
 (factorización de potencias de primos; numDiv(N))  
 (la clave: un número cuadrado multiplicado por potencias de 2, es decir,  
 $2^k \times i^2$  para  $k \geq 0$ ,  $i \geq 1$  tiene una suma *impar* de sus divisores)  
 (LCM; implica factorización de primos)

### Funciones que implican factores primos

1. **UVa 00294 - Divisors \***  
 2. UVa 00884 - Factorial Factors  
 3. UVa 01246 - Find Terrorists  
 4. **UVa 10179 - Irreducible Basic ... \***  
 5. UVa 10299 - Relatives  
 6. UVa 10820 - Send A Table  
 7. *UVa 10958 - How Many Solutions?*  
 8. UVa 11064 - Number Theory  
 9. UVa 11086 - Composite Prime  
 10. UVa 11226 - Reaching the fix-point  
 11. *UVa 11353 - A Different kind of Sorting*  
 12. **UVa 11728 - Alternate Task \***  
 13. *UVa 12005 - Find Solutions*
- (numDiv(N))  
 (numPF(N); cálculo previo)  
 (LA 4340 - Amrita08; numDiv(N))  
 (EulerPhi(N))  
 (EulerPhi(N))  
 ( $a[i] = a[i-1] + 2 * \text{EulerPhi}(i)$ )  
 ( $2 * \text{numDiv}(n * m * p * p) - 1$ )  
 ( $N - \text{EulerPhi}(N) - \text{numDiv}(N)$ )  
 (buscar números  $N$  con  $\text{numPF}(N) == 2$ )  
 (sumPF(N); obtener longitud; DP)  
 (numPF(N); variante de ordenación)  
 (sumDiv(N))  
 (numDiv(4N-3))

### Criba modificada

1. **UVa 10699 - Count the ... \***  
 2. **UVa 10738 - Riemann vs. Mertens \***  
 3. **UVa 10990 - Another New Function \***  
 4. UVa 11327 - Enumerating Rational ...  
 5. *UVa 12043 - Divisors*
- (numDiffPF(N) para un rango de  $N$ )  
 (numDiffPF(N) para un rango de  $N$ )  
 (criba modificada para calcular un rango de valores  $F_i$  eulerianos; usar  
 DP para calcular la profundidad de los valores  $F_i$ ; por último, usar DP  
 de suma de rango unidimensional máxima para mostrar la respuesta)  
 (cálculo previo de EulerPhi(N))  
 (sumDiv(N) y numDiv(N); fuerza bruta)

### Aritmética modular

1. UVa 00128 - Software CRC  
 2. **UVa 00374 - Big Mod \***  
 3. UVa 10127 - Ones  
 4. UVa 10174 - Couple-Bachelor-Spinster ...  
 5. **UVa 10176 - Ocean Deep; Make it ... \***
- $((a \times b) \% s = ((a \% s) \times (b \% s)) \% s)$   
 (se resuelve con modPow de BigInteger de Java; o escribe tu propio  
 código, sección 9.21)  
 (si no hay factores de 2 y 5 implica que no hay ceros al final)  
 (no hay número de Spinster)  
 (convertir binario a decimal, dígito a dígito; módulo 131071 para el  
 resultado intermedio)

## 6. UVa 10212 - The Last Non-zero ... \*

(multiplicar números desde  $N$  hasta  $N-M+1$ ; usar  $/10$  para eliminar ceros al final, y entonces usar  $\%1000$  millones para guardar únicamente los últimos (máximo 9) dígitos distintos de cero)  
(mantener los valores pequeños mediante módulo)  
(combinación de truco logarítmico para obtener los tres primeros dígitos y truco 'big mod' para los tres últimos)

## Euclídeo extendido

### 1. UVa 10090 - Marbles \*

(usar solución para la ecuación diofántica lineal)

### 2. UVa 10104 - Euclid Problem \*

(problema euclídeo extendido puro)

### 3. UVa 10633 - Rare Easy Problem

(digamos que  $C = N-M$  (la entrada dada),  $N = 10a+b$  ( $N$  tiene, al menos, dos dígitos, siendo  $b$  el último) y  $M = a$ ; ahora el problema trata de encontrar la solución de la ecuación diofántica lineal:  $9a+b = C$ )

(usa euclídeo extendido)

### 4. UVa 10673 - Play with Floor and Ceil \*

## Otros problemas de teoría de números

### 1. UVa 00547 - DDF

(problema sobre secuencia 'constante en algún momento')

### 2. UVa 00756 - Biorhythms

(teorema del resto chino)

### 3. UVa 10110 - Light, more light \*

(comprobar si  $n$  es un número cuadrado)

### 4. UVa 10922 - 2 the 9s

(verificar divisibilidad por 9)

### 5. UVa 10929 - You can say 11

(verificar divisibilidad por 11)

### 6. UVa 11042 - Complex, difficult and ...

(análisis de casos; solo 4 salidas posibles)

### 7. UVa 11344 - The Huge One \*

(usar teoría de divisibilidad de  $[1 \dots 12]$ )

### 8. UVa 11371 - Number Theory for ... \*

(se proporciona la estrategia de resolución)

## Perfiles de los inventores de algoritmos

**John Pollard** (nacido en 1941) es un matemático británico, que inventó algoritmos para la factorización de números grandes (el algoritmo rho de Pollard) y para el cálculo de logaritmos discretos (no tratados en este libro).

**Richard Peirce Brent** (nacido en 1946) es un matemático y científico de la computación australiano. Su campo de investigación incluye la teoría de números (en particular la factorización), generadores de números aleatorios, arquitectura de ordenadores y análisis de algoritmos. Ha inventado, o coinventado, varios algoritmos matemáticos. En este libro, tratamos el algoritmo de búsqueda de ciclos de Brent (ver el **ejercicio 5.7.1\***) y la mejora de Brent al algoritmo rho de Pollard (ver el **ejercicio 5.5.4.2\*** y la sección 9.26).

## 5.6 Teoría de probabilidad

La **teoría de probabilidad** es la rama de las matemáticas que se ocupa del análisis de fenómenos aleatorios. Aunque un hecho como el lanzamiento independiente (equilibrado) de una moneda es aleatorio, una secuencia de eventos aleatorios mostrará ciertos patrones estadísticos, si es lo suficientemente larga. Esto se puede estudiar y predecir. Por ejemplo, la probabilidad de que aparezca cara es de  $\frac{1}{2}$  (la misma que de que aparezca cruz). Por lo tanto, si lanzamos una moneda (equilibrada)  $n$  veces, la *expectativa* es que obtendremos una cara  $\frac{n}{2}$  veces.

En los concursos de programación, los problemas de probabilidad se pueden resolver con:

- Fórmula cerrada. Para estos problemas, hay que deducir la fórmula requerida, normalmente en  $O(1)$ . Por ejemplo, veamos cómo deducir la solución de UVa 10491 - Cows and Cars, que es una versión general del concurso de televisión ‘El problema de Monty Hall’<sup>15</sup>.

Tienes un número NCOWS de puertas, detrás de las que hay vacas, un número NCARS de puertas en las que hay coches y un número NSHOW de puertas (con vacas), que abre el presentador. A partir de ahí, tienes que determinar la probabilidad de ganar un coche, asumiendo que siempre cambias a una de las puertas sin abrir.

El primer paso es identificar que hay dos maneras de ganar un coche. O eliges, inicialmente, una vaca y, después, cambias a un coche, o eliges, inicialmente, un coche y cambias a otro coche. A continuación, vemos cómo calcular la probabilidad de cada situación.

Al inicio, la posibilidad de empezar eligiendo una vaca es  $(NCOWS / (NCOWS+NCARS))$ . Luego, la de cambiar a un coche es  $(NCARS / (NCARS+NCOWS-NSHOW-1))$ . Multiplicamos estos dos valores entre sí, para obtener la probabilidad del primer caso. El -1 corresponde a la puerta que ya has elegido, porque no puedes cambiar a esa.

La probabilidad del segundo caso se puede calcular de forma similar. La posibilidad de empezar eligiendo un coche es  $(NCARS / (NCARS+NCOWS))$ . Entonces, la posibilidad de cambiar a otro coche es  $((NCARS-1) / (NCARS+NCOWS-NSHOW-1))$ . Los -1 corresponden a los coches que ya has elegido.

La suma de las probabilidades de los dos casos es la respuesta final.

- Exploración del espacio de búsqueda (de ejemplo) para contar el número de eventos (normalmente es difícil de contar y puede implicar combinatoria, sección 5.4, búsqueda completa, sección 3.2, o programación dinámica, sección 3.5) que se encuentran en ese espacio de ejemplo (normalmente más fácil de contar). Ejemplos:

- ‘UVa 12024 - Hats’, es un problema sobre  $n$  personas que dejan sus  $n$  sombreros en el guardarropa, durante un evento. Al terminar el acto, las  $n$  personas recuperan sus sombreros. Algunas reciben un sombrero equivocado. Calcula la posibilidad de que *todos* terminen con un sombrero equivocado.

Este problema, al tener aquí  $n \leq 12$ , se puede resolver por fuerza bruta y cálculo previo, probando las  $n!$  permutaciones y viendo el número de veces que se produce el evento requerido. Pero un concursante de perfil más matemático puede utilizar, en su lugar, esta fórmula de desarreglo (DP):  $A_n = (n-1) \times (A_{n-1} + A_{n-2})$ .

<sup>15</sup>Se trata de un interesante rompecabezas de probabilidad. Animamos a los lectores que no hayan oído hablar de este problema a que investiguen en internet y lean su historia. En el problema original, NCOWS = 2, NCARS = 1 y NSHOW = 1. La probabilidad al quedarte con la elección original es  $\frac{1}{3}$ , y al cambiar a una de las puertas no abiertas es  $\frac{2}{3}$ , por lo que siempre resulta beneficioso cambiar.

- ‘UVa 10759 - Dice Throwing’ tiene un enunciado corto: se lanzan  $n$  dados cúbicos normales. ¿Cuál es la probabilidad de que la suma de todos ellos sea, al menos,  $x$  (límites:  $1 \leq n \leq 24$ ,  $0 \leq x < 150$ )?

El espacio de ejemplo (el denominador del valor de la probabilidad) es muy fácil de calcular. Es  $6^n$ .

El número de eventos es un poco más difícil. Necesitamos DP (simple), pues hay muchos problemas superpuestos. El estado es (*dados\_restantes, puntos*), donde *dados\_restantes* es el número de dados que todavía podemos lanzar (empezando en  $n$ ) y *puntos* la puntuación acumulada hasta el momento (empezando en 0). Nos sirve la DP, porque este problema solo tiene  $24 \times (24 \times 6) = 3456$  estados distintos.

Cuando *dados\_restantes* = 0, devolvemos 1 (evento) si *puntos*  $\geq x$ , o 0 en caso contrario. Mientras *dados\_restantes* > 0, seguimos lanzando dados. El resultado  $v$  de cada dado puede ser de un valor entre seis, y nos movemos al estado (*dados\_restantes-1, puntos+v*). Sumamos todos los eventos.

Un requisito final, es que tenemos que utilizar el GCD (ver sección 5.5.2) para simplificar la fracción de la probabilidad. En otros problemas se nos podría pedir mostrar el valor de la probabilidad correcto, hasta un cierto dígito después de la coma decimal.

## Ejercicios de programación

### Ejercicios de programación sobre teoría de la probabilidad:

1. UVa 00542 - France '98  
(divide y vencerás)
2. UVa 10056 - What is the Probability?  
(obtener la fórmula cerrada)
3. UVa 10218 - Let's Dance  
(probabilidad y un poco de coeficientes binomiales)
4. UVa 10238 - Throw the Dice  
(DP; s: (dados\_restantes, puntuación); probar F valores; BigInteger de Java; no se nos pide que simplifiquemos la fracción; similar a UVa 10759)
5. UVa 10328 - Coin Toss  
(DP; estado unidimensional; BigInteger de Java)
6. **UVa 10491 - Cows and Cars**\*  
(dos formas de conseguir el coche: elegir primero una vaca y después cambiar a un coche, o elegir primero un coche y después cambiar a otro)
7. **UVa 10759 - Dice Throwing**\*  
(DP; s: (dados\_restantes, puntuación); probar 6 valores, gcd; similar a UVa 10238)
8. UVa 10777 - God, Save me  
(valor esperado)
9. UVa 11021 - Tribbles  
(probabilidad)
10. **UVa 11176 - Winning Streak**\*  
(DP; s: (n\_restantes, racha\_max); n\_restantes = el número de partidas restantes; racha\_max = el mayor número de victorias consecutivas; t: perder esta partida, o ganar las siguientes W = [1..n\_restantes] partidas y perder la (W+1)-ésima; caso especial si W = n\_restantes)  
(fuerza bruta iterativa; probar todas las posibilidades)
11. UVa 11181 - Probability (bar) Given  
(un poco de geometría)
12. UVa 11346 - Probability  
(problema de la ruina del apostador)
13. UVa 11500 - Vampires  
(p[i] = ticket[i] / total; usar GCD para simplificar la fracción)
14. UVa 11628 - Another lottery  
(desarreglo)
15. UVa 12024 - Hats  
(probabilidad simple)
16. UVa 12114 - Bachelor Arithmetic  
(problema sencillo de valor esperado; usar DP)
17. UVa 12457 - Tennis contest  
(fuerza bruta en los  $n$  pequeños para ver que la respuesta es muy fácil)
18. UVa 12461 - Airplane

## 5.7 Búsqueda de ciclos

Dada una función  $f : S \rightarrow S$  (que conecta un número natural de un *conjunto finito*  $S$  a otro número natural del mismo conjunto finito  $S$ ) y un valor inicial  $x_0 \in N$ , la secuencia de **valores iterados de la función**:  $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \dots, x_i = f(x_{i-1}), \dots\}$  utilizará, en algún momento, el mismo valor dos veces, es decir  $\exists i \neq j$  de forma que  $x_i = x_j$ . Una vez que ocurra esto, la secuencia debe repetir el ciclo de valores desde  $x_i$  hasta  $x_{j-1}$ . Digamos que  $\mu$  (el inicio del ciclo) es el índice  $i$  más pequeño, y  $\lambda$  (la longitud del ciclo) es el entero positivo más pequeño de forma que  $x_\mu = x_{\mu+\lambda}$ . El problema de la **búsqueda de ciclos** se define como el problema de la búsqueda de  $\mu$  y  $\lambda$  dados  $f(x)$  y  $x_0$ .

Por ejemplo, en el problema UVa 350 - Pseudo-Random Numbers, tenemos un generador de números pseudoaleatorios  $f(x) = (Z \times x + I) \% M$  con  $x_0 = L$ , y queremos saber la longitud de la secuencia antes de que se repita algún número (la  $\lambda$ ). Un buen generador de números pseudoaleatorios debe tener una  $\lambda$  grande. Si no, los números no parecerán ‘aleatorios’.

Probemos el proceso con el caso de prueba de ejemplo  $Z = 7, I = 5, M = 12, L = 4$ , de forma que tenemos  $f(x) = (7 \times x + 5) \% 12$  y  $x_0 = 4$ . La secuencia de los valores iterados de la función es  $\{4, 9, 8, 1, 0, 5, 4, 9, 8, 1, 0, 5, \dots\}$ . Tenemos  $\mu = 0$  y  $\lambda = 6$ , pues  $x_0 = x_{\mu+\lambda} = x_{0+6} = x_6 = 4$ . La secuencia de valores iterados de la función forma un ciclo a partir del índice 6.

En otro caso de prueba  $Z = 3, I = 1, M = 4, L = 7$ , tenemos  $f(x) = (3 \times x + 1) \% 4$  y  $x_0 = 7$ . La secuencia de valores iterados de la función es  $\{7, 2, 3, 2, 3, \dots\}$ . En esta ocasión,  $\mu = 1$  y  $\lambda = 2$ .

### 5.7.1 Soluciones utilizando estructuras de datos eficientes

Un algoritmo sencillo, que funcionará en *muchos casos* del problema de búsqueda de ciclos, utiliza una estructura de datos eficiente, para almacenar pares de información que indiquen que un número  $x_i$  ha aparecido en la iteración  $i$ , en la secuencia de valores iterados de la función. Después, para el  $x_j$  que encontraremos ( $j > i$ ), comprobamos si  $x_j$  ya está almacenado en la estructura de datos. Si lo está, implica que  $x_j = x_i$ ,  $\mu = i$ ,  $\lambda = j-i$ . Este algoritmo se ejecuta en  $O((\mu + \lambda) \times DS\_cost)$ , donde  $DS\_cost$  es el coste por cada operación en la estructura de datos (inserción/búsqueda). El algoritmo requiere un espacio de, al menos,  $O(\mu + \lambda)$  para almacenar los valores anteriores.

En muchos problemas de búsqueda de ciclos con un  $S$  bastante grande (y probablemente  $\mu + \lambda$  también grande), podemos utilizar un `map` de la STL de C++ o un `TreeMap` de Java, con espacio  $O(\mu + \lambda)$ , para almacenar/comprobar los índices de iteración de los valores anteriores en tiempo  $O(\log(\mu + \lambda))$ . Pero si lo único que necesitamos es detener el algoritmo al encontrar el *primer* número repetido, podemos utilizar `set` de la STL de C++ o `TreeSet` de Java en su lugar.

En otros problemas de búsqueda de ciclos, con un  $S$  relativamente pequeño (y, probablemente,  $\mu + \lambda$  pequeño), podemos utilizar una tabla de direccionamiento directo de espacio  $O(|S|)$ , para almacenar/comprobar los índices de iteración de los valores anteriores en tiempo  $O(1)$ . Aquí, intercambiamos espacio en memoria por velocidad de ejecución.

### 5.7.2 Algoritmo de búsqueda de ciclos de Floyd

Existe un algoritmo mejor, llamado algoritmo de búsqueda de ciclos de Floyd, que se ejecuta con complejidad de tiempo de  $O(\mu + \lambda)$  y *solo* utiliza  $O(1)$  espacio en memoria, mucho menos

que las versiones más sencillas vistas antes. Este algoritmo también se llama de ‘la liebre y la tortuga’. Tiene tres componentes que describimos a continuación, usando el problema UVa 350 visto antes, con  $Z = 3$ ,  $I = 1$ ,  $M = 4$  y  $L = 7$ .

### Método eficiente de detección de un ciclo: búsqueda de $k\lambda$

Observemos que por cada  $i \geq \mu$ ,  $x_i = x_{i+k\lambda}$ , donde  $k > 0$ , por ejemplo, en la tabla 5.2,  $x_1 = x_{1+1 \times 2} = x_3 = x_{1+2 \times 2} = x_5 = 2$ , y así sucesivamente. Si establecemos  $k\lambda = i$ , obtenemos  $x_i = x_{i+i} = x_{2i}$ . El algoritmo de búsqueda de ciclos de Floyd utiliza este truco.

| paso   | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |
|--------|-------|-------|-------|-------|-------|-------|-------|
|        | 7     | 2     | 3     | 2     | 3     | 2     | 3     |
| Inicio | TH    |       |       |       |       |       |       |
| 1      |       | T     | H     |       |       |       |       |
| 2      |       |       | I     |       |       | H     |       |

Tabla 5.2: Parte 1: búsqueda de  $k\lambda$ ,  $f(x) = (3 \times x + 1) \% 4$ ,  $x_0 = 7$

El algoritmo de búsqueda de ciclos de Floyd mantiene dos punteros, llamados ‘tortuga’ (el lento) en  $x_i$  y ‘liebre’ (el rápido, que va saltando) en  $x_{2i}$ . Inicialmente, ambos están en  $x_0$ . A cada paso del algoritmo, la tortuga se mueve *un paso* a la derecha y la liebre se mueve *dos pasos* a la derecha<sup>16</sup> en la secuencia. Entonces, el algoritmo compara los valores de la secuencia en esos dos punteros. El menor valor de  $i > 0$ , en el que la tortuga y la liebre apunten a valores iguales, es el valor de  $k\lambda$  (múltiplo de  $\lambda$ ). Deduciremos la  $\lambda$  real a partir de  $k\lambda$ , utilizando los dos pasos siguientes. En la tabla 5.2, cuando  $i = 2$ , tenemos  $x_2 = x_4 = x_{2+2} = x_{2+k\lambda} = 3$ . Así,  $k\lambda = 2$ . En este ejemplo, veremos que  $k$  será, en algún momento, 1, igual que también será  $\lambda = 2$ .

### Búsqueda de $\mu$

A continuación, devolvemos la liebre a  $x_0$  y mantenemos a la tortuga en su posición actual. Ahora, avanzamos *ambos* punteros hacia la derecha, un paso cada vez, manteniendo el espacio  $k\lambda$  entre ellos. Cuando la tortuga y la liebre apunten al mismo valor, habremos encontrado la *primera* repetición de longitud  $k\lambda$ . Como  $k\lambda$  es un múltiplo de  $\lambda$ , debe ser cierto que  $x_\mu = x_{\mu+k\lambda}$ . La primera vez que encontraremos la primera repetición de longitud  $k\lambda$ , tendremos el valor de  $\mu$ . En la tabla 5.3, encontramos que  $\mu = 1$ .

| paso | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |
|------|-------|-------|-------|-------|-------|-------|-------|
|      | 7     | 2     | 3     | 2     | 3     | 2     | 3     |
| 1    | H     |       | T     |       |       |       |       |
| 2    |       | H     |       | I     |       |       |       |

Tabla 5.3: Parte 2: búsqueda de  $\mu$

<sup>16</sup>Para movernos un paso a la derecha desde  $x_i$ , utilizamos  $x_i = f(x_i)$ . Para movernos dos pasos desde  $x_i$ , utilizamos  $x_i = f(f(x_i))$ .

## Búsqueda de $\lambda$

Una vez que tenemos  $\mu$ , dejamos a la tortuga en su posición actual y colocamos la liebre a su lado. Ahora, movemos la liebre iterativamente hacia la derecha, de uno en uno. La liebre apuntará a un valor igual al de la tortuga, por primera vez, después de  $\lambda$  pasos. En la tabla 5.4, después de que la liebre se mueva una vez,  $x_3 = x_{3+2} = x_5 = 2$ . Así,  $\lambda = 2$ .

| paso | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |
|------|-------|-------|-------|-------|-------|-------|-------|
|      | 7     | 2     | 3     | 2     | 3     | 2     | 3     |
| 1    |       |       |       | T     | H     |       |       |
| 2    |       |       |       | I     |       | H     |       |

Tabla 5.4: Parte 3: búsqueda de  $\lambda$

Por lo tanto, devolvemos  $\mu = 1$  y  $\lambda = 2$  para  $f(x) = (3 \times x + 1) \% 4$  y  $x_0 = 7$ . El tiempo de ejecución total del algoritmo es de  $O(\mu + \lambda)$ .

## La implementación

A continuación, incluimos la implementación en C/C++ de este algoritmo (con comentarios):

```
1 ii floydCycleFinding(int x0) { // función int f(int x) ya definida antes
2 // primera parte: buscar k*lambda, velocidad liebre = 2x la tortuga
3 int tortoise = f(x0), hare = f(f(x0)); // f(x0) es el nodo vecino a x0
4 while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
5 // segunda parte: buscar mu, liebre y tortuga tienen la misma velocidad
6 int mu = 0; hare = x0;
7 while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++; }
8 // tercera parte: buscar lambda, la liebre se mueve y la tortuga no
9 int lambda = 1; hare = f(tortoise);
10 while (tortoise != hare) { hare = f(hare); lambda++; }
11 return ii(mu, lambda);
12 }
```



ch5\_07\_UVa350.cpp



ch5\_07\_UVa350.java

### Ejercicio 5.7.1\*

Richard P. Brent inventó una versión mejorada del algoritmo de búsqueda de ciclos de Floyd que hemos visto. Estudia e implementa el algoritmo de Brent [3].

## Ejercicios de programación

### Ejercicios de programación relativos a búsqueda de ciclos:

1. UVa 00202 - Repeating Decimals  
(hacer expansión dígito a dígito hasta encontrar ciclo)  
(similar a UVa 202, salvo en el formato de salida)
  2. UVa 00275 - Expanding Fractions  
(problema muy básico de búsqueda de ciclos; ejecutar el algoritmo de búsqueda de ciclos de Floyd)  
(similar a UVa 10591)
  3. **UVa 00350 - Pseudo-Random Numbers \***  
(ciclo tras 100 pasos; BigInteger para la entrada; cálculo previo)  
(concentrarse en el último dígito)
  4. UVa 00944 - Happy Numbers  
(esta secuencia llegará a ser periódica; similar a UVa 944)
  5. UVa 10162 - Last Digit  
(búsqueda de ciclos; evaluar polaca inversa f con una stack)
  6. UVa 10515 - Power et al  
(búsqueda de ciclos; la respuesta es  $N - \lambda$ )
  7. UVa 10591 - Happy Number  
(elevar al cuadrado con dígitos limitados, hasta encontrar ciclo; el algoritmo de búsqueda de ciclos de Floyd se usa solo para detectar el ciclo, no usamos los valores de  $\mu$  o  $\lambda$ ; en su lugar, mantenemos registro del valor de la función iterada más grande hallada antes de encontrar un ciclo)
  8. UVa 11036 - Eventually periodic ...  
(usar tabla de direccionamiento directo (DAT) de tamaño 10K; extraer dígitos; el truco de programación para elevar al cuadrado 4 dígitos 'a' y obtener los 4 centrales resultantes es  $a = (a*a/100) \% 10000$ )
  9. **UVa 11053 - Flavius Josephus ... \***  
(aunque  $n$  puede ser enorme, el patrón es cíclico; buscar la longitud del ciclo  $l$  y hacer módulo  $n$  con  $l$ )
  10. UVa 11549 - Calculator Conundrum
- 
11. **UVa 11634 - Generate random ... \***
  12. UVa 12464 - Professor Lazy, Ph.D.

## 5.8 Teoría de juegos

La **teoría de juegos** es un modelo matemático para situaciones de estrategia (no necesariamente *juegos*, en su significado habitual) en el que el éxito de las decisiones de un jugador depende de las decisiones de *otros*. Muchos de los problemas de programación que implican la teoría de juegos se clasifican como **juegos de suma cero**, un término matemático para indicar que, si un jugador gana, el otro, necesariamente, pierde. Por ejemplo, los juegos del tres en raya (como UVa 10111), el ajedrez, varios juegos con números/enteros (UVa 847, 10368, 10578, 10891, 11489) y otros (UVa 10165, 10404, 11311), son juegos de dos jugadores que participan de forma alternativa (normalmente obligatoria) y en los que solo puede haber un ganador.

La pregunta común de los problemas de concursos de programación, relativos a la teoría de juegos, es si el jugador que realiza la primera acción en un juego competitivo de dos jugadores tendrá el movimiento ganador, asumiendo que ambos **juegan perfectamente**, es decir, si cada jugador elige siempre su opción óptima.

### 5.8.1 Árbol de decisión

Una solución es escribir un código recursivo que explore el **árbol de decisión** del juego (también conocido como árbol del juego). Si no hay subproblemas superpuestos, el *backtracking* recursivo

puro es una opción viable. En caso contrario, será necesaria la programación dinámica. Cada vértice describe al jugador actual y el estado actual de la partida. Todo vértice está conectado al resto de vértices que le son alcanzables, según las reglas del juego. El vértice raíz describe al jugador y estado iniciales de la partida. Si el estado de la partida en un vértice hoja es un estado ganador, significa la victoria del jugador actual (y la derrota del otro). En un vértice interno, el jugador actual elige el vértice que garantice una victoria con el margen máximo (o, si la victoria no es posible, elige el vértice con la derrota mínima). La estrategia se llama **minimax**.

Por ejemplo, en el problema UVa 10368 - Euclid's Game, hay dos jugadores: Stan (jugador 0) y Ollie (jugador 1). El estado de la partida es una 3-tupla  $(id, a, b)$ . El jugador actual  $id$  puede restar cualquier múltiplo positivo del menor de los dos números, el entero  $b$ , del mayor de los dos, el entero  $a$ , asumiendo que el número resultante no sea negativo. Siempre podemos mantener que  $a \geq b$ . Stan y Ollie juegan alternativamente, hasta que un jugador es capaz de restar un múltiplo del numero menor, para hacer que el mayor llegue a 0 y, en ese momento, gana. Stan es el primer jugador. A continuación, en la figura 5.2, se muestra el árbol de decisión de una partida con estado inicial  $id = 0$ ,  $a = 34$  y  $b = 12$ .

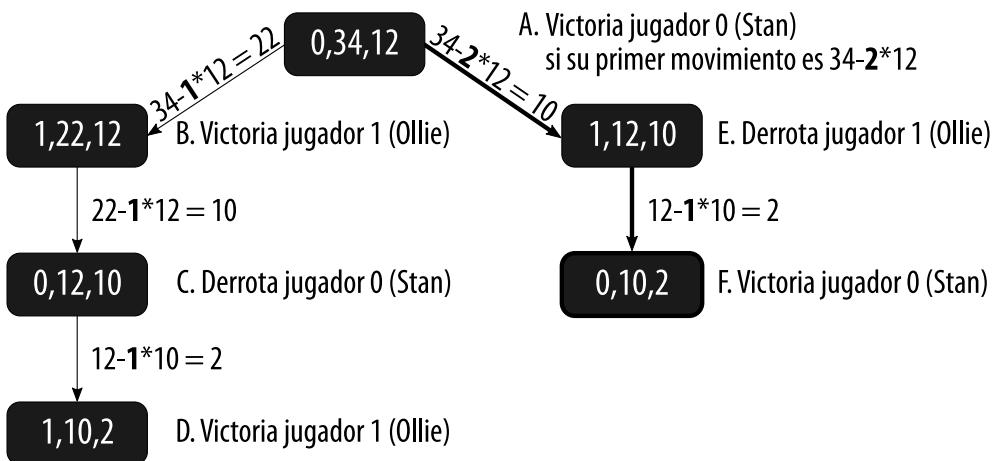


Figura 5.2: Árbol de decisión de una partida del 'Euclid's Game'

Vamos a seguir lo que ocurre en la figura 5.2. En la raíz (estado inicial), tenemos la 3-tupla  $(0, 34, 12)$ . En este punto, el jugador 0 (Stan) tiene dos opciones: restar  $a - b = 34 - 12 = 22$  y moverse al vértice  $(1, 22, 12)$  (rama izquierda), o restar  $a - 2 \times b = 34 - 2 \times 12 = 10$  y moverse al vértice  $(1, 12, 10)$  (rama derecha). Probamos ambas opciones recursivamente.

Empecemos con la rama izquierda. En el vértice  $(1, 22, 12)$  (figura 5.2.B), el jugador actual 1 (Ollie) no tiene más opción que restar  $a - b = 22 - 12 = 10$ . Ahora estamos en el vértice  $(0, 12, 10)$  (figura 5.2.C). Igualmente, Stan tiene como única opción restar  $a - b = 12 - 10 = 2$ . Nos encontramos ahora en el vértice hoja  $(1, 10, 2)$  (figura 5.2.D). Ollie tiene varias opciones, pero seguro que puede ganar, ya que  $a - 5 \times b = 10 - 5 \times 2 = 0$ , lo que implica que el vértice  $(0, 12, 10)$  es una derrota para Stan y el vértice  $(1, 22, 12)$  es una victoria para Ollie.

Exploraremos la rama derecha. En el vértice  $(1, 12, 10)$  (figura 5.2.E), el jugador actual 1 (Ollie) no tiene más opción que restar  $a - b = 12 - 10 = 2$ . Ahora estamos en el vértice hoja  $(0, 10, 2)$  (figura 5.2.F). Stan tiene varias opciones, pero seguro que ganará porque  $a - 5 \times b = 10 - 5 \times 2 = 0$ , lo que implica que el vértice  $(1, 12, 10)$  es un estado de derrota para Ollie.

Por lo tanto, para que el jugador 0 (Stan) gane esta partida, debe comenzar eligiendo  $a - 2 \times b = 34 - 2 \times 12$ , ya que es un movimiento de victoria para él (figura 5.2.A).

A nivel de implementación, el primer entero *id* en la 3-tupla se podría eliminar, ya que sabemos que las profundidades 0 (raíz), 2, 4, ..., son siempre los turnos de Stan, mientras que las profundidades 1, 3, 5, ..., lo son de Ollie. Hemos utilizado el entero *id* en la figura 5.2 para simplificar la explicación.

### 5.8.2 Mecanismos matemáticos para simplificar la solución

No todos los problemas de teoría de juegos se pueden resolver explorando el árbol de decisión *completo* de la partida, especialmente si el tamaño del árbol es grande. Si el problema implica números, podemos diseñar algunos mecanismos matemáticos para acelerar los cálculos.

Por ejemplo, en el problema UVa 847 - A multiplication game, hay dos jugadores: nuevamente Stan (jugador 0) y Ollie (jugador 1). El estado de la partida<sup>17</sup> es un entero  $p$ . El jugador actual puede multiplicar  $p$  por cualquier número entre 2 y 9. Stan y Ollie también juegan alternativamente, hasta que un jugador sea capaz de multiplicar  $p$  por un número entre 2 y 9 de forma que  $p \geq n$  ( $n$  es el número objetivo), y ahí gana. El primer jugador es Stan con  $p = 1$ .

La figura 5.3 muestra una partida de este juego de multiplicación con  $n = 17$ . Inicialmente, el jugador 0 tiene hasta 8 opciones (para multiplicar  $p = 1$  por [2..9]). Pero los 8 estados son de victoria para el jugador 1, pues este siempre podrá multiplicar el  $p$  actual por [2..9], para lograr  $p \geq 17$  (figura 5.3.B). Por lo tanto, el jugador 0 perderá siempre (figura 5.3.A).

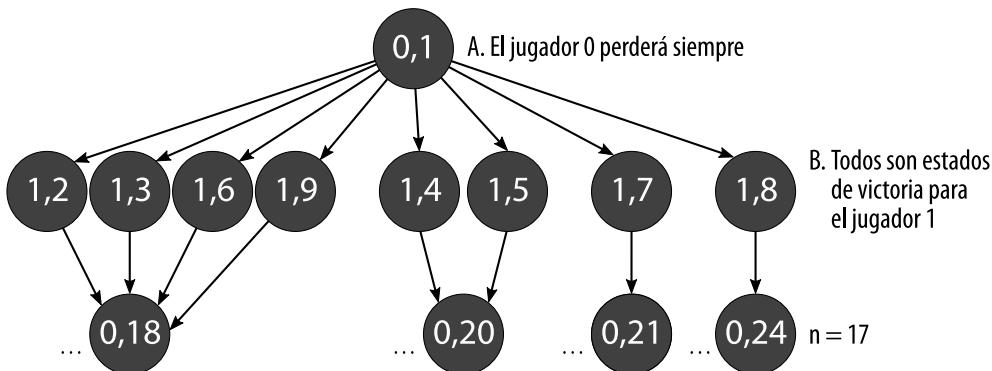


Figura 5.3: Árbol de decisión parcial de una partida de ‘A multiplication game’

Como  $1 < n < 4294967295$ , el árbol de decisión resultante, en el caso de prueba máximo, puede ser gigante. Esto se debe a que cada vértice tiene un *enorme* factor de ramificación de 8 (ya que hay 8 números para elegir entre 2 y 9). No es viable explorarlo.

Resulta que la estrategia óptima para que gane Stan es multiplicar *siempre*  $p$  por 9 (el máximo posible), mientras que Ollie multiplicará *siempre*  $p$  por 2 (el mínimo posible). Estos mecanismos de optimización se pueden deducir observando el patrón encontrado en las instancias más pequeñas del problema. Un concursante de perfil matemático puede querer demostrar esta observación, antes de programar la solución.

<sup>17</sup>Esta vez omitimos el *id* del jugador. Sin embargo, el parámetro *id* sigue apareciendo en la figura 5.3 por razones de claridad.

### 5.8.3 Juego del nim

Hay un juego especial que merece la pena mencionar, ya que aparece en concursos de programación: el *nim*<sup>18</sup>. En el *nim*, los dos jugadores se turnan para eliminar objetos de distintos montículos. En cada turno, un jugador debe eliminar *al menos un objeto*, pero puede eliminar *todos los que quiera*, siempre que estén en el mismo montículo. El estado inicial de la partida es el número de objetos  $n_i$  en cada uno de los  $k$  montículos:  $\{n_1, n_2, \dots, n_k\}$ . Hay una bonita solución para este juego. Para que el primer jugador gane, el valor de  $n_1 \wedge n_2 \wedge \dots \wedge n_k$  debe ser *distinto de cero*, donde  $\wedge$  es el operador de bits XOR (O exclusivo). Omitimos la demostración.

## Ejercicios de programación

### Ejercicios de programación relacionados con teoría de juegos:

- |                                              |                                                                                                                                                                                                                                                                                                             |
|----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00847 - A multiplication game         | (simular la partida perfecta)                                                                                                                                                                                                                                                                               |
| 2. <b>UVa 10111 - Find the Winning ... *</b> | (tres en raya; <i>minimax</i> ; <i>backtracking</i> )                                                                                                                                                                                                                                                       |
| 3. UVa 10165 - Stone Game                    | (juego del <i>nim</i> ; aplicación del teorema de Sprague-Grundy)                                                                                                                                                                                                                                           |
| 4. UVa 10368 - Euclid's Game                 | ( <i>minimax</i> ; <i>backtracking</i> )                                                                                                                                                                                                                                                                    |
| 5. UVa 10404 - Bachet's Game                 | (2 jugadores; programación dinámica)                                                                                                                                                                                                                                                                        |
| 6. UVa 10578 - The Game of 31                | ( <i>backtracking</i> ; probar todos; ver quién gana la partida)                                                                                                                                                                                                                                            |
| 7. <b>UVa 11311 - Exclusively Edible *</b>   | (teoría de juegos; reducible a juego del <i>nim</i> ; podemos ver la partida que juegan Hansel y Gretel como una partida de <i>nim</i> , donde hay 4 montículos/tartas a izquierda/debajo/derecha/encima de la cobertura; tomar la suma <i>nim</i> de estos 4 valores y, si son iguales a 0, Hansel pierde) |
| 8. <b>UVa 11489 - Integer Game *</b>         | (teoría de juegos; reducible a matemática simple)                                                                                                                                                                                                                                                           |
| 9. <i>UVa 12293 - Box Game</i>               | (analizar el árbol de la partida de instancias pequeñas, para observar la parte matemática que resuelve este problema)                                                                                                                                                                                      |
| 10. <i>UVa 12469 - Stones</i>                | (juegos; programación dinámica; poda)                                                                                                                                                                                                                                                                       |

## 5.9 Soluciones a los ejercicios no resaltados

**Ejercicio 5.2.1:** la biblioteca `<cmath>` de C/C++ tiene dos funciones: `log` (base  $e$ ) y `log10` (base 10); `Java.lang.Math` solo tiene `log` (base  $e$ ). Para obtener  $\log_b(a)$  (base  $b$ ), utilizamos el hecho de que  $\log_b(a) = \log(a) / \log(b)$ .

**Ejercicio 5.2.2:** `(int)floor(1 + log10((double)a))` devuelve el número de dígitos del número decimal  $a$ . Para contar el número de dígitos en la otra base  $b$ , podemos utilizar una fórmula similar: `(int)floor(1 + log10((double)a) / log10((double)b))`.

**Ejercicio 5.2.3:**  $\sqrt[n]{a}$  se puede reescribir como  $a^{1/n}$ . Entonces podemos utilizar una fórmula integrada como `pow((double)a, 1.0 / (double)n)` o `exp(log((double)a) * 1.0 / (double)n)`.

**Ejercicio 5.3.1.1:** es posible, manteniendo los cálculos intermedios con **módulo**  $10^6$ . Elimina los ceros del final (después de una multiplicación de  $n!$  a  $(n+1)!$ , se pueden añadir algunos ceros).

**Ejercicio 5.3.1.2:** es posible.  $9317 = 7 \times 11^3$ . También expresamos  $25!$  como sus factores

<sup>18</sup>La forma general de juegos de dos jugadores está incluida en el temario de la IOI [20], pero el *nim* no.

primos. Despues, comprobamos si hay un factor 7 (sí) y tres factores 11 (lamentablemente, no). Así que  $25!$  no es divisible por 9317. Alternativa: utilizar aritmética modular (sección 5.5.8).

**Ejercicio 5.3.2.1:** para la conversión de bases de enteros de 32 bits, utilizar `parseInt(String s, int radix)` y `toString(int i, int radix)` en la clase de Java `Integer`, que es más rápida. También se pueden utilizar `BufferedReader` y `BufferedWriter` para la E/S (sección 3.2.3).

**Ejercicio 5.4.1.1:** la fórmula cerrada de Binet para Fibonacci  $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$  debería ser correcta para un  $n$  grande. Pero como el tipo de datos de precisión doble es limitado, encontramos discrepancias en  $n$  grandes. Esta fórmula cerrada es correcta hasta  $fib(75)$ , si se implementa utilizando un tipo de datos doble típico de un programa informático. Por desgracia, esto resulta demasiado pequeño para ser útil en los problemas de concursos de programación típicos que incluyen la sucesión de Fibonacci.

**Ejercicio 5.4.2.1:**  $C(n, 2) = \frac{n!}{(n-2)! \times 2!} = \frac{n \times (n-1) \times (n-2)!}{(n-2)! \times 2} = \frac{n \times (n-1)}{2} = 0,5n^2 - 0,5n = O(n^2)$ .

**Ejercicio 5.4.4.1:** principio fundamental del conteo: si hay  $m$  formas de hacer una cosa y  $n$  formas de hacer otra, entonces hay  $m \times n$  formas de hacer las dos. Por lo tanto, la respuesta a este ejercicio es:  $6 \times 6 \times 2 \times 2 = 6^2 \times 2^2 = 36 \times 4 = 144$  resultados posibles.

**Ejercicio 5.4.4.2:** ver el principio fundamental del conteo de la cuestión anterior. La respuesta es:  $9 \times 9 \times 8 = 648$ . Inicialmente hay 9 opciones (1-9), después seguirá habiendo 9 opciones (1-9 menos 1, más 0) y, finalmente, solo habrá 8 opciones.

**Ejercicio 5.4.4.3:** una permutación es un arreglo de objetos sin repetición y donde el orden es importante. La fórmula es  ${}_nP_r = \frac{n!}{(n-r)!}$ . Por lo tanto, la respuesta a este ejercicio es:  $\frac{6!}{(6-3)!} = 6 \times 5 \times 4 = 120$  palabras de 3 letras.

**Ejercicio 5.4.4.4:** la fórmula para contar las diferentes permutaciones es:  $\frac{n!}{(n_1)! \times (n_2)! \times \dots \times (n_k)!}$  donde  $n_i$  es la frecuencia de cada letra única  $i$  y  $n_1 + n_2 + \dots + n_k = n$ . La respuesta a este ejercicio es:  $\frac{5!}{3! \times 1! \times 1!} = \frac{120}{6} = 20$ , porque hay 3 'B', 1 'O' y 1 'Y'.

**Ejercicio 5.4.4.5:** las respuestas para algunos  $n$  pequeños = 3, 4, 5, 6, 7, 8, 9 y 10 son 0, 1, 3, 7, 13, 22, 34 y 50, respectivamente. Puedes generar primero esos números, utilizando una solución de fuerza bruta. Despues, busca el patrón y utilízalo.

**Ejercicio 5.5.2.1:** multiplicar  $a \times b$  antes de dividir el resultado por  $gcd(a, b)$  tiene una probabilidad más alta de provocar un desbordamiento que  $a \times (b/gcd(a, b))$ . En el ejemplo dado, tenemos  $a = 1000000000$  y  $b = 8$ . El LCM es 1000000000, que debería caber en un entero con signo de 32 bits, y solo puede ser calculado correctamente con  $a \times (b/gcd(a, b))$ .

**Ejercicio 5.5.4.1:** como el primo más grande en `vi primes` es 9999991, este código puede manejar  $N \leq 9999991^2 = 99999820000081 \approx 9 \times 10^{13}$ . Si el factor primo más pequeño de  $N$  es mayor que 9999991, por ejemplo,  $N = 1010189899^2 = 1020483632041630201 \approx 1 \times 10^{18}$  (sigue estando dentro de la capacidad de un entero con signo de 64 bits), el código fallará o devolverá un resultado incorrecto. Si decidimos abandonar el uso de `vi primes`, en beneficio de  $PF = 3, 5, 7, \dots$  (con una comprobación especial para  $PF = 2$ ), tendremos un código más lento y el nuevo límite de  $N$  será ahora el  $N$  con el factor primo más pequeño hasta  $2^{63}-1$ . Sin embargo, si recibimos una entrada así, tendremos que utilizar los algoritmos mencionados en el **ejercicio 5.5.4.2\*** y en la sección 9.26.

**Ejercicio 5.5.4.2:** ver la sección 9.26.

**Ejercicio 5.5.5.1:**  $GCD(A, B)$  se puede obtener tomando la potencia más pequeña de los factores primos comunes de A y B.  $LCM(A, B)$  se puede obtener tomando la potencia más

grande de todos los factores primos de A y B. Así,  $\text{GCD}(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2) = 2^5 = 32$  y  $\text{LCM}(2^6 \times 3^3 \times 97^1, 2^5 \times 5^2 \times 11^2) = 2^6 \times 3^3 \times 5^2 \times 11^2 \times 97^1 = 507038400$ .

**Ejercicio 5.5.6.1:**

```
1 ll numDiffPF(ll N) {
2 ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
3 while (PF*PF <= N) {
4 if (N%PF == 0) ans++; // contar este PF una sola vez
5 while (N%PF == 0) N /= PF;
6 PF = primes[++PF_idx];
7 }
8 if (N != 1) ans++;
9 return ans;
10 }
```

```
1 ll sumPF(ll N) {
2 ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
3 while (PF*PF <= N) {
4 while (N%PF == 0) { N /= PF; ans += PF; }
5 PF = primes[++PF_idx];
6 }
7 if (N != 1) ans += N;
8 return ans;
9 }
```

**Ejercicio 5.5.7.1:** el código de la criba modificada para calcular la función indicatriz de Euler hasta  $10^6$  se incluye a continuación:

```
1 for (int i = 1; i <= 1000000; i++) EulerPhi[i] = i;
2 for (int i = 2; i <= 1000000; i++)
3 if (EulerPhi[i] == i) // i es un número primo
4 for (int j = i; j <= 1000000; j += i)
5 EulerPhi[j] = (EulerPhi[j]/i) * (i-1);
```

**Ejercicio 5.5.8.1:** las afirmaciones 2 y 4 no son válidas. Las otras 3 sí lo son.

## 5.10 Notas del capítulo

Este capítulo ha crecido significativamente desde la primera edición de este libro. Sin embargo, incluso ahora, somos conscientes de que todavía hay muchos problemas y algoritmos matemáticos que no hemos tratado, como:

- Muchos más problemas y fórmulas de **combinatoria**, menos habituales, que no hemos visto: **el lema de Burnside**, **los números de Stirling**, etc.
- Teoremas, hipótesis y conjeturas que no se podrían tratar individualmente, como **función de Carmichael**, **hipótesis de Riemann**, **comprobación del pequeño teorema de Fermat**, **teorema chino del resto**, **teorema de Sprague-Grundy**, etc.
- Solo hemos mencionado brevemente el algoritmo de búsqueda de ciclos de Brent (que es un poco más rápido que la versión de Floyd) en el **ejercicio 5.7.1\***.
- La geometría (computacional) también es parte de las matemáticas, pero como la veremos en el capítulo 7, guardamos para entonces el tratamiento de los problemas de geometría.
- Más adelante, en el capítulo 9, veremos brevemente algunos algoritmos más relacionados con las matemáticas, que son poco habituales en comparación a los vistos aquí, como, por ejemplo, la eliminación gausiana para resolver sistemas de ecuaciones lineales (sección 9.9), el algoritmo rho de Pollard (sección 9.26), la calculadora posfija y su conversión (de infija a posfija) (sección 9.27) y números romanos (sección 9.28).

Realmente, hay *muchos* temas que implican a las matemáticas. No resulta sorprendente, ya que se han investigado varios problemas matemáticos desde hace cientos de años. Algunos han sido tratados en este capítulo, muchos otros no y, sin embargo, solo aparecerán uno o dos en un conjunto de problemas. Para obtener un buen resultado en el ICPIC, es una buena idea tener, al menos, a *un buen matemático* entre los miembros del equipo, para asegurar la solución de ese par de problemas. El conocimiento matemático también es importante para los concursantes de la IOI. Aunque la cantidad de temas específicos de problemas es menor, muchas tareas de la IOI requieren algún grado de conocimiento matemático.

Damos fin a este capítulo con algunas referencias que pueden resultar interesantes al lector: lee libros de teoría de números, como [57], investiga temas matemáticos en [mathworld.wolfram.com](http://mathworld.wolfram.com) o en Wikipedia y trata de resolver muchos más ejercicios de programación relacionados con problemas matemáticos, como los que hay en las páginas web <http://projecteuler.net> [17] y <https://brilliant.org> [4].

# Capítulo 6

---

## Procesamiento de cadenas

*El genoma humano tiene aproximadamente 3,2 gigas de parejas base.*  
— Proyecto del genoma humano

### 6.1 Introducción y motivación

En este capítulo, presentamos otro tema que aparece en el ICPC, aunque no con tanta frecuencia<sup>1</sup> como los problemas de grafos y matemáticos. Hablamos del procesamiento de cadenas. Esta técnica es muy común en el campo de la investigación en *bioinformática*. Como las cadenas (por ejemplo, cadenas de ADN) con las que trabajan los investigadores son, normalmente, muy largas, se hace imprescindible utilizar estructuras de datos y algoritmos específicos. Algunos de estos problemas aparecen en los concursos del ICPC. Al dominar el contenido del presente capítulo, los concursantes del ICPC tendrán más posibilidades de resolver los problemas de procesamiento de cadenas.

En la IOI, también aparecen tareas de procesamiento de cadenas, pero normalmente no requieren estructuras de datos o algoritmos avanzados, debido a la restricción de la materia a tratar [20]. Además, los formatos de entrada y salida de las tareas de la IOI son, normalmente, sencillos<sup>2</sup>. Esto elimina la necesidad de programar procesadores de entrada o formatos de salida complejos, que sí se encuentran normalmente en los problemas del ICPC. Las tareas de la IOI que requieren procesamiento de cadenas, se suelen resolver utilizando los paradigmas de resolución de problemas mencionados en el capítulo 3. Para los concursantes de la IOI, basta con ojear las secciones de este capítulo, con la excepción de la sección 6.5, que trata del procesamiento de cadenas mediante programación dinámica. Sin embargo, creemos que puede resultar ventajoso para ellos aprender algunas técnicas avanzadas por adelantado, aunque queden fuera de la materia programática.

La estructura de este capítulo comienza con una introducción de técnicas básicas de procesamiento de cadenas y una *larga* lista de problemas de cadenas *ad hoc*, que pueden resolverse con esas técnicas básicas. Aunque los problemas *ad hoc* conforman la mayoría de los que aparecen en este capítulo, tenemos que poner de relieve que, en concursos recientes del ICPC (y

---

<sup>1</sup>Una razón potencial es que la entrada y salida de cadenas son más difíciles de procesar y formatear correctamente, haciendo que resulte menos atractiva que la de números enteros.

<sup>2</sup>La IOI de 2010 a 2012 requería que los concursantes utilizasen funciones en vez de rutinas de E/S.

la IOI), no se suelen pedir soluciones con técnicas básicas, *excepto* en el problema ‘de regalo’, que la mayoría de equipos (o concursantes) deberían ser capaces de resolver. Las secciones más importantes son las que tratan la coincidencia de cadenas (sección 6.4), el procesamiento de cadenas mediante programación dinámica (sección 6.5) y, por último, una discusión extensa sobre problemas que tratan con cadenas razonablemente **largas** (sección 6.6). La última parte trata sobre estructuras eficientes para manejar *tries*, *árboles* y *arrays* de sufijos.

## 6.2 Habilidades básicas de procesamiento de cadenas

Comenzamos este capítulo con una lista de varias técnicas *básicas* de procesamiento de cadenas, que todo programador competitivo debería conocer. En esta sección, proporcionamos una serie de pequeñas tareas que deberías poder resolver sin obviar ninguna. Puedes utilizar cualquiera de los lenguajes de programación C, C++ y/o Java. Haz lo que puedas para obtener la implementación más corta y eficiente que se te ocurra. Después, compara tus implementaciones con las nuestras (en las respuestas al final del capítulo). Si ninguna de nuestras opciones te sorprende (o, incluso, las has mejorado), significa que estás en buena forma para lidiar con varios problemas de procesamiento de cadenas, por lo que podrás pasar a las siguientes secciones. En caso contrario, dedica algo de tiempo a estudiar nuestras versiones.

1. Dado un archivo de texto que contiene únicamente caracteres alfabéticos [A-Za-z], dígitos [0-9], el espacio y el punto (‘.’), escribe un programa que lea el mencionado archivo, línea a línea, hasta que encuentre una línea que *comience con* siete puntos (“.....”). Concatena (combina) cada línea en un cadena más larga T. Cuando se combinen dos líneas, coloca un espacio entre ellas, de forma que la última palabra de la primera línea quede separada de la primera palabra de la segunda. En la entrada puede haber hasta 30 caracteres por línea y no más de 10 líneas. No hay un espacio al final de cada línea y todas terminan con el carácter de salto de línea. Nota: mostramos el texto del archivo de ejemplo de entrada ‘ch6.txt’ después de la pregunta 1.(d) y antes de la tarea 2.
  - a) ¿Sabes cómo almacenar una cadena en tu lenguaje de programación favorito?
  - b) ¿Cómo leer un texto de entrada línea a línea?
  - c) ¿Cómo concatenar (combinar) dos cadenas en una más larga?
  - d) ¿Cómo comprobar si una línea comienza con ‘.....’, para dejar de leer la entrada?

```
I love CS3233 Competitive
Programming. i also love
AlGoRiTThM
.....you must stop after reading this line as it starts with 7 dots
after the first input block, there will be one loooooooooooooong line...
```

2. Supongamos que tenemos una cadena larga T. Queremos comprobar si otra cadena P se encuentra dentro de T, notificar todos los índices donde P aparece en T, o indicar -1 si P no se encuentra en T. Por ejemplo, si T = “I love CS3233 Competitive Programming. i also love AlGoRiTThM” y P = ‘I’, la salida será, únicamente, {0} (con un índice que comienza en 0). Si la mayúscula ‘I’ y la minúscula ‘i’ se consideran diferentes, el

carácter ‘i’ de la posición {39}, no formará parte de la salida. Si  $P = \text{‘love’}$ , la salida será {2, 46} y, si  $P = \text{‘book’}$ , el resultado mostrará {-1}.

- a) ¿Cómo encontrar la primera aparición de una subcadena en una cadena (si existe)? ¿Necesitamos implementar un algoritmo de coincidencia de cadenas (como el de Knuth-Morris-Pratt, tratado en la sección 6.4, etc.) o será suficiente con utilizar funciones de una biblioteca?
  - b) ¿Cómo encontrar el resto de apariciones de una subcadena en un cadena (si existen)?
3. Supongamos ahora que queremos hacer un análisis sencillo del contenido de  $T$ , y convertir cada uno de sus caracteres en minúsculas. ¿El análisis implicará contar el número de dígitos, vocales [aeiouAEIOU] y consonantes (cualquier letra que no sea una vocal) que hay en  $T$ ? ¿Es posible realizar todas las operaciones en  $O(n)$ , donde  $n$  es la longitud de la cadena  $T$ ?
4. A continuación, queremos dividir la cadena larga  $T$  en *trozos* (subcadenas) y almacenarlos en un *array* de cadenas llamado *tokens*. Para esta tarea, los *delimitadores* de los bloques serán los espacios en blanco y los signos de puntuación (por lo tanto, dividiremos las frases en palabras). Por ejemplo, si queremos *trocear* la cadena  $T$  (en minúsculas), obtendremos los siguientes *tokens* = {‘i’, ‘love’, ‘cs3233’, ‘competitive’, ‘programming’, ‘i’, ‘also’, ‘love’, ‘algorithm’}. Después, ordenaremos lexicográficamente este *array* de cadenas<sup>3</sup> y buscaremos la cadena de menor valor lexicográfico. Es decir, habremos ordenado *tokens*: {‘algorithm’, ‘also’, ‘competitive’, ‘cs3233’, ‘i’, ‘i’, ‘love’, ‘love’, ‘programming’}. Por lo tanto, la cadena de menor valor lexicográfico de este ejemplo será ‘algorithm’.
- a) ¿Cómo troceamos una cadena?
  - b) ¿Cómo almacenamos los trozos (las cadenas más cortas) en un *array* de cadenas?
  - c) ¿Cómo ordenamos lexicográficamente un *array* de cadenas?
5. Ahora, identifiquemos la palabra que aparece más veces en  $T$ . Para resolver esta cuestión, necesitamos contar la frecuencia con la que aparece cada palabra. En el caso de  $T$ , la salida será ‘i’ o ‘love’, ya que ambas aparecen dos veces. ¿Qué estructura de datos deberíamos utilizar para esta pequeña tarea?
6. El texto dado tiene una línea más, después de la que comienza con ‘.....’, pero la longitud de esta última línea no está limitada. La tarea consistirá en contar cuántos caracteres hay en ella. ¿Cómo podemos leer una cadena si no conocemos su longitud por adelantado?



ch6\_01\_basic\_string.html



ch6\_01\_basic\_string.cpp



ch6\_01\_basic\_string.java

<sup>3</sup>Que, básicamente, consiste en ordenar las palabras como aparecerían en un diccionario.

## Perfiles de los inventores de algoritmos

**Donald Ervin Knuth** (nacido en 1938) es un informático y profesor emérito de la Stanford University. Es el autor del conocido libro sobre ciencias de la computación “*The Art of Computer Programming*”. Knuth es considerado el ‘padre’ del análisis de algoritmos. También es el creador de **TeX**, el sistema de procesamiento de textos utilizado para componer este libro.

**James Hiram Morris** (nacido en 1941) es profesor de ciencias de la computación. Es uno de los descubridores del algoritmo Knuth-Morris-Pratt, para la búsqueda de cadenas.

**Vaughan Ronald Pratt** (nacido en 1944) es profesor emérito en la Stanford University. Fue uno de los pioneros en el campo de las ciencias de la computación. Ha hecho varias contribuciones a áreas fundamentales, como los algoritmos de búsqueda, algoritmos de ordenación y comprobación de primalidad. También es uno de los descubridores del algoritmo Knuth-Morris-Pratt, para la búsqueda de cadenas.

**Saul B. Needleman** y **Christian D. Wunsch** publicaron conjuntamente el algoritmo de programación dinámica para la alineación de cadenas en 1970. Lo tratamos en este libro.

**Temple F. Smith** es un profesor de ingeniería biomédica que ayudó a desarrollar el algoritmo Smith-Waterman, junto a Michael Waterman, en 1981. El algoritmo Smith-Waterman sirve como base para comparaciones de secuencias múltiples, identificando el segmento con la mayor similitud de secuencia *local*, para identificar segmentos similares de ADN, ARN y proteínas.

**Michael S. Waterman** es profesor en la University of Southern California. Waterman es uno de los fundadores y líderes actuales en el campo de la biología computacional. Su trabajo ha contribuido a algunas de las herramientas más utilizadas en ese ámbito. En concreto, el algoritmo Smith-Waterman (desarrollado junto a Temple F. Smith) es la base de muchos programas de comparación de secuencias.

## 6.3 Problemas de procesamiento de cadenas *ad hoc*

Ahora, nos ocuparemos de algo más sencillo, los problemas de procesamiento de cadenas *ad hoc*. Hay problemas relativos a cadenas, en los concursos de programación, que se pueden resolver con habilidades de programación básicas y, quizás, con un conocimiento elemental del procesamiento de cadenas, como el tratado en la sección 6.2 anterior. Basta con leer cuidadosamente los requisitos en el enunciado del problema y programar una solución, que suele ser corta. A continuación, incluimos una lista de esos problemas de procesamiento de cadenas *ad hoc*, con algunas pistas. Hemos dividido estos ejercicios en diferentes categorías.

- Cifrar/codificar/decodificar/descifrar

A todo el mundo le gusta que sus comunicaciones digitales privadas sean seguras. Es decir, que sus mensajes (cadenas) solo puedan ser leídos por los destinatarios deseados. Se han inventado muchos sistemas de cifrado con este propósito y una buena parte (de los más sencillos) acaban convirtiéndose en problemas *ad hoc* en concursos de programación, con sus diferentes reglas de codificado/decodificado. Se puede encontrar un buen número en el Online Judge [49]. Así que vamos a dividir esta categoría en otras dos: los fáciles y los difíciles. Conviene intentar resolver algunos de ellos, especialmente los que clasificamos como **obligatorio** \*. Es conveniente tener conocimientos sobre *seguridad informática* y *criptografía* para resolverlos.

- Frecuencia de aparición

En este grupo de problemas, a los concursantes se les pide que cuenten con qué frecuencia aparecen una letra (fácil, utilizando una tabla de direccionamiento directo) o una palabra (más difícil, porque la solución pasa por utilizar un árbol de búsqueda binaria equilibrado, como `map` de la STL de C++ o `TreeMap` de Java, o una tabla de hash). Algunos de estos problemas están, en realidad, relacionados con la criptografía (la categoría anterior).

- Análisis sintáctico de la entrada

Este grupo de problemas no está dirigido a los concursantes de la IOI, ya que, en esta competición, el formato de la entrada debe ser, por definición, lo más sencilla posible. Sin embargo, tal restricción no existe en el ICPC. Los problemas de análisis sintáctico de la entrada van desde los más sencillos, que se pueden resolver con un procesador iterativo, hasta los más complejos, que implican utilizar métodos recursivos o la clase `String/Pattern` de Java.

- Se pueden resolver con la clase `String/Pattern` de Java (expresiones regulares)

Algunos (aunque no muchos) problemas de procesamiento de cadenas se pueden resolver con una línea de código<sup>4</sup>, utilizando `matches(String regex)`, `replaceAll(String regex, String replacement)` y/o otras funciones de la clase `String` de Java. Para poder hacerlo, es necesario dominar el concepto de las expresiones regulares (*regex*). No vamos a tratar las expresiones regulares con detalle, pero podemos ilustrar el tema con dos ejemplos:

1. En el problema UVa 325 Identifying Legal Pascal Real Constants, se nos pide decidir si la línea de entrada dada es una constante válida y real de Pascal. Si suponemos que la línea se almacena en `String s`, el siguiente código de Java de una línea es la solución pedida:

```
s.matches("[+-]?\d+(\.\d+([eE][+-]?\d+)?|[eE][+-]?\d+)")
```

2. En el problema UVa 494 - Kindergarten Counting Game, nos piden contar las palabras que hay en una línea dada. Aquí, una palabra se define como una secuencia consecutiva de letras (mayúsculas y/o minúsculas). Si suponemos que la línea se almacena en `String s`, el siguiente código Java de una línea es la solución pedida:

```
s.replaceAll("[^a-zA-Z]+", " ").trim().split(" ").length
```

- Formato de salida

Este es otro grupo de problemas que tampoco está dirigido a los concursantes de la IOI. En esta ocasión nos preocupa la salida. En un conjunto de problemas del ICPC, estos se utilizan como ‘problemas de calentamiento’ o ‘para gastar tiempo’. Debes practicar tus capacidades de programación, resolviendo estos problemas *lo más rápidamente posible*, ya que la penalización de tiempo que tengas puede marcar la diferencia en la clasificación frente a otro equipo.

---

<sup>4</sup>También es posible resolver estos problemas sin utilizar expresiones regulares, pero el código será más largo.

- Comparación de cadenas

En estos problemas, los concursantes deben comparar cadenas según varios criterios. Esta subcategoría es similar a la de los problemas de coincidencia de cadenas que veremos en la siguiente sección, pero casi siempre se utilizarán funciones relacionadas con `strcmp`.

- Solo *ad hoc*

Otros problemas con cadenas *ad hoc*, que no caben en ninguna de las categorías anteriores.

## Ejercicios de programación

### Ejercicios de programación relacionados con el procesamiento de cadenas *ad hoc*:

#### Cifrar/codificar/decodificar/describir, fáciles

1. UVa 00245 - Uncompress  
(LA 5184 - WorldFinals Nashville95; usar el algoritmo dado)
2. UVa 00306 - Cipher  
(se puede acelerar evitando los ciclos)
3. UVa 00444 - Encoder and Decoder  
(cada carácter se vincula a 2 o 3 dígitos)
4. UVa 00458 - The Decoder  
(desplazar los valores ASCII por -7)
5. UVa 00483 - Word Scramble  
(leer carácter por carácter de izquierda a derecha)
6. UVa 00492 - Pig Latin  
(*ad hoc*; similar a UVa 483)
7. UVa 00641 - Do the Untwist  
(invertir la fórmula dada y simular)
8. UVa 00739 - Soundex Indexing  
(problema de conversión directo)
9. UVa 00795 - Sandorf's Cipher  
(preparar un 'mapeo inverso')
10. UVa 00865 - Substitution Cypher  
(sustitución de caracteres sencilla)
11. UVa 10019 - Funny Encryption Method  
(buscar el patrón)
12. UVa 10222 - Decode the Mad Man  
(mecanismo de decodificación sencillo)
13. **UVa 10851 - 2D Hieroglyphs ... \***  
(ignorar el límite; tratar '\/' como 1/0; leer desde abajo)
14. **UVa 10878 - Decode the Tape \***  
(tratar el espacio/o' como 0/1, después es una conversión de binario a decimal)  
(probar todas las claves posibles; trocear)
15. UVa 10896 - Known Plaintext Attack  
(problema de conversión sencilla)
16. UVa 10921 - Find the Telephone  
(seguir las instrucciones del problema)
17. UVa 11220 - Decoding the message  
(mapa de teclas QWERTY a DVORAK)
18. **UVa 11278 - One-Handed Typist \***  
(leer carácter por carácter y simular)
19. UVa 11541 - Decoding  
(cifrafo sencillo)
20. UVa 11716 - Digital Fortress  
(seguir el enunciado)
21. UVa 11787 - Numeral Hieroglyphs  
(*ad hoc*)
22. UVa 11946 - Code Number

#### Cifrar/codificar/decodificar/describir, difíciles

1. UVa 00213 - Message ...  
(LA 5152 - WorldFinals SanAntonio91; descifrar el mensaje)
2. UVa 00468 - Key to Success  
(mapa de frecuencia de letras)
3. **UVa 00554 - Caesar Cypher \***  
(probar todos los desplazamientos; formato de salida)
4. **UVa 00632 - Compression (II)**  
(simular el proceso; usar ordenación)
5. **UVa 00726 - Decode**  
(cifrado por frecuencia)
6. UVa 00740 - Baudot Data ...  
(basta simular el proceso)
7. UVa 00741 - Burrows Wheeler Decoder  
(simular el proceso)

- |                                          |                                                      |
|------------------------------------------|------------------------------------------------------|
| 8. UVa 00850 - Crypt Kicker II           | (ataque de texto plano; casos de prueba complicados) |
| 9. UVa 00856 - The Vigenère Cipher       | (3 bucles anidados, uno por cada dígito)             |
| 10. <u>UVa 11385 - Da Vinci Code</u> *   | (manipulación de cadenas y Fibonacci)                |
| 11. <u>UVa 11697 - Playfair Cipher</u> * | (seguir la descripción; un poco tedioso)             |

### Conteo de frecuencia

- |                                                |                                                                                     |
|------------------------------------------------|-------------------------------------------------------------------------------------|
| 1. UVa 00499 - What's The Frequency ...        | (usar <i>array</i> unidimensional para conteo de frecuencia)                        |
| 2. UVa 00895 - Word Problem                    | (obtener la frecuencia de letras de cada palabra; comparar con la línea del puzzle) |
| 3. <u>UVa 00902 - Password Search</u> *        | (leer carácter por carácter; contar frecuencia de palabras)                         |
| 4. UVa 10008 - What's Cryptanalysis?           | (contar frecuencia de caracteres)                                                   |
| 5. UVa 10062 - Tell me the frequencies         | (contar frecuencia de caracteres ASCII)                                             |
| 6. <u>UVa 10252 - Common Permutation</u> *     | (contar frecuencia de cada alfabeto)                                                |
| 7. UVa 10293 - Word Length and Frequency       | (directo)                                                                           |
| 8. UVa 10374 - Election                        | (usar <i>map</i> para conteo de frecuencias)                                        |
| 9. UVa 10420 - List of Conquests               | (conteo de frecuencia de palabras; usar <i>map</i> )                                |
| 10. UVa 10625 - GNU = GNU'sNotUnix             | (suma de frecuencia $n$ veces)                                                      |
| 11. UVa 10789 - Prime Frequency                | (comprobar si la frecuencia de una letra es un primo)                               |
| 12. <u>UVa 11203 - Can you decide it ...</u> * | (el enunciado parece complicado, pero el problema es sencillo)                      |
| 13. UVa 11577 - Letter Frequency               | (problema directo)                                                                  |

### Procesamiento de la entrada (no recursivo)

- |                                             |                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 00271 - Simply Syntax                | (comprobación gramatical; barrido lineal)                                                                                                                                                                                                                                                                                                                                     |
| 2. UVa 00327 - Evaluating Simple C ...      | (la implementación tiene su complejidad)                                                                                                                                                                                                                                                                                                                                      |
| 3. UVa 00391 - Mark-up                      | (usar etiquetas; procesamiento tedioso)                                                                                                                                                                                                                                                                                                                                       |
| 4. UVa 00397 - Equation Elation             | (realizar iterativamente la siguiente operación)                                                                                                                                                                                                                                                                                                                              |
| 5. UVa 00442 - Matrix Chain Multiplication  | (propiedades de multiplicación de cadenas de matrices)                                                                                                                                                                                                                                                                                                                        |
| 6. UVa 00486 - English-Number Translator    | (procesamiento)                                                                                                                                                                                                                                                                                                                                                               |
| 7. UVa 00537 - Artificial Intelligence?     | (fórmula sencilla; el procesamiento es difícil)                                                                                                                                                                                                                                                                                                                               |
| 8. UVa 01200 - A DP Problem                 | (LA 2972 - Tehran03; la parte difícil está en procesar y trocear la ecuación lineal)                                                                                                                                                                                                                                                                                          |
| 9. <u>UVa 10906 - Strange Integration</u> * | (procesamiento BNF; solución iterativa)                                                                                                                                                                                                                                                                                                                                       |
| 10. UVa 11148 - Moliu Fractions             | (extraer enteros; fracciones sencillas/mezcladas de una línea; un poco de GCD (sección 5.5.2))                                                                                                                                                                                                                                                                                |
| 11. <u>UVa 11357 - Ensuring Truth</u> *     | (el enunciado asusta un poco; problema de SAT (satisfacción); la presencia de gramática BNF nos hace pensar en un procesador descendente recursivo; sin embargo, basta con satisfacer una cláusula para obtener VERDADERO; una cláusula queda satisfecha si, para todas las variables de la misma, su inversa no es la propia cláusula; ahora el problema ya es más sencillo) |
| 12. <u>UVa 11878 - Homework Checker</u> *   | (procesamiento de expresiones matemáticas)                                                                                                                                                                                                                                                                                                                                    |
| 13. UVa 12543 - Longest Word                | (LA 6150 - HatYai12; procesador iterativo)                                                                                                                                                                                                                                                                                                                                    |

## Procesamiento de la entrada (recursivo)

- |                                            |                                                     |
|--------------------------------------------|-----------------------------------------------------|
| 1. UVa 00384 - Slurpys                     | (comprobación gramatical recursiva)                 |
| 2. UVa 00464 - Sentence/Phrase Generator   | (generar la salida en base a la gramática BNF dada) |
| 3. UVa 00620 - Cellular Structure          | (comprobación gramatical recursiva)                 |
| 4. <b>UVa 00622 - Grammar Evaluation *</b> | (comprobación/evaluación grammatical recursiva)     |
| 5. UVa 00743 - The MTM Machine             | (comprobación grammatical recursiva)                |
| 6. <b>UVa 10854 - Number of Paths *</b>    | (procesamiento recursivo y conteo)                  |
| 7. UVa 11070 - The Good Old Times          | (evaluación grammatical recursiva)                  |
| 8. <b>UVa 11291 - Smeech *</b>             | (comprobación grammatical recursiva)                |

Se resuelve con String/Pattern de Java (expresiones regulares)

- |                                               |                                                 |
|-----------------------------------------------|-------------------------------------------------|
| 1. <b>UVa 00325 - Identifying Legal ... *</b> | (ver la solución de Java anterior)              |
| 2. <b>UVa 00494 - Kindergarten ... *</b>      | (ver la solución de Java anterior)              |
| 3. UVa 00576 - Haiku Review                   | (procesamiento; gramática)                      |
| 4. <b>UVa 10058 - Jimmi's Riddles *</b>       | (se resuelve con expresiones regulares de Java) |

## Formato de salida

- |                                                  |                                                                                                  |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------|
| 1. UVa 00110 - Meta-loopless sort                | (en realidad, un problema de ordenación <i>ad hoc</i> )                                          |
| 2. <i>UVa 00159 - Word Crosses</i>               | (problema de formato de salida tedioso)                                                          |
| 3. UVa 00320 - Border                            | (requiere relleno por difusión)                                                                  |
| 4. <i>UVa 00330 - Inventory Maintenance</i>      | (usar map ayuda)                                                                                 |
| 5. <i>UVa 00338 - Long Multiplication</i>        | (tedioso)                                                                                        |
| 6. <i>UVa 00373 - Romulan Spelling</i>           | (comprobar 'g' frente a 'p'; <i>ad hoc</i> )                                                     |
| 7. <i>UVa 00426 - Fifth Bank of ...</i>          | (trocear; ordenar; formato de salida)                                                            |
| 8. UVa 00445 - Marvelous Mazes                   | (simulación; formato de salida)                                                                  |
| 9. <b>UVa 00488 - Triangle Wave *</b>            | (usar varios bucles)                                                                             |
| 10. UVa 00490 - Rotating Sentences               | (manipulación de array bidimensional; formato de salida)                                         |
| 11. <i>UVa 00570 - Stats</i>                     | (usar map ayuda)                                                                                 |
| 12. <i>UVa 00645 - File Mapping</i>              | (usar recursión para simular la estructura de directorios, ya que ayuda en el formato de salida) |
| 13. <i>UVa 00890 - Maze (II)</i>                 | (simulación, seguir los pasos; tedioso)                                                          |
| 14. UVa 01219 - Team Arrangement                 | (LA 3791 - Tehran06)                                                                             |
| 15. <i>UVa 10333 - The Tower of ASCII</i>        | (una auténtica pérdida de tiempo)                                                                |
| 16. UVa 10500 - Robot maps                       | (simular; formato de salida)                                                                     |
| 17. UVa 10761 - Broken Keyboard                  | (formato de salida complicado; 'END' es parte de la entrada)                                     |
| 18. <b>UVa 10800 - Not That Kind of Graph *</b>  | (problema tedioso)                                                                               |
| 19. <i>UVa 10875 - Big Math</i>                  | (problema sencillo pero tedioso)                                                                 |
| 20. UVa 10894 - Save Hridoy                      | (¿cuánto tardas en resolver este problema?)                                                      |
| 21. UVa 11074 - Draw Grid                        | (formato de salida)                                                                              |
| 22. <i>UVa 11482 - Building a Triangular ...</i> | (tedioso)                                                                                        |
| 23. UVa 11965 - Extra Spaces                     | (sustituir espacios consecutivos con uno solo)                                                   |
| 24. <b>UVa 12155 - ASCII Diamondi *</b>          | (usar manipulación de índices adecuada)                                                          |
| 25. <i>UVa 12364 - In Braille</i>                | (comprobación de array bidimensional; comprobar todos los dígitos posibles [0 . . 9])            |

## Comparación de cadenas

- |                                                  |                                                                  |
|--------------------------------------------------|------------------------------------------------------------------|
| 1. UVa 00409 - Excuses, Excuses                  | (trocear y comparar con la lista de excusas)                     |
| 2. <b>UVa 00644 - Immediate Decodability *</b>   | (usar fuerza bruta)                                              |
| 3. UVa 00671 - Spell Checker                     | (comparación de cadenas)                                         |
| 4. UVa 00912 - Live From Mars                    | (simulación; buscar y sustituir)                                 |
| 5. <b>UVa 11048 - Automatic Correction ... *</b> | (comparación de cadenas flexible en relación a un diccionario)   |
| 6. <b>UVa 11056 - Formula 1 *</b>                | (ordenación; comparación independiente de mayúsculas/minúsculas) |
| 7. UVa 11233 - Deli Deli                         | (comparación de cadenas)                                         |
| 8. UVa 11713 - Abstract Names                    | (comparación de cadenas modificada)                              |
| 9. UVa 11734 - Big Number of ...                 | (comparación de cadenas modificada)                              |

## Solo *ad hoc*

- |                                                 |                                                                        |
|-------------------------------------------------|------------------------------------------------------------------------|
| 1. UVa 00153 - Permalex                         | (encontrar la fórmula; similar a UVa 941)                              |
| 2. UVa 00263 - Number Chains                    | (ordenar dígitos; convertir a enteros; comprobar ciclos)               |
| 3. UVa 00892 - Finding words                    | (problema de procesamiento de cadenas básico)                          |
| 4. <b>UVa 00941 - Permutations *</b>            | (fórmula para obtener la permutación $n$ -ésima)<br>(LA 3669; Hanoi06) |
| 5. UVa 01215 - String Cutting                   | (LA 4144; Jakarta08; fuerza bruta)                                     |
| 6. UVa 01239 - Greatest K-Palindrome ...        | (ordenar las palabras para simplificar el problema)                    |
| 7. UVa 10115 - Automatic Editing                | (seguir el enunciado con mucha atención)                               |
| 8. UVa 10126 - Zipf's Law                       | (leer; trocear; procesar como se pide)                                 |
| 9. UVa 10197 - Learning Portuguese              | (parece más un problema de estructuras de datos)                       |
| 10. UVa 10361 - Automatic Poetry                | (seguir el enunciado de problema)                                      |
| 11. UVa 10391 - Compound Words                  | (número de palabras = número de letras + 1)                            |
| 12. <b>UVa 10393 - The One-Handed Typist *</b>  | (mal caso de prueba; si $T$ es un prefijo de $S$ , obtiene AC)         |
| 13. UVa 10508 - Word Morphing                   | (periodo de cadenas; entrada pequeña; fuerza bruta)                    |
| 14. UVa 10679 - I Love Strings                  | (directo; usar el 'carácter de escapado')                              |
| 15. <b>UVa 11452 - Dancing the Cheeky ... *</b> | (illegal cuando la marca es 0 o $> 1$ )                                |
| 16. UVa 11483 - Code Creator                    | (encontrar la fórmula; similar a UVa 941; base 4)                      |
| 17. UVa 11839 - Optical Reader                  | (problema de troceo de cadenas sencillo)                               |
| 18. UVa 11962 - DNA II                          | (fuerza bruta con cadenas)                                             |
| 19. UVa 12243 - Flowers Flourish ...            |                                                                        |
| 20. UVa 12414 - Calculating Yuan Fen            |                                                                        |

## 6.4 Coincidencia de cadenas

La *coincidencia* de cadenas (también llamada *búsqueda*<sup>5</sup>), consiste encontrar el índice, o índices, de una (sub)cadena (llamada *patrón P*) dentro de una cadena más larga (llamada *texto T*). Por ejemplo, asumamos que tenemos  $T = \text{'STEVEN EVENT'}$ . Si  $P = \text{'EVE'}$ , la respuesta serán los índices 2 y 7 (contando desde 0). Si  $P = \text{'EVENT'}$ , la respuesta será solo 8. Si  $P = \text{'EVENING'}$ , no habrá respuesta (no se encontrará ninguna coincidencia y devolveremos -1 o NULL).

<sup>5</sup>Nos encontramos ante un problema de coincidencia de cadenas prácticamente cada vez que leemos o escribimos texto utilizando un ordenador. ¿Cuántas veces has utilizado la combinación de teclas 'CTRL + F' (el acceso directo habitual de Windows para realizar una búsqueda) en procesadores de textos, navegadores, etc.?

## 6.4.1 Soluciones con bibliotecas

En la mayoría de los problemas de coincidencia de cadenas *puros*, con cadenas razonablemente cortas, podemos utilizar la biblioteca de cadenas de nuestro lenguaje de programación. Esta es `strstr` en `<string.h>` de C, `find` en `<string>` de C++ o `indexOf` en la clase `String` de Java. Puedes repasar estas bibliotecas en la tarea 2 de la sección 6.2.

## 6.4.2 Algoritmo de Knuth-Morris-Pratt (KMP)

En la pregunta 7 de la sección 1.2.3, tenemos un ejercicio que plantea encontrar todas las apariciones de la subcadena  $P$  (de longitud  $m$ ) en una cadena (larga)  $T$  (de longitud  $n$ ), si hay alguna. El código que incluimos a continuación, es una implementación *ingenua* de un algoritmo de coincidencia de cadenas:

```
1 void naiveMatching() {
2 for (int i = 0; i < n; i++) { // probar los índices de inicio potenciales
3 bool found = true;
4 for (int j = 0; j < m && found; j++) // usar etiqueta booleana 'found'
5 if (i + j >= n || P[j] != T[i + j]) // si es no coincidencia
6 found = false; // abortar, desplazar +1 el índice de inicio i
7 if (found) // si P[0..m-1] == T[i..i+m-1]
8 printf("P is found at index %d in T\n", i);
9 }
```

Este algoritmo ingenuo se puede ejecutar, *de media*, en  $O(n)$  si se aplica a un texto natural, como el de los párrafos de este libro, pero puede llegar a ejecutarse en  $O(nm)$  con un caso extremo, de los que podemos encontrar en un concurso de programación, como  $T = \text{'AAAAAAAAAAAB'}$  (diez veces ‘A’ seguido de una ‘B’) y  $P = \text{'AAAAB'}$ . El algoritmo ingenuo fallará una y otra vez con el último carácter del patrón  $P$  y volverá a empezar en el siguiente índice, que corresponderá al anterior más uno. No resulta eficiente. Por desgracia para nosotros, un buen autor de problemas incluirá un caso así entre sus casos de prueba.

En 1977, Knuth, Morris y Pratt, de ahí el nombre de KMP, inventaron un algoritmo de coincidencia de cadenas mejor, que utiliza la información obtenida en las comparaciones anteriores, especialmente en las que se ha encontrado una coincidencia. El algoritmo KMP *nunca* vuelve a comparar un carácter de  $T$  que haya coincidido con un carácter de  $P$ . Sin embargo, el funcionamiento es similar al algoritmo ingenuo, si el *primer* carácter del patrón  $P$  y el carácter actual de  $T$  no son iguales. En el siguiente ejemplo<sup>6</sup>, la comparación de  $P[j]$  y  $T[i]$  con  $i = 0$  a 13 y con  $j = 0$  (el primer carácter de  $P$ ), no es diferente de la del algoritmo ingenuo.

|                                                         |   |   |   |   |  |
|---------------------------------------------------------|---|---|---|---|--|
| 1                                                       | 2 | 3 | 4 | 5 |  |
| 012345678901234567890123456789012345678901234567890     |   |   |   |   |  |
| T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN |   |   |   |   |  |
| P = SEVENTY SEVEN                                       |   |   |   |   |  |
| 0123456789012                                           |   |   |   |   |  |
| 1                                                       |   |   |   |   |  |

<sup>6</sup>La frase de la cadena  $T$  que mostramos tiene efectos ilustrativos, aunque no sea correcta gramaticalmente.

```

^ el primer carácter de P no coincide con T[i] desde índice i=0 a 13
KMP debe desplazar +1 el índice inicial i, como con la versión ingenua
... en i = 14 y j = 0 ...
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =
 SEVENTY SEVEN
 0123456789012
 1
 ^ sin coincidencia en índices i = 25 y j = 11

```

Hay 11 coincidencias entre los índices  $i = 14$  hasta  $24$ , pero una no coincidencia en  $i = 25$  ( $j = 11$ ). El algoritmo de coincidencia ingenuo volverá a comenzar, de forma poco eficiente, en  $i = 15$ , pero KMP puede continuar desde  $i = 25$ . Esto es debido a que los caracteres coincidentes antes de la no coincidencia son ‘SEVENTY SEV’. ‘SEV’ (de longitud 3) aparece TANTO como sufijo y como prefijo de ‘SEVENTY SEV’. Este ‘SEV’ también es el **límite** de ‘SEVENTY SEV’. Podemos saltar con seguridad los índices desde  $i = 14$  hasta  $21$ : ‘SEVENTY’ en ‘SEVENTY SEV’, ya que no los volveremos a encontrar, pero no podemos descartar que la siguiente coincidencia comience en el segundo ‘SEV’. Así que KMP establece de nuevo  $j$  a 3, saltando  $11 - 3 = 8$  caracteres de ‘SEVENTY’ (cuidado con el espacio final), mientras que  $i$  sigue en 25. Esta es la principal diferencia entre los algoritmos de coincidencia KMP e ingenuo.

```

... en i = 25 y j = 3 (esto hace eficiente a KMP) ...
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =
 SEVENTY SEVEN
 0123456789012
 1
 ^ no coincidencia inmediata en i = 25, j = 3

```

Esta vez el prefijo de P, antes de la no coincidencia, es ‘SEV’, pero no tiene un límite, así que KMP establece  $j$  a 0 (o, en otras palabras, reinicia el patrón P desde el principio).

```

... no coincide de i = 25 a i = 29... después coincide de i = 30 a i = 42 ...
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =
 SEVENTY SEVEN
 0123456789012
 1

```

Aquí hay una coincidencia, de forma que  $P = ‘SEVENTY SEVEN’$  aparece en el índice  $i = 30$ . Después de esto, KMP sabe que ‘SEVENTY SEVEN’ tiene ‘SEVEN’ (de longitud 5) como límite, por lo que retoma  $j$  desde 5, saltando, por tanto,  $13 - 5 = 8$  caracteres de ‘SEVENTY’ (cuidado con el espacio final) y retomando inmediatamente la búsqueda desde  $i = 43$ , encontrando otra coincidencia. Es una implementación eficiente.

```

... en i = 43 y j = 5, coincidencias desde i = 43 a i = 50 ...
Así P = 'SEVENTY SEVEN' aparece de nuevo en el índice i = 38.
 1 2 3 4 5
012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
 0123456789012
 1

```

Para lograr esa mejora de velocidad, KMP debe procesar previamente la cadena patrón y obtener la ‘tabla de reinicios’  $b$  (hacia atrás). Si el patrón de búsqueda  $P = 'SEVENTY SEVEN'$ , la tabla  $b$  tendrá este aspecto:

```

1
0 1 2 3 4 5 6 7 8 9 0 1 2 3
P = S E V E N T Y S E V E N
b = -1 0 0 0 0 0 0 0 1 2 3 4 5

```

Esto significa que, si ocurre una no coincidencia en  $j = 11$  (ver el ejemplo anterior), es decir, después de encontrar coincidencias para ‘SEVENTY SEV’, sabremos que debemos reintentar la coincidencia de  $P$  desde el índice  $j = b[11] = 3$ . KMP asumirá entonces que únicamente ha localizado los tres primeros caracteres de ‘SEVENTY SEV’, que es ‘SEV’, porque la próxima coincidencia puede comenzar con el prefijo ‘SEV’. Incluimos, a continuación, una implementación comentada y relativamente corta del algoritmo de KMP. Aquí la complejidad es de  $O(n + m)$ :

```

1 #define MAX_N 100010
2 char T[MAX_N], P[MAX_N]; // T = texto, P = patrón
3 int b[MAX_N], n, m; // b = tabla inversa, n = long. de T, m = long. de P
4
5 void kmpPreprocess() { // llamar antes de kmpSearch()
6 int i = 0, j = -1; b[0] = -1; // valores iniciales
7 while (i < m) { // proceso previo de cadena patrón P
8 while (j >= 0 && P[i] != P[j]) j = b[j]; // diferente, reiniciar j con b
9 i++; j++; // si es igual, avanzar ambos punteros
10 b[i] = j; // observar i = 8, 9, 10, 11, 12, 13 con j = 0, 1, 2, 3, 4, 5
11 } } // en el ejemplo anterior de P = "SEVENTY SEVEN"
12
13 void kmpSearch() { // similar a kmpPreprocess(), pero sobre la cadena T
14 int i = 0, j = 0; // valores iniciales
15 while (i < n) { // buscar en la cadena T
16 while (j >= 0 && T[i] != P[j]) j = b[j]; // diferente, reiniciar j con b
17 i++; j++; // si es igual, avanzar ambos punteros
18 if (j == m) { // se encuentra coincidencia cuando j == m
19 printf("P is found at index %d in T\n", i - j);
20 j = b[j]; // preparar j para la siguiente posible coincidencia
21 } } }

```



ch6\_02\_kmp.cpp



ch6\_02\_kmp.java

### Ejercicio 6.4.1\*

Ejecuta `kmpPreprocess()` con  $P = 'ABABA'$  y muestra la tabla de reincios  $b$ .

### Ejercicio 6.4.2\*

Ejecuta `kmpSearch()` con  $P = 'ABABA'$  y  $T = 'ACABAABABDABABA'$ .

### 6.4.3 Coincidencia de cadenas en una rejilla bidimensional

El problema de procesamiento de cadenas también se puede plantear en dos dimensiones. Dada una rejilla/*array* bidimensional (en vez del ya conocido *array* de una dimensión), encontrar las apariciones del patrón  $P$  en la rejilla. Dependiendo de los requisitos del problema, el sentido de la búsqueda puede ser de 4 u 8 direcciones cardinales, y el patrón debe aparecer en línea recta o puede tener giros. Veamos el siguiente ejemplo:

```
abcdefhigg // De UVa 10010 - Where's Waldorf?
hebkWaldork // Podemos ir en 8 direcciones, pero en línea recta
ftyawAldorm // 'WALDORF' aparece en mayúsculas en la rejilla
ftsimrLqsrc
byoarbeDeyv // ¿Puedes encontrar 'BAMBI' y 'BETTY'?
klcbqwik0mk
strgbgdhRb // ¿Encuentras 'DAGBERT' en esta fila?
yuiqlxcnbjF
```

La solución para este problema de coincidencia de cadenas en un rejilla bidimensional pasa normalmente por el *backtracking recursivo* (ver la sección 3.2.2). Esto es debido a que, a diferencia de su compañera unidimensional, donde siempre nos desplazamos a la derecha, en cada coordenada (fila, columna) de la rejilla bidimensional tenemos *más de una opción* a explorar.

Para acelerar el proceso de *backtracking*, normalmente debemos utilizar esta sencilla estrategia de poda: una vez que la profundidad de la búsqueda recursiva supera la longitud del patrón  $P$ , podemos podar, inmediatamente, esa rama. Esto se llama también *búsqueda profunda limitada* (ver la sección 8.2.5).

## Ejercicios de programación

### Ejercicios de programación relacionados con coincidencia de cadenas

#### Estándar

1. UVa 00455 - Periodic String
  2. UVa 00886 - *Named Extension Dialing*
  
  3. **UVa 10298 - Power Strings \***
  4. UVa 11362 - Phone List
  5. **UVa 11475 - Extend to Palindromes \***
  6. **UVa 11576 - Scrolling Sign \***
  7. UVa 11888 - Abnormal 89's
  8. UVa 12467 - *Secret word*
- (encontrar  $s$  en  $s + s$ )  
(convertir la primera letra del nombre dado y todas las del apellido en dígitos; después realizar una coincidencia de cadenas especial donde queremos que la coincidencia se inicie en el *prefijo* de una cadena)  
(encontrar  $s$  en  $s + s$ ; similar a UVa 455)  
(ordenación de cadenas; coincidencia)  
('límite' de KMP)  
(coincidencia de cadenas modificada; búsqueda completa)  
(para comprobar 'alíndromo', encontrar la inversa de  $s$  en  $s + s$ )  
('límite' de KMP; ver también UVa 11475)

#### En rejilla bidimensional

1. **UVa 00422 - Word Search Wonder \***
  2. UVa 00604 - *The Boggle Game*
  3. UVa 00736 - *Lost in Space*
  4. **UVa 10010 - Where's Waldorf? \***
  5. **UVa 11283 - Playing Boggle \***
- (rejilla bidimensional; *backtracking*)  
(rejilla bidimensional; *backtracking*; ordenar y comparar)  
(rejilla bidimensional; un poco modificada)  
(rejilla bidimensional; *backtracking*)  
(rejilla bidimensional; *backtracking*; no contar dos veces)

## 6.5 Procesamiento de cadenas con programación dinámica

En esta sección, trataremos varios problemas de procesamiento de cadenas que se pueden resolver mediante la técnica de programación dinámica, vista en la sección 3.5. Los dos primeros (alineación de cadenas y subsecuencia común más larga), son problemas *clásicos* que deben ser conocidos por todos los participantes en concursos de programación. Además, hemos añadido un compendio de algunas de las variantes conocidas de estos problemas.

Es importante poner de relevancia que, en varios problemas de DP con cadenas, normalmente manipulamos los *índices enteros* de las cadenas y no las propias cadenas (o subcadenas). No recomendamos en absoluto utilizar subcadenas como parámetros de funciones recursivas, ya que es muy lento y consume mucha memoria.

### 6.5.1 Alineación de cadenas (distancia de edición)

El problema de alineación de cadenas (o distancia de edición<sup>7</sup>) se define de la siguiente manera: la alineación de dos cadenas A y B, con la mayor puntuación de alineación (o el menor número de operaciones de edición).

<sup>7</sup>También llamada 'distancia de Levenshtein'. Una aplicación notable es la utilidad de corrección ortográfica de los editores de texto. Si el usuario escribe incorrectamente una palabra, como 'probelma', se detectará que esta tiene una distancia de edición muy cercana a la correcta 'problema' y podrá corregirla automáticamente.

Después de alinear<sup>8</sup> A con B, hay algunas posibilidades entre los caracteres A[i] y B[i]:

1. Los caracteres A[i] y B[i] **coinciden** y no hacemos nada (puntúa '+2').
2. Los caracteres A[i] y B[i] **no coinciden** y sustituimos A[i] con B[i] (puntúa '-1').
3. Insertamos un espacio en A[i] (también puntúa '-1').
4. Eliminamos una letra de A[i] (y también puntúa '-1').

Por ejemplo (utilizamos el símbolo '\_' para indicar un espacio):

```
A = 'ACAATCC' -> 'A_CAAATCC' // Ejemplo de alineación no óptima
B = 'AGCATGC' -> 'AGCAGTC_'
2-22--2- // Comprobar la óptima a continuación
 // Puntuación de alineación = 4*2 + 4*-1 = 4
```

Una solución de fuerza bruta, que pruebe todas las alineaciones posibles, tendrá un veredicto de TLE, incluso para cadenas A y/o B de longitud mediana. La solución para este problema es el algoritmo de DP de Needleman-Wunsch (de abajo a arriba) [62]. Consideremos dos cadenas A[1..n] y B[1..m]. Definimos  $V(i,j)$  como la puntuación de la alineación óptima, para los prefijos A[1..i] y B[1..j] y  $score(C1, C2)$  como la función que devuelve la puntuación, si el carácter  $C1$  está alineado con el carácter  $C2$ .

Casos base:

- $V(0,0) = 0$   
Dos cadenas vacías no puntúan.
- $V(i,0) = i \times score(A[i], \_)$   
Eliminar la subcadena A[1..i] para realizar la alineación,  $i > 0$ .
- $V(0,j) = j \times score(\_, B[j])$   
Insertar espacios en B[1..j] para realizar la alineación,  $j > 0$ .

Recurrencias: Para  $i > 0$  y  $j > 0$ :

- $V(i,j) = max(opción1, opción2, opción3)$ , donde
  - $opción1 = V(i-1, j-1) + score(A[i], B[j])$   
Puntuación de coincidencia o no coincidencia.
  - $opción2 = V(i-1, j) + score(A[i], \_)$   
Eliminar  $A_i$ .
  - $opción3 = V(i, j-1) + score(\_, B[j])$   
Insertar  $B_j$ .

En resumen, este algoritmo de DP se concentra en las tres posibilidades del último par de caracteres, que deben ser una coincidencia/no coincidencia, una eliminación o una inserción. Aunque no sabemos cuál de ellos es el mejor, podemos probar todas las posibilidades, evitando recalcular los subproblemas superpuestos (es decir, básicamente una técnica de DP).

<sup>8</sup>Proceso por el que se insertan espacios en las cadenas A y B de forma que tengan el mismo número de caracteres. 'Insertar espacios en B' puede entenderse como 'eliminar caracteres alineados correspondientes de A'.

|                  |                  |                  |
|------------------|------------------|------------------|
| $A = 'xxx...xx'$ | $A = 'xxx...xx'$ | $A = 'xxx...x_'$ |
|                  |                  |                  |
| $B = 'yyy...yy'$ | $B = 'yyy...y_'$ | $B = 'yyy...yy'$ |
| no/coincidencia  | eliminar         | insertar         |

|   | _  | A          | G  | C  | A  | T  | G  | C  |
|---|----|------------|----|----|----|----|----|----|
| _ | 0  | -1         | -2 | -3 | -4 | -5 | -6 | -7 |
| A | -1 |            |    |    |    |    |    |    |
| C | -2 |            |    |    |    |    |    |    |
|   |    | Casos base |    |    |    |    |    |    |
| A | -3 |            |    |    |    |    |    |    |
| A | -4 |            |    |    |    |    |    |    |
| T | -5 |            |    |    |    |    |    |    |
| C | -6 |            |    |    |    |    |    |    |
| C | -7 |            |    |    |    |    |    |    |

|   | _  | A  | G  | C  | A  | T  | G  | C  |
|---|----|----|----|----|----|----|----|----|
| _ | 0  | -1 | -2 | -3 | -4 | -5 | -6 | -7 |
| A | -1 | 2  | 1  | 0  | -1 | -2 | -3 | -4 |
| C | -2 | 1  | 1  | 3  |    |    |    |    |
|   |    |    |    |    |    |    |    |    |
| A | -3 |    |    |    |    |    |    |    |
| A | -4 |    |    |    |    |    |    |    |
| T | -5 |    |    |    |    |    |    |    |
| C | -6 |    |    |    |    |    |    |    |
| C | -7 |    |    |    |    |    |    |    |

|   | _  | A  | G  | C  | A  | T  | G  | C  |
|---|----|----|----|----|----|----|----|----|
| _ | 0  | -1 | -2 | -3 | -4 | -5 | -6 | -7 |
| A | -1 | 2  | 1  | 0  | -1 | -2 | -3 | -4 |
| C | -2 | 1  | 1  | 3  | 2  | 1  | 0  | -1 |
| A | -3 | 0  | 0  | 2  | 5  | 4  | 3  | 2  |
| A | -4 | -1 | 1  | 1  | 4  | 3  | 2  |    |
| T | -5 | -2 | 0  | 3  | 6  | 5  | 4  |    |
| C | -6 | -3 | -3 | 0  | 2  | 5  | 5  | 7  |
| C | -7 | -4 | -4 | -1 | 1  | 4  | 4  | 7  |

Figura 6.1: Ejemplo:  $A = 'ACAATCC'$  y  $B = 'AGCATGC'$  (puntuación de la alineación = 7)

Con una función de puntuación sencilla, donde una coincidencia obtiene +2 puntos y una no coincidencia, una inserción o una eliminación obtienen -1 punto, mostramos el detalle de la puntuación de alineación de  $A = 'ACAATCC'$  y  $B = 'AGCATGC'$  en la figura 6.1. Inicialmente, solo se conocen los casos base. Después, podemos llenar los valores fila por fila, de izquierda a derecha. Para llenar  $V(i, j)$  para  $i, j > 0$ , únicamente necesitamos otros tres valores:  $V(i - 1, j - 1)$ ,  $V(i - 1, j)$  y  $V(i, j - 1)$  (ver la figura 6.1, centro, fila 2, columna 3). La puntuación de alineación máxima se almacena en la celda inferior derecha (7 en el presente ejemplo).

Para reconstruir la solución, seguimos las celdas más oscuras desde la inferior derecha. La solución para las cadenas dadas A y B se muestra a continuación. Una flecha diagonal significa una coincidencia o una no coincidencia (por ejemplo, el último carácter ..C). Una flecha vertical significa una eliminación (por ejemplo, ..CA.. a ..C\_A..). Una flecha horizontal significa una inserción (por ejemplo, A\_C.. a AGC..).

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| $A = 'A\_CAAT[C]C'$ | <i>// Alineación óptima</i>                         |
| $B = 'AGC\_AT[G]C'$ | <i>// Puntuación de alineación = 5*2 + 3*-1 = 7</i> |

La complejidad espacial de este algoritmo de DP (de abajo a arriba) es  $O(nm)$ , el tamaño de la tabla de DP. Debemos llenar todas las celdas de la tabla en  $O(1)$  por celda. Por lo tanto, la complejidad en tiempo es  $O(nm)$ .



ch6\_03\_str\_align.cpp



ch6\_03\_str\_align.java

### Ejercicio 6.5.1.1

¿Por qué el coste de una coincidencia es +2 y el coste de sustituir, insertar o eliminar es -1? ¿Existen números mágicos? ¿Funcionaría +1 para las coincidencias? ¿Pueden ser los costes de sustituir, insertar o eliminar diferentes? Vuelve a analizar el algoritmo.

### Ejercicio 6.5.1.2

El código fuente de ejemplo `ch6_03_str_align.cpp/java` solo muestra la *puntuación* de la alineación óptima. Modifica el código para mostrar la *alineación real*.

### Ejercicio 6.5.1.3

Muestra cómo utilizar el ‘truco de ahorro de espacio’ de la sección 3.5, para mejorar este algoritmo de DP (de abajo a arriba) de Needleman-Wunsch. ¿Cuáles serán las nuevas complejidades de espacio y tiempo? ¿Hay inconveniente en utilizar una formulación así?

### Ejercicio 6.5.1.4

El problema de la alineación de cadenas de esta sección, se llama problema de alineación **global**, y se ejecuta en  $O(nm)$ . Si el problema concreto del concurso tiene un límite de  $d$  inserciones y eliminaciones, podemos utilizar un algoritmo más veloz. Encuentra una sencilla modificación del algoritmo de Needleman-Wunsch, para que realice un máximo de  $d$  inserciones o eliminaciones y se ejecute más rápido.

### Ejercicio 6.5.1.5

Investiga la mejora del algoritmo de Needleman-Wunsch (el algoritmo de **Smith-Waterman** [62]), para resolver el problema de alineación **local**.

## 6.5.2 Subsecuencia común más larga

El problema de la subsecuencia común más larga (LCS), se define de la siguiente manera: dadas dos cadenas A y B, determinar cuál es la subsecuencia común más larga entre ellas. Por ejemplo, A = ‘ACAATCC’ y B = ‘AGCATGC’, tienen una LCS de longitud 5, es decir, ‘ACATC’.

Este problema de la LCS se puede reducir al problema de alineación de cadenas presentado antes, por lo que podemos utilizar el mismo algoritmo de DP. Establecemos el coste de una no coincidencia como infinito negativo (por ejemplo, -1.000.000.000), el coste de la inserción y la eliminación en 0 y el coste de una coincidencia como 1. Esto provoca que el algoritmo de Needleman-Wunsch, para alineación de cadenas, nunca considere las no coincidencias.

### Ejercicio 6.5.2.1

¿Cuál es la LCS de A = ‘apple’ y B = ‘people’?

### Ejercicio 6.5.2.2

El problema de la distancia de Hamming, es decir, la búsqueda del número de caracteres diferentes entre dos cadenas de igual longitud, se puede reducir a un problema de alineación de cadenas. Asigna un coste adecuado a la coincidencia, no coincidencia, inserción y eliminación, de forma que podamos calcular la distancia de Hamming entre dos cadenas, utilizando el algoritmo de Needleman-Wunsch.

### Ejercicio 6.5.2.3

El problema de la LCS se puede resolver en  $O(n \log k)$ , cuando todos los caracteres son distintos. Por ejemplo, si recibes dos permutaciones, como en el problema UVa 10635. Resuelve esta variante.

## 6.5.3 Procesamiento de cadenas no clásico con DP

### UVa 11151 - Longest Palindrome

Un palíndromo es una cadena que se puede leer igualmente en cualquier dirección. Algunas variantes de problemas de palíndromos se pueden resolver mediante programación dinámica, por ejemplo, el UVa 11151 - Longest Palindrome: dada una cadena de hasta  $n = 1000$  caracteres, determinar la longitud del palíndromo más largo que se pueda conseguir, eliminando cero o más caracteres. Ejemplos:

- ‘ADAM’ → ‘ADA’ (de longitud 3, se elimina la ‘M’)
- ‘MADAM’ → ‘MADAM’ (de longitud 5, no se elimina nada)
- ‘NEVERODDOREVENING’ → ‘NEVERODDOREVEN’ (de longitud 14, se elimina ‘ING’)
- ‘RACEF1CARFAST’ → ‘RACECAR’ (de longitud 7, se elimina ‘F1’ y ‘FAST’)

Solución de DP: digamos que  $\text{len}(l, r)$  es la longitud del palíndromo más largo de  $A[1..r]$ .

Casos base:

- Si  $(l = r)$ , entonces  $\text{len}(l, r) = 1$ . Palíndromo de longitud impar.
- Si  $(l + 1 = r)$ , entonces  $\text{len}(l, r) = 2$  si  $(A[l] = A[r])$ , o 1 en caso contrario. Longitud par.

Recurrencias:

- Si  $(A[l] = A[r])$ , entonces  $\text{len}(l, r) = 2 + \text{len}(l + 1, r - 1)$ . Los dos extremos son iguales.
- En otro caso  $\text{len}(l, r) = \max(\text{len}(l, r - 1), \text{len}(l + 1, r))$ . Aumentamos el lado izquierdo o reducimos el derecho.

Esta solución de programación dinámica tiene una complejidad de tiempo de  $O(n^2)$ .

### Ejercicio 6.5.3.1\*

¿Podemos utilizar la solución de la subsecuencia común más larga, vista en la sección 6.5.2, para resolver el problema UVa 11151? ¿Si es posible, cómo? ¿Cuál será la complejidad de tiempo?

### Ejercicio 6.5.3.2\*

Supón que estás interesado en encontrar el palíndromo más largo en una cadena dada, de longitud hasta  $n = 10000$  caracteres. Esta vez no es posible eliminar ningún carácter. ¿Cuál sería la solución?

## Ejercicios de programación

### Ejercicios de programación relacionados con procesamiento de cadenas con DP:

#### Clásicos

1. UVa 00164 - String Computer
2. **UVa 00526 - String Distance ... \***  
(alineación de cadenas/distancia de edición)
3. UVa 00531 - Compromise  
(subsecuencia común más larga; mostrar la solución)
4. UVa 01207 - AGTC  
(LA 3170 - Manila06; problema de edición de cadenas clásico)
5. UVa 10066 - The Twin Towers  
(problema de subsecuencia común más larga; pero no en 'cadenas')
6. UVa 10100 - Longest Match  
(subsecuencia común más larga)
7. **UVa 10192 - Vacation \***  
(subsecuencia común más larga)
8. UVa 10405 - Longest Common ...  
(problema muy conocido de subsecuencia común más larga)
9. **UVa 10635 - Prince and Princess \***  
(encontrar la LCS de dos permutaciones)
10. UVa 10739 - String to Palindrome  
(variación de distancia de edición)

#### No clásicos

1. **UVa 00257 - Palinwords**  
(palíndromo DP estándar y comprobaciones por fuerza bruta)
2. **UVa 10456 - Make Palindrome**  
(s: (L,R); t: (L+1, R-1) si  $S[L] == S[R]$ ; o 1 más el mínimo de (L + 1, R) o (L, R - 1); mostrar la solución)
3. UVa 10617 - Again Palindrome  
(manipulación de índices, no de la cadena)
4. **UVa 11022 - String Factoring \***  
(s: el peso mínimo de la subcadena  $[i \dots j]$ )
5. **UVa 11151 - Longest Palindrome \***  
(tratado en esta sección)
6. **UVa 11258 - String Partition \***  
(tratado en esta sección)
7. **UVa 11552 - Fewest Flops**  
(dp(i, c) = número mínimo de trozos después de considerar los primeros  $i$  segmentos terminados con el carácter c)

## Perfiles de los inventores de algoritmos

**Udi Manber** es un científico de la computación israelí. Es uno de los vicepresidentes de ingeniería de Google. Junto a Gene Myers, Manber inventó la estructura de datos de *array* de sufijos en 1991.

**Eugene “Gene” Wimberly Myers, Jr.** es un científico de la computación y bioinformático estadounidense, principalmente conocido por su desarrollo de la herramienta BLAST (*Basic Local Alignment Search Tool*), para el análisis de secuencias. Su artículo de 1990, en el que describe BLAST, ha sido citado más de 24000 veces, siendo uno de los más mencionados de la historia. También inventó el *array* de sufijos, junto a Udi Manber.

## 6.6 Trie/Árbol/Array de sufijos

Los *tries*, árboles y *arrays* de sufijos, son estructuras de datos eficientes y adecuadas para las cadenas. No hemos tratado este tema en la sección 2.4, ya que son estructuras de datos específicas para cadenas.

### 6.6.1 Trie de sufijos y aplicaciones

El **sufijo  $i$**  (o sufijo  $i$ -ésimo) de una cadena, es un ‘caso especial’ de subcadena, formado por la secuencia que va desde el carácter  $i$ -ésimo de la cadena hasta el *último* carácter de la misma. Por ejemplo, el sufijo 2-ésimo de ‘STEVEN’ es ‘EVEN’ y el 4-ésimo es ‘EN’ (con indexación desde 0).

El **trie de sufijos**<sup>9</sup> de un conjunto de cadenas  $S$ , es un árbol de todos los posibles sufijos de las cadenas contenidas en  $S$ . Cada etiqueta de una arista representa un carácter. Cada vértice representa un sufijo indicado por la etiqueta de su ruta: una secuencia de etiquetas de aristas desde la raíz hasta ese vértice. Cada vértice está conectado a (algunos de) los otros 26 vértices (asumiendo que utilicemos el alfabeto latino internacional en mayúsculas), según los sufijos de las cadenas en  $S$ . El prefijo común de dos sufijos se comparte. Cada vértice tiene dos etiquetas booleanas, que indican que en  $S$  existe un sufijo o palabra, respectivamente, que *termina* en ese vértice. Por ejemplo: si tenemos que  $S = \{\text{CAR}, \text{CAT}, \text{RAT}\}$ , tendremos los sufijos  $\{\text{CAR}, \text{AR}, \text{R}, \text{CAT}, \text{AT}, \text{T}, \text{RAT}, \text{AT}, \text{T}\}$ . Después de ordenarlos y eliminar los duplicados, obtendremos  $\{\text{AR}, \text{AT}, \text{CAR}, \text{CAT}, \text{R}, \text{RAT}, \text{T}\}$ . La figura 6.2 muestra el *trie* de sufijos con 7 vértices de terminación de sufijos (los círculos rellenos) y 3 vértices de terminación de palabras (los círculos rellenos etiquetados como ‘en el diccionario’).

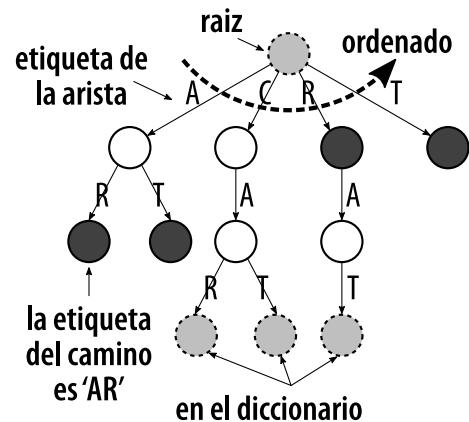


Figura 6.2: *Trie* de sufijos

<sup>9</sup>No se trata de una errata. La palabra ‘TRIE’ viene del término inglés ‘information retrieval’.

Un *trie* de sufijos se utiliza, normalmente, como una estructura de datos eficiente como *diccionario*. Asumiendo que se ha construido el *trie* de sufijos de un conjunto de cadenas como un diccionario, podemos determinar si una cadena de búsqueda/patrón  $P$  existe en este diccionario (*trie* de sufijos) en  $O(m)$ , donde  $m$  es la longitud de  $P$ . Este mecanismo es eficiente<sup>10</sup>. Operamos recorriendo el *trie* de sufijos desde la raíz. Por ejemplo, si queremos saber si la palabra  $P = \text{'CAT'}$  existe en el *trie* de sufijos de la figura 6.2, podemos empezar desde el nodo raíz, seguir la arista etiquetada como ‘C’, después ‘A’ y, finalmente, ‘T’. Como en este último vértice, el marcador de final de palabra es verdadero, sabemos que la palabra ‘CAT’ existe en el diccionario. Sin embargo, si buscásemos  $P = \text{'CAD'}$ , seguiríamos este camino: raíz → ‘C’ → ‘A’, pero, como después no encontraríamos una arista etiquetada como ‘D’, la conclusión sería que ‘CAD’ no está en el diccionario.

### Ejercicio 6.6.1.1\*

Implementar esta estructura de datos de *trie* de sufijos utilizando las ideas mencionadas, es decir, crear un objeto de vértices con hasta 26 aristas ordenadas, que representen de la ‘A’ a la ‘Z’, con sus correspondientes marcadores de terminación de sufijo/palabra. Insertar cada sufijo de cada cadena de  $S$  en el *trie* de sufijos, uno a uno. Analizar la complejidad de tiempo de esa construcción de *trie* de sufijos y compararla con la estrategia de construcción del *array* de sufijos de la sección 6.6.4. Realizar también  $O(m)$  consultas con varios patrones de búsqueda  $P$ , comenzando desde la raíz y siguiendo las etiquetas de las aristas correspondientes.

## 6.6.2 Árbol de sufijos

Ahora, en vez de trabajar con varias cadenas cortas, lo haremos con una (*más*) larga. Consideremos la cadena  $T = \text{'GATAGACA\$'}$ . El último carácter ‘\$’ es un carácter especial de terminación, que se añade a la cadena original ‘GATAGACA’. Tiene un valor ASCII menor que los caracteres de  $T$ . Este carácter de terminación asegura que todos los sufijos terminen en vértices hoja.

El *trie* de sufijos de  $T$  aparece en el centro de la figura 6.3. Esta vez el **vértice de terminación** almacena el *índice* del sufijo que termina en ese vértice. Hay que notar que, cuanto más larga es la cadena  $T$ , habrá más vértices duplicados en el *trie* de sufijos, lo que puede resultar poco eficiente. El **árbol** de sufijos de  $T$  es un *trie* de sufijos, donde *combinamos* los vértices con un único hijo (esencialmente realizamos una compresión de rutas). Al comparar los elementos del centro y de la derecha de la figura 6.3, podemos ver este proceso de compresión. Merecen especial atención las **etiquetas de aristas** y las **etiquetas de caminos** de la figura. Esta vez, la etiqueta de una arista puede contener más de un carácter. Un **árbol** de sufijos es mucho más *compacto* que un *trie* de sufijos, con un máximo de  $2n$  vértices<sup>11</sup> (y, por tanto, un máximo de  $2n - 1$  aristas). Por todo ello, utilizaremos un árbol de sufijos, en vez de un *trie* de sufijos, en las siguientes secciones.

<sup>10</sup>Otra estructura de datos para diccionarios es el BST equilibrado (ver la sección 2.3). Tiene un rendimiento de  $O(\log n \times m)$  para cada búsqueda en el diccionario, donde  $n$  es el número de palabras que hay en el mismo. Esto se debe a que una comparación de cadenas ya tiene un coste de  $O(m)$ .

<sup>11</sup>Hay un máximo de  $n$  hojas para  $n$  sufijos. Todos los vértices internos que no sean el raíz originan ramas, por lo que no puede haber más de  $n - 1$  de esos vértices. Total:  $n$  (hojas) +  $(n - 1)$  (nodos internos) + 1 (raíz) =  $2n$  vértices.

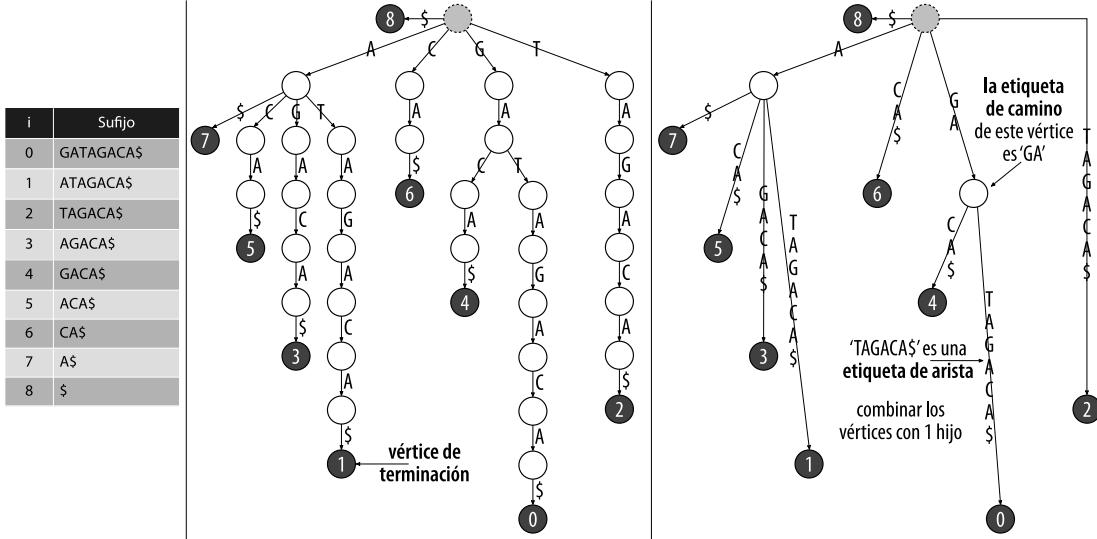


Figura 6.3: Sufijos, *trie* de sufijos y árbol de sufijos de  $T = \text{'GATAGACAS'}$

El árbol de sufijos puede ser una estructura de datos nueva para la mayoría de lectores de este libro. Por lo tanto, en tercera edición, hemos añadido una herramienta de visualización, para mostrar el árbol de sufijos de cualquier cadena (relativamente corta)  $T$ , especificada por el propio lector. También se incluyen en la visualización varias aplicaciones de un árbol de sufijos, que mostramos en la siguiente sección 6.6.3.

[visualgo.net/suffixtree](http://visualgo.net/suffixtree)

### Ejercicio 6.6.2.1\*

Dibuja el *trie* y el árbol de sufijos de  $T = \text{'COMPETITIVE$'}$ . Consejo: utiliza la herramienta de visualización del árbol de sufijos mencionada antes.

### Ejercicio 6.6.2.2\*

Dados dos vértices que representan dos sufijos diferentes, por ejemplo los sufijos 1 y 5 de la sección derecha de la figura 6.3, determinar su prefijo común más largo ('A').

## 6.6.3 Aplicaciones del árbol de sufijos

Asumiendo que el árbol de sufijos de una cadena  $T$  ya ha sido construido, podemos utilizarlo en las siguientes aplicaciones (la lista no pretende ser exhaustiva):

### Coincidencia de cadenas en $O(m + occ)$

Con un árbol de sufijos, podemos encontrar (exactamente) todas las apariciones de un patrón  $P$  en  $T$  en  $O(m + occ)$ , donde  $m$  es la longitud de la cadena patrón  $P$  y  $occ$  es el número total de apariciones de  $P$  en  $T$ , *independientemente* de la longitud de  $T$ . Cuando el árbol de sufijos *ya está construido*, esta técnica es *mucho más rápida* que los algoritmos de coincidencia de cadenas tratados en la sección 6.4.

Dado el árbol de sufijos de  $T$ , nuestra tarea consiste en buscar los vértices  $x$  en el árbol de sufijos, cuya etiqueta de camino represente el patrón  $P$ . Hay que recordar que una coincidencia es, después de todo, un *prefijo común* entre la cadena patrón  $P$  y algunos sufijos de la cadena  $T$ . Esto se hace en un único recorrido de la raíz a una hoja del árbol de sufijos de  $T$ , siguiendo las etiquetas de las aristas. El vértice cuya etiqueta de ruta sea igual a  $P$ , es el vértice  $x$  buscado. A continuación, los índices de sufijos almacenados en los vértices de terminación (hojas) del subárbol con raíz en  $x$ , son las apariciones de  $P$  en  $T$ .

Por ejemplo, en el árbol de sufijos de  $T = \text{'GATAGACA\$'}$  de la figura 6.4 y con  $P = \text{'A'}$ , podemos, sencillamente, recorrer desde la raíz, siguiendo la arista etiquetada como ‘A’, hasta encontrar el vértice  $x$ , con la etiqueta de camino ‘A’. Hay 4 apariciones<sup>12</sup> de ‘A’ en el subárbol con raíz en  $x$ . Son los sufijos 7 (‘A\$’), 5 (‘ACA\$’), 3 (‘AGACA\$’ ) y 1 (‘ATAGACA\$’ ).

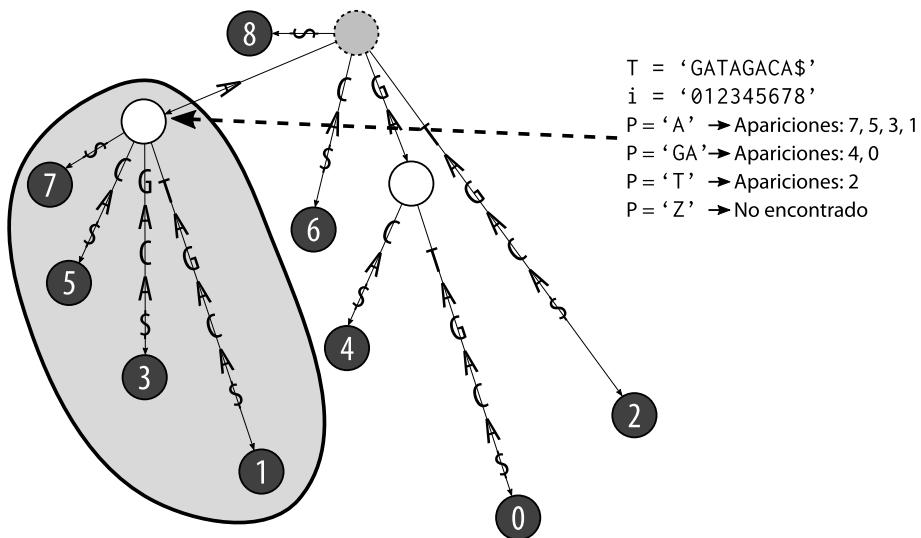


Figura 6.4: Coincidencia de cadenas de  $T = \text{'GATAGACA$'}$  con varios patrones

### Búsqueda de la subcadena repetida más larga en $O(n)$

Dado el árbol de sufijos de  $T$ , también podemos encontrar la subcadena repetida más larga<sup>13</sup> (LRS) en  $T$  de forma eficiente. El problema de la LRS es el problema de encontrar la subcadena

<sup>12</sup>Para ser más precisos,  $occ$  es el *tamaño* del subárbol con raíz en  $x$ , que puede ser más grande (aunque no más del doble) del número real ( $occ$ ) de vértices de terminación (hojas) del subárbol con raíz en  $x$ .

<sup>13</sup>Este problema tiene algunas aplicaciones interesantes: encontrar el estribillo de una canción (que se repite varias veces), encontrar las frases (más largas) que se repiten en un discurso político, etc.

más larga de una cadena, que aparece, *al menos*, dos veces. La etiqueta de camino del vértice *interno más profundo*  $x$  en el árbol de sufijos de  $T$  será la respuesta. El vértice  $x$  se puede encontrar en  $O(n)$  con un recorrido. El hecho de que  $x$  sea un vértice interno, implica que representa más de un sufijo de  $T$  (habrá  $> 1$  vértices de terminación en el subárbol con raíz en  $x$ ) y esos sufijos comparten un prefijo común (lo que implica una subcadena repetida). Al ser  $x$  el vértice interno *más profundo* (desde la raíz), su etiqueta de camino es la subcadena repetida *más larga*.

Ejemplo: en el árbol de sufijos de  $T = \text{'GATAGACA\$'}$  de la figura 6.5, la LRS es ‘GA’, ya que resulta ser la etiqueta de ruta del vértice interno más profundo  $x$  (‘GA’ se repite dos veces en ‘GATAGACA\$’).

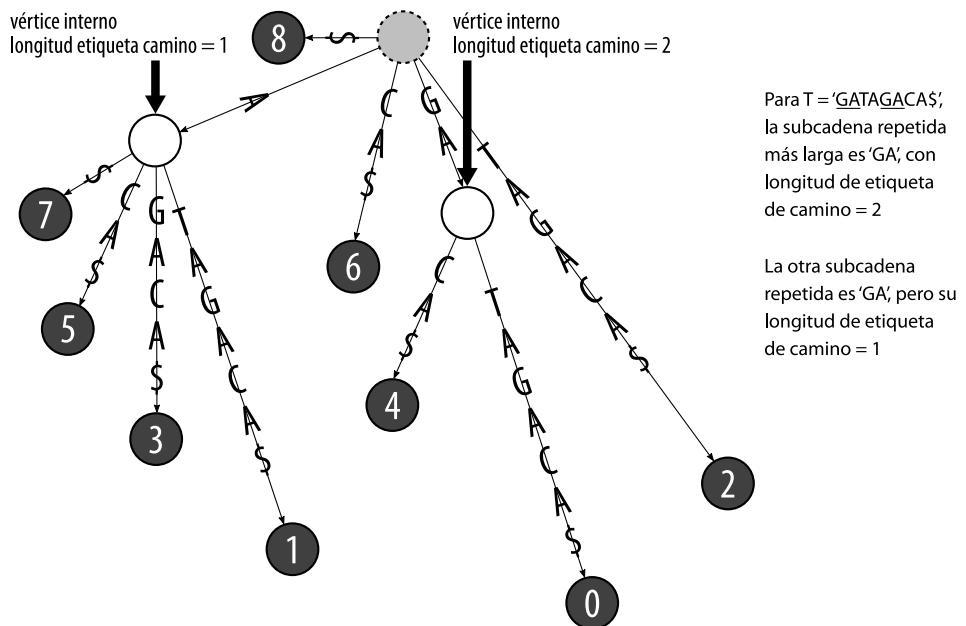


Figura 6.5: Subcadena repetida más larga de  $T = \text{'GATAGACA\$'}$

### Búsqueda de la subcadena común más larga en $O(n)$

El problema de encontrar la **subcadena** común más larga (LCS<sup>14</sup>) de dos o más cadenas, se puede resolver en tiempo lineal<sup>15</sup> con un árbol de sufijos. Sin perder la generalidad, consideremos el caso de solo *dos* cadenas  $T_1$  y  $T_2$ . Podemos construir un **árbol de sufijos generalizado** que combine el árbol de sufijos de  $T_1$  y  $T_2$ . Para diferenciar el origen de cada sufijo, podemos emplear dos símbolos de vértice de terminación diferentes, uno para cada cadena. Después, marcamos los vértices *internos* que tengan vértices en sus subárboles, con símbolos de terminación *diferentes*. Los sufijos representados por estos vértices de terminación internos marcados, comparten un

<sup>14</sup>Hay que tener en cuenta que ‘subcadena’ no es lo mismo que ‘subsecuencia’. Por ejemplo, “BCE” es una subsecuencia, pero no una subcadena, de “ABCDEF”, mientras que “BCD” (letras contiguas) es tanto una subsecuencia como una subcadena de “ABCDEF”.

<sup>15</sup>Solo si utilizamos el algoritmo de construcción de un árbol de sufijos en tiempo lineal (que no tratamos en este libro, ver [65]).

prefijo común y vienen *conjuntamente* de  $T_1$  y  $T_2$ . Esto es, los vértices internos marcados representan las subcadenas comunes entre  $T_1$  y  $T_2$ . Como nos interesa la subcadena común *más larga*, nuestra respuesta será la etiqueta de la ruta del vértice marcado *más profundo*.

Por ejemplo, con  $T_1 = \text{'GATAGACA\$'}$  y  $T_2 = \text{'CATA#}'$ , la subcadena común más larga es ‘ATA’, de longitud 3. En la figura 6.6, podemos ver que los vértices con etiquetas de ruta ‘A’, ‘ATA’, ‘CA’ y ‘TA’, tienen dos símbolos de terminación diferentes (hay que notar que el vértice con la etiqueta de camino ‘GA’ *no* se considera, ya que ambos sufijos ‘GACA\$’ y ‘GATAGACA\$’ vienen de  $T_1$ ). Esas son las subcadenas comunes entre  $T_1$  y  $T_2$ . El vértice más profundo marcado es ‘ATA’ y es la subcadena común más larga de  $T_1$  y  $T_2$ .

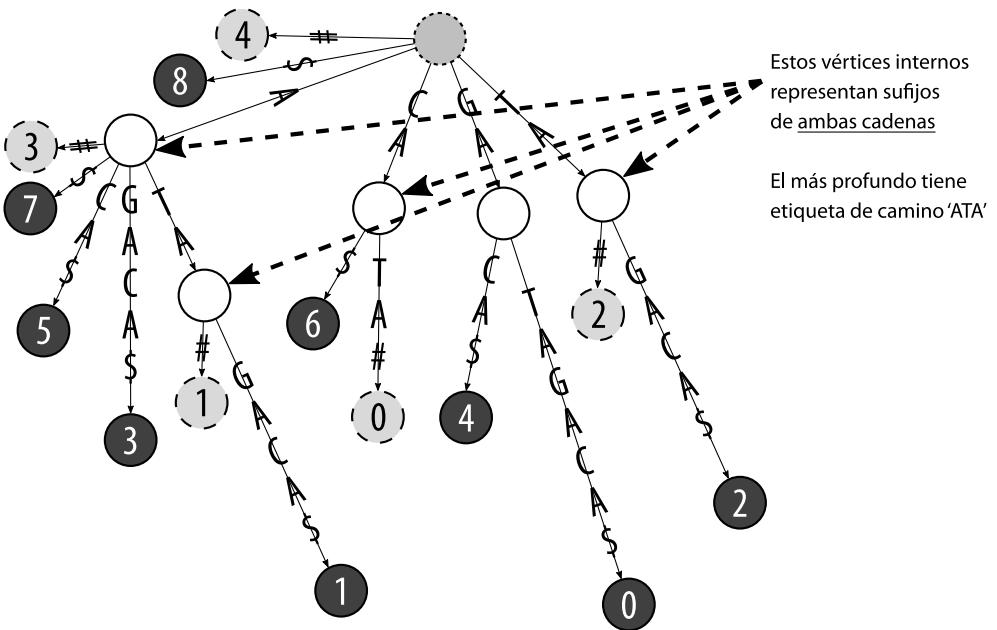


Figura 6.6: Árbol de sufijos generalizado de  $T_1 = \text{'GATAGACA\$'}$  y  $T_2 = \text{'CATA#}'$  y su LCS

### Ejercicio 6.6.3.1

Dado el mismo árbol de sufijos de la figura 6.4, encuentra  $P_1 = \text{'CA'}$  y  $P_2 = \text{'CAT'}$ .

### Ejercicio 6.6.3.2

Encuentra la LRS de  $T = \text{'CGACATTACATTA\$'}$ . Construye primero el árbol de sufijos.

### Ejercicio 6.6.3.3\*

En vez de encontrar la LRS, ahora queremos localizar la subcadena repetida *que aparezca más veces*. Entre varias candidatas posibles, selecciona la más larga. Por ejemplo, si  $T = \text{'DEFG1ABC2DEFG3ABC4ABC\$'}$ , la respuesta es ‘ABC’, de longitud 3, que aparece tres veces (y no ‘BC’, de longitud 2, o ‘C’, de longitud 1, que también aparecen tres veces), en vez de ‘DEFG’, de longitud 4, pero que solo aparece dos veces. Analiza la estrategia para resolverlo.

### Ejercicio 6.6.3.4

Encuentra la LCS de  $T_1 = \text{'STEVEN\$'}$  y  $T_2 = \text{'SEVEN#'}.$

### Ejercicio 6.6.3.5\*

Piensa en cómo generalizar esta técnica para encontrar la LCS de *más de dos cadenas*. Por ejemplo, dadas tres cadenas  $T_1 = \text{'STEVEN\$'}$ ,  $T_2 = \text{'SEVEN#'}$  y  $T_3 = \text{'EVE@'}$ , ¿cómo podemos determinar que su LCS es ‘EVE’?

### Ejercicio 6.6.3.6\*

Modifica más la solución, para encontrar la LCS de  $k$  cadenas de un conjunto de  $n$ , donde  $k \leq n$ . Por ejemplo, dadas las mismas tres cadenas  $T_1$ ,  $T_2$  y  $T_3$ , del ejercicio anterior, ¿cómo podemos determinar que la LCS de 2 de las 3 cadenas es ‘EVEN’?

## 6.6.4 Array de sufijos

En la subsección anterior, hemos mostrado algunos problemas de procesamiento de cadenas que se pueden resolver *si el árbol de sufijos ya ha sido construido*. Sin embargo, la implementación eficiente de la construcción del árbol de sufijos en tiempo lineal (ver [65]) es complejo y, por ello, un riesgo en el entorno de un concurso de programación. Por suerte, la siguiente estructura de datos que vamos a describir, el **array de sufijos**, inventado por Udi Manber y Gene Myers [43], tiene una funcionalidad similar a un árbol de sufijos, pero con una construcción y uso (mucho) más sencillos, especialmente durante un concurso de programación. No vamos, por lo tanto, a tratar la construcción del árbol de sufijos en  $O(n)$  [65] y, en su lugar, nos centraremos en la construcción del array de sufijos en  $O(n \log n)$  [68], que es más fácil de utilizar. En la siguiente subsección, mostraremos que podemos aplicar el array de sufijos para resolver problemas que ya hemos resuelto con el árbol de sufijos.

Un *array* de sufijos es, básicamente, un *array* de enteros que almacena una permutación de  $n$  índices de sufijos *ordenados*. Por ejemplo, consideremos la misma  $T = \text{'GATAGACA\$'}$  con  $n = 9$ .

| i | Sufijo     | i | SA[i]    | Sufijo     |
|---|------------|---|----------|------------|
| 0 | GATAGACA\$ | 0 | <b>8</b> | \$         |
| 1 | ATAGACA\$  | 1 | <b>7</b> | A\$        |
| 2 | TAGACA\$   | 2 | <b>5</b> | ACA\$      |
| 3 | AGACA\$    | 3 | <b>3</b> | AGACA\$    |
| 4 | GACA\$     | 4 | <b>1</b> | ATAGACA\$  |
| 5 | ACA\$      | 5 | <b>6</b> | CA\$       |
| 6 | CA\$       | 6 | <b>4</b> | GACA\$     |
| 7 | A\$        | 7 | <b>0</b> | GATAGACA\$ |
| 8 | \$         | 8 | <b>2</b> | TAGACA\$   |

Ordenado →

Figura 6.7: Ordenación de los sufijos de  $T = \text{'GATAGACA$'}$

El *array* de sufijos de  $T$  es una permutación de enteros  $[0..n-1] = \{8, 7, 5, 3, 1, 6, 4, 0, 2\}$ , como se muestra en la figura 6.7. Esto es, los sufijos ordenados son sufijo  $SA[0] = \text{sufijo } 8 = \$$ , sufijo  $SA[1] = \text{sufijo } 7 = \text{'A\$'}$ , sufijo  $SA[2] = \text{sufijo } 5 = \text{'ACA\$'}$ , ..., y, finalmente, sufijo  $SA[8] = \text{sufijo } 2 = \text{'TAGACA\$'}$ .

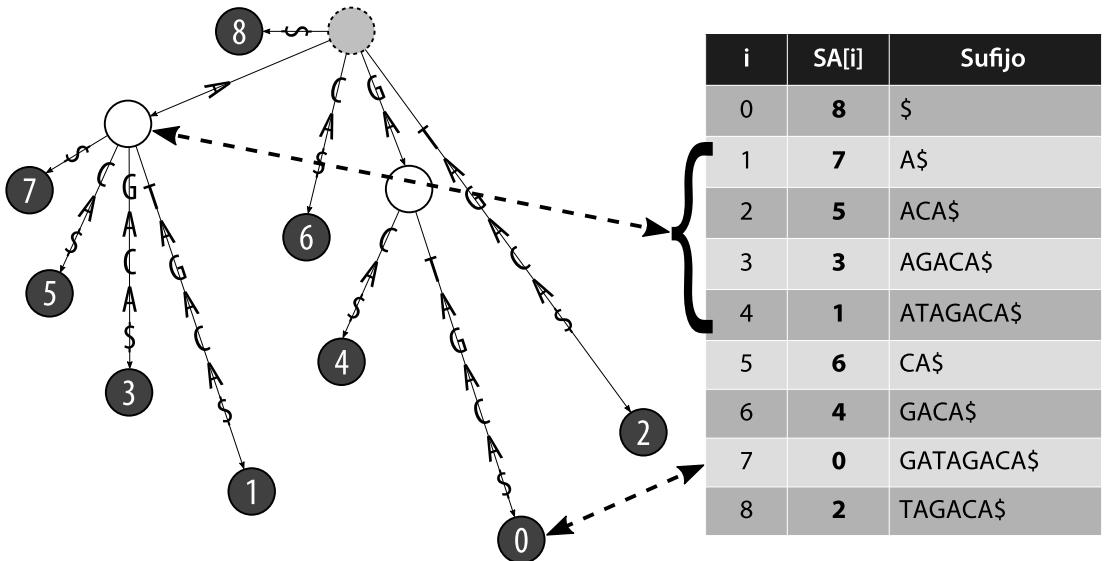


Figura 6.8: Árbol y *array* de sufijos de  $T = \text{'GATAGACA$'}$

El árbol y el *array* de sufijos están muy relacionados entre sí. Como vemos en la figura 6.8, el recorrido del árbol de sufijos visita los vértices de terminación (las hojas) en el orden del *array* de sufijos. Un **vértice interno** del árbol de sufijos, corresponde a un **rango** del *array* de sufijos (una colección de sufijos ordenados que comparten un prefijo común). Un **vértice de terminación** (siempre en una hoja, debido al uso del carácter de terminación) del árbol de sufijos, corresponde a un **índice individual** del *array* (un único sufijo). Conviene no olvidar

estas correspondencias, ya que nos serán útiles en la siguiente subsección, cuando veamos las aplicaciones del *array* de sufijos.

Un *array* de sufijos es suficiente, en los concursos de programación, para muchos problemas de cadenas difíciles que incorporan *cadenas largas*. Incluimos dos formas de construir un *array* de sufijos, dada una cadena  $T[0..n-1]$ . El primero es muy sencillo, como podemos ver:

```
1 #include <algorithm>
2 #include <cstdio>
3 #include <cstring>
4 using namespace std;
5
6 #define MAX_N 1010 // primera aproximación: $O(n^2 \log n)$
7 char T[MAX_N]; // esta construcción SA ingenua no pasa de 1000 caracteres
8 int SA[MAX_N], i, n; // en el entorno de un concurso de programación
9
10 bool cmp(int a, int b) { return strcmp(T+a, T+b) < 0; } // $O(n)$
11
12 int main() {
13 n = (int)strlen(gets(T)); // leer la línea y calcular su longitud
14 for (int i = 0; i < n; i++) SA[i] = i; // SA inicial: {0, 1, 2, ..., n-1}
15 sort(SA, SA+n, cmp); // ordenar: $O(n \log n) * cmp: O(n) = O(n^2 \log n)$
16 for (i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T+SA[i]);
17 } // return 0;
```

Este sencillo código, cuando se aplica a la cadena  $T = \text{'GATAGACA\$'}$ , ordena todos los sufijos con un sistema de ordenación incorporado en el lenguaje y una *biblioteca* de comparación de cadenas, generando correctamente el *array* de sufijos  $\{8, 7, 5, 3, 1, 6, 4, 0, 2\}$ . Sin embargo, esto no resulta muy útil en un concurso de programación, salvo que  $n \leq 1000$ . El tiempo de ejecución de este algoritmo es  $O(n^2 \log n)$ , porque la operación `strcmp` que se usa para determinar el orden de dos sufijos, posiblemente largos, tiene un coste de hasta  $O(n)$  por cada par de sufijos comparados.

Un *método mejor* para construir un *array* de sufijos, consiste en ordenar los *pares de rangos* (enteros pequeños) de los sufijos en  $O(\log_2 n)$  iteraciones, desde  $k = 1, 2, 4, \dots$ , la última **potencia de 2** que sea menor que  $n$ . Este algoritmo de construcción ordena, en cada iteración, los sufijos basados en el par de rangos  $(RA[SA[i]], RA[SA[i]+k])$  del sufijo  $SA[i]$ . Está basado en las técnicas tratadas en [68]. A continuación, mostramos un ejemplo de ejecución para  $T = \text{'GATAGACA$'}$  y  $n = 9$ .

- En primer lugar,  $SA[i] = i$  y  $RA[i] = \text{valor ASCII de } T[i] \forall i \in [0..n-1]$  (izquierda de la tabla 6.1). En la iteración  $k = 1$ , el par de rangos de  $SA[i]$  es  $(RA[SA[i]], RA[SA[i]+1])$ .

Ejemplo 1: el rango del sufijo 5 ‘ACA\$’ es  $(\text{'A'}, \text{'C'}) = (65, 67)$ .

Ejemplo 2: el rango del sufijo 3 ‘AGACA\$’ es  $(\text{'A'}, \text{'G'}) = (65, 71)$ .

Después de ordenar los pares de rangos, el orden de los sufijos será el de la parte derecha de la tabla 6.1, donde el sufijo 5 ‘ACA\$’ se coloca antes que el sufijo 3 ‘AGACA\$', etc.

- En la iteración  $k = 2$ , el par de rangos del sufijo  $SA[i]$  es  $(RA[SA[i]], RA[SA[i]+2])$ . Este par de rangos se obtiene consultando únicamente el primer y el segundo par de caracteres.

| i                                                                                       | SA[i] | Sufijo     | RA[SA[i]] | RA[SA[i]+1] | i                                                                                                            | SA[i] | Sufijo     | RA[SA[i]] | RA[SA[i]+1] |
|-----------------------------------------------------------------------------------------|-------|------------|-----------|-------------|--------------------------------------------------------------------------------------------------------------|-------|------------|-----------|-------------|
| 0                                                                                       | 0     | GATAGACA\$ | 71 (G)    | 65 (A)      | 0                                                                                                            | 8     | \$         | 36 (\$)   | 00 (-)      |
| 1                                                                                       | 1     | ATAGACA\$  | 65 (A)    | 84 (T)      | 1                                                                                                            | 7     | A\$        | 65 (A)    | 36 (\$)     |
| 2                                                                                       | 2     | TAGACA\$   | 84 (T)    | 65 (A)      | 2                                                                                                            | 5     | ACA\$      | 65 (A)    | 67 (C)      |
| 3                                                                                       | 3     | AGACA\$    | 65 (A)    | 71 (G)      | 3                                                                                                            | 3     | AGACA\$    | 65 (A)    | 71 (G)      |
| 4                                                                                       | 4     | GACA\$     | 71 (G)    | 65 (A)      | 4                                                                                                            | 1     | ATAGACA\$  | 65 (A)    | 84 (T)      |
| 5                                                                                       | 5     | ACA\$      | 65 (A)    | 67 (C)      | 5                                                                                                            | 6     | CA\$       | 67 (C)    | 65 (A)      |
| 6                                                                                       | 6     | CA\$       | 67 (C)    | 65 (A)      | 6                                                                                                            | 0     | GATAGACA\$ | 71 (G)    | 65 (A)      |
| 7                                                                                       | 7     | A\$        | 65 (A)    | 36 (\$)     | 7                                                                                                            | 4     | GACA\$     | 71 (G)    | 65 (A)      |
| 8                                                                                       | 8     | \$         | 36 (\$)   | 00 (-)      | 8                                                                                                            | 2     | TAGACA\$   | 84 (T)    | 65 (A)      |
| Rangos iniciales RA[i] = valor ASCII de T[i]<br>\$ = 36, A = 65, C = 67, G = 71, T = 84 |       |            |           |             | Si SA[i] + k >= n (mayor a la longitud de la cadena T),<br>damos un rango predeterminado de 0 con etiqueta - |       |            |           |             |

Tabla 6.1: I/D: antes y después de ordenar;  $k = 1$ ; aparece la ordenación inicial

Para obtener los nuevos pares de rangos, no es necesario recalcular mucho. Establecemos el primero, es decir, el sufijo 8 '\$' con el nuevo rango  $r = 0$ . Después, iteramos desde  $i = [1..n-1]$ . Si el par de rangos del sufijo  $SA[i]$  difiere del par de rangos del sufijo anterior  $SA[i-1]$  ya ordenado, incrementamos el rango  $r = r + 1$ . En caso contrario, el rango se mantiene en  $r$  (ver la parte izquierda de la tabla 6.2).

| i                                                                                                                        | SA[i] | Sufijo     | RA[SA[i]] | RA[SA[i]+2] | i                                                                                                            | SA[i] | Sufijo     | RA[SA[i]] | RA[SA[i]+2] |
|--------------------------------------------------------------------------------------------------------------------------|-------|------------|-----------|-------------|--------------------------------------------------------------------------------------------------------------|-------|------------|-----------|-------------|
| 0                                                                                                                        | 8     | \$         | 0 (\$-)   | 0 (--)      | 0                                                                                                            | 8     | \$         | 0 (\$-)   | 0 (--)      |
| 1                                                                                                                        | 7     | A\$        | 1 (A\$)   | 0 (--)      | 1                                                                                                            | 7     | A\$        | 1 (A\$)   | 0 (--)      |
| 2                                                                                                                        | 5     | ACA\$      | 2 (AC)    | 1 (A\$)     | 2                                                                                                            | 5     | ACA\$      | 2 (AC)    | 1 (A\$)     |
| 3                                                                                                                        | 3     | AGACA\$    | 3 (AG)    | 2 (AC)      | 3                                                                                                            | 3     | AGACA\$    | 3 (AG)    | 2 (AC)      |
| 4                                                                                                                        | 1     | ATAGACA\$  | 4 (AT)    | 3 (AG)      | 4                                                                                                            | 1     | ATAGACA\$  | 4 (AT)    | 3 (AG)      |
| 5                                                                                                                        | 6     | CA\$       | 5 (CA)    | 0 (\$-)     | 5                                                                                                            | 6     | CA\$       | 5 (CA)    | 0 (\$-)     |
| 6                                                                                                                        | 0     | GATAGACA\$ | 6 (GA)    | 7 (TA)      | 6                                                                                                            | 4     | GACA\$     | 6 (GA)    | 5 (CA)      |
| 7                                                                                                                        | 4     | GACA\$     | 6 (GA)    | 5 (CA)      | 7                                                                                                            | 0     | GATAGACA\$ | 6 (GA)    | 7 (TA)      |
| 8                                                                                                                        | 2     | TAGACA\$   | 7 (TA)    | 6 (GA)      | 8                                                                                                            | 2     | TAGACA\$   | 7 (TA)    | 6 (GA)      |
| A \$- (primer elemento) tiene rango 0, para i = 1 hasta n-1,<br>comparar los pares de rango de esta fila con la anterior |       |            |           |             | Si SA[i] + k >= n (mayor a la longitud de la cadena T),<br>damos un rango predeterminado de 0 con etiqueta - |       |            |           |             |

Tabla 6.2: I/D: antes/después de ordenar;  $k = 2$ ; se intercambian 'GATAGACA' y 'GACA'

Ejemplo 1: en la parte derecha de la tabla 6.1, el par de rangos del sufijo 7 'A\$' es (65, 36), que es diferente del par de rangos del sufijo 8 anterior '\$-', que es (36, 0). Por ello, en la parte izquierda de la tabla 6.2, el sufijo 7 tiene el nuevo rango 1.

Ejemplo 2: en la parte derecha de la tabla 6.1, el par de rangos del sufijo 4 'GACA\$' es (71, 65), que es similar al par de rangos del sufijo 0 anterior 'GATAGACA\$', que también es (71, 65). Por ello, en la parte izquierda de la tabla 6.2, ya que al sufijo 0 se le da un nuevo rango 6, el sufijo 4 obtiene el mismo nuevo rango 6.

Una vez que hemos actualizado  $RA[SA[i]] \forall i \in [0..n-1]$ , se puede determinar, también fácilmente, el valor de  $RA[SA[i]+k]$ . En nuestra explicación, si  $SA[i]+k \geq n$ , damos un rango predeterminado de 0. En el **ejercicio 6.6.4.2\*** se pueden encontrar más detalles sobre la implementación de este paso.

En este punto, el par de rangos del sufijo 0 ‘GATAGACA\$’ es (6, 7), y del sufijo 4 ‘GACA\$’ es (6, 5). Estos dos sufijos todavía no están ordenados, mientras que el resto sí. Después de otra ordenación, los sufijos se colocarán como se ve en el parte derecha del tabla 6.2.

- En la iteración  $k = 4$ , el par de rangos del sufijo  $SA[i]$  es  $(RA[SA[i]], RA[SA[i]+4])$ . Este par de rangos se obtiene consultando únicamente las primera y segunda 4-tuplas de caracteres. Ahora, nos encontramos con que el par de rangos anterior de los sufijos 4 (6, 5) y 0 (6,7), de la parte derecha de la tabla 6.2, son diferentes. Por lo tanto, después de recalcular los rangos, todos los  $n$  sufijos de la tabla 6.3 tienen un rango diferente. Es fácilmente verificable si  $RA[SA[n-1]] == n-1$ . Cuando esto ocurre, hemos obtenido el *array* de sufijos con éxito. Es relevante el hecho de que la principal carga de ordenación se ha realizado en las primeras iteraciones y, normalmente, no son necesarias muchas.

| i | SA[i] | Sufijo     | RA[SA[i]] | RA[SA[i]+4] |
|---|-------|------------|-----------|-------------|
| 0 | 8     | \$         | 0 (\$---  | 0 (----     |
| 1 | 7     | A\$        | 1 (A\$--) | 0 (----)    |
| 2 | 5     | ACA\$      | 2 (ACA\$) | 0 (----)    |
| 3 | 3     | AGACA\$    | 3 (AGAC)  | 1 (A\$--)   |
| 4 | 1     | ATAGACA\$  | 4 (ATAG)  | 2 (ACA\$)   |
| 5 | 6     | CA\$       | 5 (CA\$-) | 0 (----)    |
| 6 | 4     | GACA\$     | 6 (GACA)  | 0 (----)    |
| 7 | 0     | GATAGACA\$ | 7 (GATA)  | 6 (GACA)    |
| 8 | 2     | TAGACA\$   | 8 (TAGA)  | 5 (CA\$-)   |

Ahora todos los sufijos tienen rango diferente  
 y hemos terminado

Tabla 6.3: Antes/Después de ordenar;  $k = 4$ ; sin cambios

Este algoritmo de construcción de un *array* de sufijos puede resultar una novedad para la mayoría de lectores, por lo que, en la presente edición de este libro, hemos añadido una herramienta de visualización de un *array* de sufijos, para mostrar los pasos de cualquier cadena  $T$  (relativamente corta), determinada por el propio lector. También incluimos, en la siguiente sección 6.6.5, varias aplicaciones del *array* de sufijos.



Podemos implementar la ordenación de pares de rangos anterior, utilizando la biblioteca de ordenación (integrada) en  $O(n \log n)$ . Como repetimos el proceso de ordenación hasta  $\log n$  veces, la complejidad de tiempo global es de  $O(\log n \times n \log n) = O(n \log^2 n)$ . Con esta complejidad de tiempo, podemos trabajar con cadenas de una longitud de hasta  $\approx 10000$  caracteres. Sin embargo, ya que el proceso de ordenación trata solo con *pares de enteros pequeños*, podemos utilizar una ordenación *radix* de dos pasadas en *tiempo lineal* (que utiliza internamente la ordenación por cuentas, ver la sección 9.32), para reducir el tiempo de ordenación a  $O(n)$ . Como repetimos el proceso de ordenación hasta  $\log n$  veces, la complejidad de tiempo global será de  $O(\log n \times n) = O(n \log n)$ . Ahora podremos trabajar con cadenas de hasta  $\approx 100000$  caracteres, lo que es bastante habitual en concursos de programación.

A continuación, incluimos nuestra implementación en  $O(n \log n)$ . Repasa el código hasta entender cómo funciona. Solo para concursantes del ICPC: como está permitido llevar material escrito al concurso, es una buena idea incluir este código entre las notas del equipo.

```

1 #define MAX_N 100010 // segunda aproximación: O(n log n)
2
3 char T[MAX_N]; // cadena de entrada, hasta 100K caracteres
4 int n; // longitud de cadena de entrada
5 int RA[MAX_N], tempRA[MAX_N]; // array de rangos y array temporal de rangos
6 int SA[MAX_N], tempSA[MAX_N]; // array de sufijos y su temporal
7 int c[MAX_N]; // para conteo/ordenación radix
8
9 void countingSort(int k) { // O(n)
10 int i, sum, maxi = max(300, n); // 255 caracteres ASCII o longitud de n
11 memset(c, 0, sizeof c); // limpiar tabla de frecuencias
12 for (i = 0; i < n; i++) // contar frecuencia de cada rango entero
13 c[i+k < n ? RA[i+k] : 0]++;
14 for (i = sum = 0; i < maxi; i++) {
15 int t = c[i]; c[i] = sum; sum += t; }
16 for (i = 0; i < n; i++) // mezclar el array de sufijos si es necesario
17 tempSA[c[SA[i]]+k < n ? RA[SA[i]+k] : 0]++ = SA[i];
18 for (i = 0; i < n; i++) // actualizar el array de sufijos SA
19 SA[i] = tempSA[i];
20 }
21
22 void constructSA() { // esta versión admite hasta 100000 caracteres
23 int i, k, r;
24 for (i = 0; i < n; i++) RA[i] = T[i]; // rangos iniciales
25 for (i = 0; i < n; i++) SA[i] = i; // SA inicial: {0, 1, 2, ..., n-1}
26 for (k = 1; k < n; k <= 1) { // repetir ordenación log n veces
27 countingSort(k); // ordenación radix: en base al segundo elemento
28 countingSort(0); // entonces ordenación (estable) en base al primero
29 tempRA[SA[0]] = r = 0; // nuevo rango; comenzar en rango r = 0
30 for (i = 1; i < n; i++) // comparar sufijos adyacentes
31 tempRA[SA[i]] = r; // si mismo par => mismo rango r; si no, aumentar r
32 (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
33 for (i = 0; i < n; i++) // actualizar el array de rangos RA
34 RA[i] = tempRA[i];
35 if (RA[SA[n-1]] == n-1) break; // buen truco de optimización
36 } }
37
38 int main() {
39 n = (int)strlen(gets(T)); // entrada T normal, sin el '$'
40 T[n++] = '$'; // añadir carácter de terminación
41 constructSA();
42 for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T+SA[i]);
43 } // return 0;

```

### Ejercicio 6.6.4.1\*

Muestra los pasos para calcular el *array* de sufijos de  $T = \text{'COMPETITIVE$'}$  con  $n = 12$ . ¿Cuántas iteraciones de ordenación son necesarias para obtener ese *array*? Consejo: utiliza la herramienta de visualización del *array* de sufijos mencionada antes.

### Ejercicio 6.6.4.2\*

En el código del *array* de sufijos mostrado antes, ¿provocará la línea

```
(RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
```

que el índice se desborde en algunos casos? Es decir, ¿serán alguna vez  $SA[i]+k$  o  $SA[i-1]+k \geq n$  y harán fallar el programa? Razona tu respuesta.

### Ejercicio 6.6.4.3\*

¿Funcionará el código del *array* de sufijos mostrado antes, si la cadena de entrada  $T$  contiene un espacio en blanco (valor ASCII = 32)? Pista: el carácter de terminación predeterminado ('\$') tiene un valor ASCII = 36.

## 6.6.5 Aplicaciones del *array* de sufijos

Ya hemos mencionado que el *array* de sufijos está íntimamente relacionado con el árbol de sufijos. En esta subsección, mostramos que con un *array* de sufijos (que es más fácil de construir) podemos resolver también los problemas de procesamiento de cadenas que aparecían en la sección 6.6.3.

### Coincidencia de cadenas en $O(m \log n)$

Después de obtener el *array* de sufijos de  $T$ , podemos buscar una cadena patrón  $P$  (de longitud  $m$ ) dentro de  $T$  (de longitud  $n$ ) en  $O(m \log n)$ . Esto supone un factor de  $\log n$  veces más lento que en la versión del árbol de sufijos, pero en la práctica es bastante aceptable. La complejidad  $O(m \log n)$  viene del hecho de que podemos hacer dos búsquedas binarias  $O(\log n)$  en sufijos ordenados y hasta  $O(m)$  comparaciones de sufijos<sup>16</sup>. La primera y segunda búsquedas binarias consisten en encontrar los límites inferior y superior, respectivamente. Estos límites, inferior y superior, corresponden a las  $i$  más pequeña y más grande, de forma que el prefijo de los sufijos  $SA[i]$  coincida con el patrón  $P$ , también respectivamente. Todos los sufijos entre los límites superior e inferior son las apariciones de la cadena patrón  $P$  en  $T$ . A continuación, mostramos nuestra implementación:

<sup>16</sup>Esto se consigue utilizando la función `strncmp`, para comparar solo los  $m$  primeros caracteres de ambos.

```

1 int stringMatching() { // coincidencia de cadenas en O(m log n)
2 int lo = 0, hi = n-1, mid = lo; // coincidencia válida = [0..n-1]
3 while (lo < hi) { // encontrar límite inferior
4 mid = (lo+hi) / 2; // redondeo hacia abajo
5 int res = strncmp(T+SA[mid], P, m); // encontrar P en el sufijo 'mid'
6 if (res >= 0) hi = mid; // podar mitad superior (atención signo >=)
7 else lo = mid+1; // podar mitad inferior incluyendo mid
8 } // observar '=' en "res >= 0" antes
9 if (strncmp(T+SA[lo], P, m) != 0) return ii(-1, -1); // si no encontrado
10 ii ans; ans.first = lo;
11 lo = 0; hi = n-1; mid = lo;
12 while (lo < hi) { // si encontrado límite inferior, buscar superior
13 mid = (lo+hi) / 2;
14 int res = strncmp(T+SA[mid], P, m);
15 if (res > 0) hi = mid; // podar mitad superior
16 else lo = mid+1; // podar mitad inferior incluyendo mid
17 } // (notar rama seleccionada cuando res == 0)
18 if (strncmp(T+SA[hi], P, m) != 0) hi--;
19 ans.second = hi;
20 return ans;
21 } // devolver límite inferior/superior como primer/segundo elemento del par
22
23 int main() {
24 n = (int)strlen(gets(T)); // entrada T normal, sin el '$'
25 T[n++] = '$'; // añadir carácter de terminación
26 constructSA();
27 for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T+SA[i]);
28
29 while (m = (int)strlen(gets(P)), m) { // parar si P es una cadena vacía
30 ii pos = stringMatching();
31 if (pos.first != -1 && pos.second != -1) {
32 printf("%s found, SA [%d..%d] of %s\n", P, pos.first, pos.second, T);
33 printf("They are:\n");
34 for (int i = pos.first; i <= pos.second; i++)
35 printf(" %s\n", T+SA[i]);
36 } else printf("%s is not found in %s\n", P, T);
37 } } // return 0;

```

En la tabla 6.4 se muestra una ejecución de ejemplo de este algoritmo de coincidencia de cadenas, con el *array* de sufijos de  $T = \text{'GATAGACA\$'}$  y con  $P = \text{'GA'}$ .

Empezamos localizando el límite inferior. El rango actual es  $i = [0..8]$  y, por ello, el índice central es  $i = 4$ . Comparamos los dos primeros caracteres del sufijo  $SA[4]$ , que son 'ATAGACA\$', con  $P = \text{'GA'}$ . Como  $P = \text{'GA'}$  es más grande, continuamos explorando  $i = [5..8]$ . Después, comparamos los dos primeros caracteres del sufijo  $SA[6]$ , que son 'GACA\$', con  $P = \text{'GA'}$ . Coincidén. Como estamos buscando el límite *inferior*, no nos detenemos aquí, sino que continuamos explorando  $i = [5..6]$ .  $P = \text{'GA'}$  es más grande que el sufijo  $SA[5]$ , que es 'CA\$'. Nos detenemos. El índice  $i = 6$  es el límite inferior, es decir, en el sufijo  $SA[6]$ , que es 'GACA\$', aparece

por primera vez el patrón  $P = 'GA'$ , como prefijo de un sufijo contenido en la lista de sufijos ordenados.

**Búsqueda del límite inferior**

| i | SA[i] | Sufijo     |
|---|-------|------------|
| 0 | 8     | \$         |
| 1 | 7     | A\$        |
| 2 | 5     | ACA\$      |
| 3 | 3     | AGACA\$    |
| 4 | 1     | XATAGACA\$ |
| 5 | 6     | XCA\$      |
| 6 | 4     | GACA\$     |
| 7 | 0     | GATAGACA\$ |
| 8 | 2     | TAGACA\$   |

**Búsqueda del límite superior**

| i | SA[i] | Sufijo     |
|---|-------|------------|
| 0 | 8     | \$         |
| 1 | 7     | A\$        |
| 2 | 5     | ACA\$      |
| 3 | 3     | AGACA\$    |
| 4 | 1     | XATAGACA\$ |
| 5 | 6     | CA\$       |
| 6 | 4     | GACA\$     |
| 7 | 0     | GATAGACA\$ |
| 8 | 2     | XTAGACA\$  |

Tabla 6.4: Coincidencia de cadenas utilizando un *array* de sufijos

Después, buscamos el límite superior. El primer paso es igual al anterior. Pero, en el segundo, tenemos una coincidencia entre el sufijo  $SA[6]$ , que es 'GACA\$', con  $P = 'GA'$ . Como lo que buscamos es el límite *superior*, debemos continuar explorando  $i = [7..8]$ . Encontramos otra coincidencia al comparar el sufijo  $SA[7]$ , que es 'GATAGACA\$', con  $P = 'GA'$ . Nos detenemos aquí. El límite superior en este ejemplo es  $i = 7$ , es decir, el sufijo  $SA[7]$ , que corresponde a 'GATAGACA\$', es la *última vez* que el patrón  $P = 'GA'$  aparece como prefijo de un sufijo contenido en la lista de sufijos ordenados.

#### Búsqueda del prefijo común más largo en $O(n)$

Dado el *array* de sufijos de  $T$ , podemos calcular el prefijo común más largo (LCP) entre sufijos *consecutivos* en el orden del *array* de sufijos. Por definición,  $LCP[0] = 0$ , ya que el sufijo  $SA[0]$  es el primero del *array* de sufijos ordenado, no habiendo otro que lo anteceda. Para  $i > 0$ ,  $LCP[i] =$  la longitud del prefijo común entre los sufijos  $SA[i]$  y  $SA[i-1]$ . Podemos verlo en la parte izquierda de la tabla 6.5. Es posible calcular directamente el LCP, por definición, utilizando el código que reproducimos a continuación. Sin embargo, este método es lento, ya que puede incrementar el valor de  $L$  en hasta  $O(n^2)$  veces. Esto hace que no tenga sentido construir el *array* de sufijos en  $O(n \log n)$ , como se muestra en la sección 6.8.

```

1 void computeLCP_slow() {
2 LCP[0] = 0; // valor predeterminado
3 for (int i = 1; i < n; i++) { // calcular LCP por definición
4 int L = 0; // reiniciar siempre L a 0
5 while (T[SA[i]+L] == T[SA[i-1]+L]) L++; // mismo carácter L-ésimo, L++
6 LCP[i] = L;
7 }
}

```

A continuación, se describe una solución mejor, utilizando el teorema del prefijo común más largo permutado [37]. La idea es sencilla: es *más fácil* calcular el LCP en su posición en el orden

original de los sufijos, que en el orden lexicográfico. En la parte derecha de la tabla 6.5, tenemos las posiciones según el orden original de los sufijos de  $T = 'GATAGACAS'$ . Hay que observar que la columna  $PLCP[i]$  forma un patrón: un bloque que decrece 2 ( $2 \rightarrow 1 \rightarrow 0$ ), crece a 1, vuelve a decrecer 1 ( $1 \rightarrow 0$ ), vuelve a crecer a 1, vuelve a decrecer 1 ( $1 \rightarrow 0$ ), etc.

| i | $SA[i]$ | $LCP[i]$ | Sufijo     | i | $\Phi[i]$ | $PLCP[i]$ | Sufijo     |
|---|---------|----------|------------|---|-----------|-----------|------------|
| 0 | 8       | 0        | \$         | 0 | 4         | 2         | GATAGACA\$ |
| 1 | 7       | 0        | A\$        | 1 | 3         | 1         | ATAGACA\$  |
| 2 | 5       | 1        | ACA\$      | 2 | 0         | 0         | TAGACA\$   |
| 3 | 3       | 1        | AGACA\$    | 3 | 5         | 1         | AGACA\$    |
| 4 | 1       | 1        | ATAGACA\$  | 4 | 6         | 0         | GACA\$     |
| 5 | 6       | 0        | CA\$       | 5 | 7         | 1         | ACA\$      |
| 6 | 4       | 0        | GACA\$     | 6 | 1         | 0         | CA\$       |
| 7 | 0       | 2        | GATAGACA\$ | 7 | 8         | 0         | A\$        |
| 8 | 2       | 0        | TAGACA\$   | 8 | -1        | 0         | \$         |

$LCP[7] = PLCP[SA[7]] = PLCP[0] = 2$

$\Phi[\Phi[SA[3]]] = SA[3-1]$   
 $\Phi[3] = SA[2]$   
 $\Phi[3] = 5$

Tabla 6.5: Cálculo del LCP dado el SA de  $T = 'GATAGACA$'$

El teorema PLCP dice que el número total de operaciones de incremento (y decremento) tiene un máximo de  $O(n)$ . En el código mostrado más adelante, se utilizan tanto ese patrón como la garantía de  $O(n)$ .

Comenzamos calculando  $\Phi[i]$ , que almacena el índice del sufijo anterior al sufijo  $SA[i]$ , en orden de *array* de sufijos. Por definición,  $\Phi[SA[0]] = -1$ , es decir, no hay un sufijo que preceda a  $SA[0]$ . Podemos tomarnos un tiempo para verificar que la columna  $\Phi[i]$  de la parte derecha de la tabla 6.5 es correcta. Por ejemplo,  $\Phi[SA[3]] = SA[3-1]$ , por ello  $\Phi[3] = SA[2] = 5$ .

Ahora, con  $\Phi[i]$ , podemos calcular el LCP permutado. A continuación, se desarrollan los primeros pasos de este algoritmo. Cuando  $i = 0$ , tenemos  $\Phi[0] = 4$ . Esto significa que el sufijo 0 ‘GATAGACA\$’ está después del sufijo 4 ‘GACA\$’ en orden del *array* de sufijos. Los dos primeros caracteres ( $L = 2$ ) de estos dos sufijos coinciden, por lo que  $PLCP[0] = 2$ .

Cuando  $i = 1$ , sabemos que hay *al menos*  $L - 1 = 1$  caracteres que pueden coincidir, al tener el siguiente sufijo ordenado un carácter inicial menos que el actual. Tenemos  $\Phi[1] = 3$ . Esto significa que el sufijo 1 ‘ATAGACA\$’, está después del sufijo 3 ‘AGACA\$’, en orden del *array* de sufijos. Hay que fijarse en que estos dos sufijos tienen, efectivamente, al menos una coincidencia de 1 carácter (es decir, no empezamos en  $L = 0$ , como en la función `computeLCP_slow()` vista anteriormente y, por ello, esta opción es más eficiente). Como no podemos extenderlo más, tenemos que  $PLCP[1] = 1$ .

Continuamos este proceso hasta que  $i = n - 1$ , saltando el caso cuando  $\Phi[i] = -1$ . Como el teorema PLCP dice que  $L$  será incrementado/decrementado  $n$  veces como mucho, esta parte se ejecuta adecuadamente en  $O(n)$ . Por último, una vez que tenemos el *array* PLCP, podemos poner el LCP permutado nuevamente en su posición correcta. El código es relativamente corto, como puede verse a continuación:

```

1 void computeLCP() {
2 int i, L;
3 Phi[SA[0]] = -1; // valor predeterminado

```

```

4 for (i = 1; i < n; i++) // calcular Φ en $O(n)$
5 Φ [SA[i]] = SA[i-1]; // recordar qué sufijo está detrás de este
6 for (i = L = 0; i < n; i++) { // calcular LCP permutada en $O(n)$
7 if (Φ [i] == -1) { PLCP[i] = 0; continue; } // caso especial
8 while (T[i+L] == T[Φ [i]+L]) L++; // L aumentado máximo n veces
9 PLCP[i] = L;
10 L = max(L-1, 0); // L reducido máximo n veces
11 }
12 for (i = 0; i < n; i++) // calcular LCP en $O(n)$
13 LCP[i] = PLCP[SA[i]]; // poner LCP permutada en su posición correcta
14 }

```

### Búsqueda de la subcadena repetida más larga en $O(n)$

Si hemos calculado el *array* de sufijos en  $O(n \log n)$  y el LCP entre sufijos consecutivos en orden del *array* de sufijos en  $O(n)$ , podemos determinar la longitud de la subcadena repetida más larga (LRS) de  $T$  en  $O(n)$ .

La longitud de la subcadena repetida más larga es, precisamente, el número mayor del *array* LCP. En la parte izquierda de la tabla 6.5, que corresponde al *array* de sufijos y al LCP de  $T = \text{'GATAGACA\$'}$ , el número mayor es 2, en el índice  $i = 7$ . Los dos primeros caracteres del sufijo correspondiente  $SA[7]$  (sufijo 0) son ‘GA’. Esta es la subcadena repetida más larga en  $T$ .

### Búsqueda de la subcadena común más larga en $O(n)$

| i  | SA[i] | LCP[i] | Propietario | Sufijo          |
|----|-------|--------|-------------|-----------------|
| 0  | 13    | 0      | 2           | #               |
| 1  | 8     | 0      | 1           | \$CATA#         |
| 2  | 12    | 0      | 2           | A#              |
| 3  | 7     | 1      | 1           | A\$CATA#        |
| 4  | 5     | 1      | 1           | ACA\$CATA#      |
| 5  | 3     | 1      | 1           | AGACA\$CATA#    |
| 6  | 10    | 1      | 2           | ATA#            |
| 7  | 1     | 3      | 1           | ATAGACA\$CATA#  |
| 8  | 6     | 0      | 1           | CA\$CATA#       |
| 9  | 9     | 2      | 2           | CATA#           |
| 10 | 4     | 0      | 1           | GACA\$CATA#     |
| 11 | 0     | 2      | 1           | GATAGACA\$CATA# |
| 12 | 11    | 0      | 2           | TA#             |
| 13 | 2     | 2      | 1           | TAGACA\$CATA#   |

Tabla 6.6: *Array* de sufijos, LCP y propietario de  $T = \text{'GATAGACA\$CATA#}'$

Sin perder la generalidad, consideremos un caso en el que solo tenemos *dos* cadenas. Utilizaremos el mismo ejemplo que en la sección del árbol de sufijos:  $T_1 = \text{'GATAGACA\$'}$  y  $T_2 = \text{'CATA#'}$ .

Para resolver el problema del LCS utilizando un *array* de sufijos, primero debemos concatenar ambas cadenas (los caracteres de terminación de ambas *deben ser diferentes*), para obtener  $T = \text{'GATAGACA\$CATA#}'$ . Después, calculamos los *arrays* de sufijos y LCP de  $T$ , como se muestra en la figura 6.6.

A continuación, recorremos los sufijos consecutivos en  $O(n)$ . Si dos sufijos consecutivos pertenecen a un propietario diferente (se puede comprobar muy fácilmente<sup>17</sup>, podemos, por ejemplo, saber si el sufijo  $SA[i]$  pertenece a  $T_1$ , verificando si  $SA[i] <$  la longitud de  $T_1$ ), consultamos el *array* LCP y comprobamos si el LCP máximo encontrado hasta el momento se puede incrementar. Después de una pasada en  $O(n)$ , podremos determinar la subcadena común más larga. Esto ocurre, en la figura 6.6, cuando  $i = 7$ , ya que el sufijo  $SA[7] =$  sufijo 1 = ‘ATAGACA\$CATA#’ (que pertenece a  $T_1$ ) y su sufijo anterior  $SA[6] =$  sufijo 10 = ‘ATA#’ (que pertenece a  $T_2$ ), tienen un prefijo común, que es ‘ATA’. Esta es la subcadena común más larga.

Cerramos la sección, y el capítulo, recordando la disponibilidad de nuestro código fuente. Dedica tiempo a entender ese código, que podría no resultar evidente para aquellos sin conocimiento previo de los *arrays* de sufijos.



ch6\_04\_sa.cpp



ch6\_04\_sa.java

### Ejercicio 6.6.5.1\*

Sugiere posibles mejoras a la función `stringMatching()` que aparece en esta sección.

### Ejercicio 6.6.5.2\*

Compara el algoritmo KMP de la sección 6.4 con la coincidencia de cadenas utilizando un *array* de sufijos. ¿En qué casos resulta más beneficioso utilizar el *array* de sufijos frente a KMP o la funciones de cadenas estándar, al realizar comparaciones de cadenas?

### Ejercicio 6.6.5.3\*

Resuelve todos los ejercicios de las aplicaciones del árbol de sufijos, es decir, los **ejercicios 6.6.3.1, 2, 3\*, 4, 5\* y 6\*** utilizando, en su lugar, un *array* de sufijos.

<sup>17</sup>Si tenemos tres o más cadenas, esta comprobación necesitará más expresiones ‘if’.

## Ejercicios de programación

Ejercicios de programación relacionados con *array* de sufijos<sup>18</sup>:

- |                                        |                                                                                |
|----------------------------------------|--------------------------------------------------------------------------------|
| 1. UVa 00719 - Glass Beads             | (rotación lexicográfica mínima <sup>19</sup> ; construir SA en $O(n \log n)$ ) |
| 2. <b>UVa 00760 - DNA Sequencing *</b> | (subcadena común más larga de dos cadenas)                                     |
| 3. UVa 01223 - Editor                  | (LA 3901 - Seoul07; subcadena repetida más larga (o KMP))                      |
| 4. UVa 01254 - Top 10                  | (LA 4657 - Jakarta09; array de sufijos con árbol de segmentos)                 |
| 5. <b>UVa 11107 - Life Forms *</b>     | (subcadena común más larga de $> \frac{1}{2}$ de las cadenas)                  |
| 6. <b>UVa 11512 - GATTACA *</b>        | (subcadena repetida más larga)                                                 |
| 7. SPOJ 6409 - Suffix Array            | (autor del problema: Felix Halim)                                              |
| 8. IOI 2008 - Type Printer             | (recorrido DFS de <i>trie</i> de sufijos)                                      |

## 6.7 Soluciones a los ejercicios no resaltados

### Soluciones en C para la sección 6.2

#### Ejercicio 6.2.1:

- Se almacena una cadena como un *array* de caracteres terminado por *null*, por ejemplo `char str[30x10+50], line[30+50];`. Es recomendable declarar el *array* con un tamaño un poco mayor del requerido, para evitar el *bug* “pasado por uno”.
- Para leer la entrada línea a línea, utilizamos<sup>20</sup> `gets(line);` o `fgets(line, 40, stdin);`, incluidos en la biblioteca `string.h` (o `cstring`). Hay que recordar que `scanf("%s", line)` no sirve en este caso, ya que solo leerá la primera palabra.
- Establecemos, inicialmente, `strcpy(str, "");` y, a continuación, combinamos las líneas leídas en una cadena más grande, utilizando `strcat(str, line);`. Si la línea actual no es la última, añadimos un espacio en blanco al final de `str`, utilizando `strcat(str, " ");`, para que la última palabra no quede unida a la primera de la siguiente línea.
- Detenemos la lectura de la entrada cuando `strcmp(line, ".....", 7) == 0`. Hay que tener en cuenta que `strcmp` solo compara los primeros *n* caracteres.

#### Ejercicio 6.2.2:

- Para encontrar una subcadena dentro de una cadena relativamente corta (la versión estándar del problema de coincidencia de cadenas), nos basta la función de la biblioteca. Podemos utilizar `p = strstr(str, substr);`. El valor de `p` será `NULL` si `substr` no aparece dentro de `str`.

<sup>18</sup>Puedes intentar resolver estos problemas con un árbol de sufijos, pero tendrás que aprender a programar el algoritmo de construcción del árbol tú mismo. Los problemas de programación que incluimos a continuación, se pueden resolver con un *array* de sufijos. Ten en cuenta, también, que nuestro código de ejemplo utiliza `gets` para leer las cadenas de entrada. Si utilizas `scanf("%s")` o `getline`, no olvides ajustar las diferencias potenciales del ‘fin de línea’ de DOS/UNIX.

<sup>19</sup>Este problema se puede resolver concatenando la cadena consigo misma, construyendo el *array* de sufijos, y buscando el primer sufijo en el orden del *array* de sufijos que tenga una longitud mayor o igual a *s*.

<sup>20</sup>La función `gets` no es segura, porque no realiza comprobaciones de límites en el tamaño de la entrada.

- (b) Si hay varias copias de `substr` dentro de `str`, podemos utilizar `pos = strstr(str+pos, substr)`. Empezamos con `pos = 0`, es decir, buscamos desde el primer carácter de `str`. Una vez hayamos encontrado una aparición de `substr` en `str`, podemos llamar a `pos = strstr(str+pos, substr)` nuevamente, pero esta vez `pos` será el índice de la aparición actual de `substr` en `str más 1`, para obtener la siguiente aparición. Repetiremos el proceso hasta que `pos == NULL`. Esta solución en C requiere entender el direccionamiento de memoria de un `array` en C.

**Ejercicio 6.2.3:** en muchas tareas de procesamiento de cadenas, nos vemos en la necesidad de iterar una vez en todos los caracteres de `str`. Si hay  $n$  caracteres en `str`, esta búsqueda supone  $O(n)$ . Tanto en C como en C++, podemos utilizar `tolower(ch)` y `toupper(ch)`, de `ctype.h`, para convertir un carácter a su versión minúscula o mayúscula. También disponemos de `isalpha(ch)/isdigit(ch)`, para verificar si un carácter determinado es alfabético ([A-Za-z]) o numérico, respectivamente. Para comprobar si un carácter es una vocal, un método puede ser crear una cadena `vocales = "aeiou";` y verificar si está contenido en ella. Si queremos saber si es consonante, basta comprobar que el carácter es alfabético pero no una vocal.

**Ejercicio 6.2.4:** soluciones combinadas en C y C++:

- (a) Una forma muy sencilla de trocear una cadena es con `strtok(str, delimiters);` en C.
- (b) Los segmentos se pueden almacenar en un `vector<string>` `tokens` de C++.
- (c) Podemos utilizar `algorithm::sort` de la STL de C++ para ordenar `vector<string>` `tokens`. Si es necesario, podemos convertir una `string` de C++ a C utilizando `str.c_str()`.

**Ejercicio 6.2.5:** ver la solución en C++.

**Ejercicio 6.2.6:** leer la entrada carácter por carácter y contarlos, buscar la presencia de '\n', que indica el final de línea. Reservar previamente un espacio de almacenamiento de tamaño fijo no es una buena idea, ya que el autor del problema podría utilizar cadenas extraordinariamente largas con el objetivo de hacer fallar el código de las soluciones.

## Soluciones en C++ para la sección 6.2

**Ejercicio 6.2.1:**

- (a) Podemos utilizar la clase `string`.
- (b) Podemos utilizar `cin.getline()` de la biblioteca `string`.
- (c) Podemos utilizar directamente el operador '+' para concatenar cadenas.
- (d) Podemos utilizar directamente el operador '==' para comparar dos cadenas.

**Ejercicio 6.2.2:**

- (a) Podemos utilizar la función `find` de la clase `string`.
- (b) La misma idea que en C. Podemos establecer el valor de desplazamiento en el segundo parámetro de la función `find` de la clase `string`.

**Ejercicio 6.2.3-4:** las mismas soluciones que en C.

**Ejercicio 6.2.5:** podemos utilizar `map<string, int>` de la STL de C++ para mantener un registro de la frecuencia de cada palabra. Cada vez que encontremos un nuevo trozo (que es una cadena), incrementamos en uno la frecuencia correspondiente de ese trozo. Por último, examinamos todos los trozos y determinamos el que tiene mayor frecuencia.

**Ejercicio 6.2.6:** La misma solución que en C.

## Soluciones en Java para la sección 6.2

**Ejercicio 6.2.1:**

- (a) Podemos utilizar las clases `String`, `StringBuffer` o `StringBuilder` (esta es más rápida que `StringBuffer`).
- (b) Podemos utilizar el método `nextLine` de `Scanner` de Java. Para conseguir una E/S más rápida, podemos considerar utilizar el método `readLine` de `BufferedReader` de Java.
- (c) Podemos utilizar el método `append` de `StringBuilder`. No debemos concatenar cadenas en Java con el operador '+', ya que la clase `String` de Java es inmutable y, por ello, esa operación es (muy) costosa.
- (d) Podemos utilizar el método `startsWith` de `String` de Java.

**Ejercicio 6.2.2:**

- (a) Podemos utilizar el método `indexOf` de la clase `String`.
- (b) La misma idea que en C. Podemos establecer el valor de desplazamiento en el segundo parámetro del método `indexOf` de la clase `String`.

**Ejercicio 6.2.3:** utilizar las clases `StringBuilder` y `Character` de Java para estas operaciones.

**Ejercicio 6.2.4:**

- (a) Podemos utilizar la clase  `StringTokenizer` o el método `split` de la clase `String` de Java.
- (b) Podemos utilizar un `Vector` de `String` de Java.
- (c) Podemos utilizar `Collections.sort` de Java.

**Ejercicio 6.2.5:** la misma idea que en C++. Podemos utilizar `TreeMap<String, Integer>`.

**Ejercicio 6.2.6:** necesitamos utilizar el método `read` de la clase `BufferedReader` de Java.

## Soluciones para el resto de secciones

**Exercise 6.5.1.1:** un esquema de puntuación diferente producirá una alineación (global) diferente. Si te ves ante un problema de alineación de cadenas, lee el enunciado y determina cuál

es el coste necesario para la coincidencia, no coincidencia, inserción y eliminación. Adapta el algoritmo de acuerdo a eso.

**Ejercicio 6.5.1.2:** tienes que guardar la información del precedente (las flechas) durante el cálculo de DP. Después, sigue las flechas utilizando *bactracking* recursivo. Ver la sección 3.5.1.

**Ejercicio 6.5.1.3:** la solución de DP solo necesita referirse a la fila anterior, por lo que puede utilizar el ‘truco de ahorro de espacio’ y emplear, únicamente, dos filas, la actual y la anterior. La nueva complejidad de espacio será de solo  $O(\min(n, m))$ , esto es, coloca la cadena con menor longitud como cadena 2, de forma que cada fila tenga menos columnas (menos memoria). La complejidad de tiempo de esta solución sigue siendo  $O(nm)$ . El único inconveniente de esta técnica, al igual que con cualquier otro truco de ahorro de espacio, es que no podremos reconstruir la solución óptima. Así que, si ésta es necesaria, no podemos ahorrar espacio. Ver la sección 3.5.1.

**Ejercicio 6.5.1.4:** basta concentrarse a lo largo de la diagonal principal con ancho  $d$ . Al hacerlo, podemos acelerar el algoritmo de Needleman-Wunsch hasta  $O(dn)$ .

**Ejercicio 6.5.1.5:** implica nuevamente el algoritmo de Kadane (ver el problema de la suma máxima en la sección 3.5.2).

**Ejercicio 6.5.2.1:** ‘pple’.

**Ejercicio 6.5.2.2:** establecer la puntuación para coincidencia = 0, no coincidencia = 1, inserción y eliminación = infinito negativo. Sin embargo, la solución no es ni eficiente ni natural, ya que podemos utilizar, simplemente, un algoritmo  $O(\min(n, m))$  para explorar ambas cadenas y contar el número de caracteres diferentes.

**Ejercicio 6.5.2.3:** reducido a LIS, solución en  $O(n \log k)$ . No mostramos la reducción a LIS. Dibújalo y comprueba cómo se puede reducir este problema a LIS.

**Ejercicio 6.6.3.1:** ‘CA’ se encuentra, ‘CAT’ no.

**Ejercicio 6.6.3.2:** ‘ACATTA’.

**Ejercicio 6.6.3.4:** ‘EVEN’.

## 6.8 Notas del capítulo

El material sobre alineación de cadenas (distancia de edición), subsecuencia común más larga y *trie*/árbol/*array* de sufijos pertenece originalmente a **Sung Wing Kin, Ken** [62], de la Escuela de Informática de la Universidad Nacional de Singapur. Desde entonces, el contenido ha evolucionado desde un estilo más teórico al de programación competitiva actual.

La sección sobre habilidades básicas de procesamiento de cadenas (sección 6.2) y los problemas de procesamiento de cadenas *ad hoc*, nacen de nuestra experiencia con problemas y técnicas relativos a cadenas. El número de ejercicios de programación mencionado aquí supone, más o menos, un cuarto de todos los problemas de procesamiento de cadenas tratados en este capítulo. Somos conscientes de que no son problemas del ICPC, o tareas de la IOI, típicos, pero siguen siendo buenos ejercicios de programación para mejorar la capacidad de escribir código.

En la sección 6.4, tratamos las soluciones con bibliotecas y un algoritmo rápido (el de Knuth-Morris-Pratt/KMP) para el problema de coincidencia de cadenas. La implementación del KMP resultará útil si tienes que modificar elementos de coincidencia de cadenas básicos y sigues necesitando un rendimiento rápido. Consideramos que KMP es lo suficientemente veloz como para encontrar una cadena patrón, dentro de una cadena larga en un problema típico de un concurso. De forma empírica, podemos concluir que la implementación del KMP mostrada en este libro es un poco más rápida que las integradas en `strstr` de C, `string.find` de C++ y `String.indexOf` de Java. Si se necesita un algoritmo de coincidencia de cadenas todavía más rápido, durante un concurso, para una cadena más larga y muchas más consultas, sugerimos utilizar el *array* de sufijos tratado en la sección 6.8. Hay otros algoritmos de coincidencia de cadenas que no hemos visto, como los de **Boyer-Moore**, **Rabin-Karp**, **Aho-Corasick**, **autómata de estados finitos**, etc. El lector interesado debería conocerlos.

Hemos extendido la sección sobre problemas de DP *no clásicos*, que implican cadenas, en la sección 6.5. Creemos que los problemas clásicos ya prácticamente no aparecerán en los concursos de programación modernos.

La implementación práctica del *array* de sufijos (sección 6.6) está inspirada, principalmente, en el artículo “Suffix arrays - a programming contest approach” de [68]. Hemos integrado y sincronizado muchos de los ejemplos que aparecen ahí con nuestro estilo de escritura de la implementación del *array* de sufijos. En la presente edición de este libro, hemos reincorporado el concepto de carácter de terminación en los *arrays* y árboles de sufijos, ya que simplifica el tratamiento. Es una buena idea resolver *todos* los ejercicios de programación incluidos en la sección 6.6, aunque, *de momento*, no haya muchos. Hablamos de una estructura de datos importante que será más popular en un futuro cercano.

Aunque consideramos que hemos cubierto una parte importante de la materia, sigue habiendo otros problemas de procesamiento de cadenas que no hemos mencionado: **técnicas de hashing** para resolver algunos problemas de procesamiento de cadenas, el problema de la **supercadena común más corta**, el algoritmo de la **transformación de Burrows-Wheeler**, **autómata de sufijos**, **árbol de radix**, etc.

# Capítulo 7

---

## Geometría (computacional)

*Que ningún hombre ignorante de la geometría entre aquí.*

— Academia de Platón en Atenas

### 7.1 Introducción y motivación

La geometría (computacional<sup>1</sup>) es otro de los temas recurrentes en los concursos de programación. Casi todos los conjuntos de problemas del ICPC incluyen, *al menos*, un problema de geometría. Si hay suerte, se te pedirá alguna solución geométrica que ya conozcas. Normalmente, dibujas los objetos geométricos y deduces la solución a partir de algunas fórmulas básicas. Sin embargo, muchos de ellos son de tipo *computacional*, y requieren algoritmos complejos.

En la IOI, la aparición de problemas específicos de geometría depende de las tareas seleccionadas por el comité científico de cada año. En los más recientes (2009-2012), las tareas de la IOI no han incluido problemas específicos de geometría *pura*. Sin embargo, en años anteriores [67], cada IOI presentaba uno o dos problemas relacionados con la materia.

Hemos observado que los problemas relacionados con la geometría no se suelen abordar durante las fases tempranas del concurso por *razones estratégicas*, ya que sus soluciones tienen *menos* probabilidades de obtener un veredicto de aceptado (AC), frente a las soluciones de otros tipos de problemas como, por ejemplo, los de búsqueda completa o programación dinámica. Los inconvenientes más habituales con problemas de geometría suelen ser:

- Muchos problemas de geometría tiene uno y, normalmente, varios ‘casos límite’ complicados. Por ejemplo, ¿qué ocurre si las líneas son verticales (gradiente infinito)? ¿O los puntos son colineales? ¿O si el polígono es cóncavo? ¿Y si la envolvente convexa de un conjunto de puntos es el propio conjunto? Así, normalmente es una buena idea que el equipo compruebe sus soluciones con muchos casos límite, antes de enviar la solución.
- Existe la posibilidad de sufrir errores de precisión de coma flotante, que pueden provocar que un algoritmo ‘correcto’ obtenga un veredicto de respuesta incorrecta (WA).
- Las soluciones a los problemas de geometría suelen implicar una programación *tediosa*.

---

<sup>1</sup>Diferenciamos entre problemas de geometría *pura* y de geometría *computacional*. Normalmente, los problemas de geometría pura se pueden resolver a mano (con lápiz y papel). Los problemas de geometría computacional suelen necesitar la ejecución de un algoritmo informático para llegar a la solución.

Estas razones, provocan que muchos concursantes sientan que utilizar un tiempo precioso en intentar resolver *otros* tipos de problemas, merezca más la pena que afrontar un problema de geometría, que tiene una probabilidad de aceptación menor.

Sin embargo, otra razón más preocupante, que justifica la falta de intentos de resolución de problemas de geometría, es que los concursantes no están bien preparados.

- Los concursantes olvidan algunas de las fórmulas básicas importantes, o no son capaces de deducir las más complejas, a partir de las primeras.
- Los concursantes no preparan correctamente funciones de biblioteca bien escritas, *antes* del concurso, y sus intentos de programar esas funciones en el ámbito, necesariamente estresante, del concurso, introduce uno, o varios<sup>2</sup>, errores. Los mejores equipos del ICPC suelen utilizar una parte muy importante de sus notas escritas (que pueden consultar durante el concurso) con muchas fórmulas y funciones de biblioteca sobre geometría.

El objetivo principal de este capítulo es, por lo tanto, aumentar el número de intentos (y soluciones correctas) de los problemas relacionados con la geometría, en los concursos de programación. Estudia cómo extraer algunas ideas para afrontar problemas de geometría (computacional) en los ICPC y las IOI. Este capítulo consta de solo dos secciones.

En la sección 7.2, presentamos mucha (pero no toda) terminología sobre geometría<sup>3</sup> y varias fórmulas básicas para **objetos geométricos** de 0, 1, 2 y 3 dimensiones, que suelen encontrarse en concursos de programación. Esta sección se puede utilizar como una guía de referencia rápida, cuando los concursantes se enfrenten a problemas de geometría y no estén seguros del sentido de alguno de los términos utilizados, o hayan olvidado algunas fórmulas básicas.

En la sección 7.3, tratamos algoritmos sobre **polígonos** bidimensionales. Encontrarás varias rutinas de biblioteca, ya escritas, que pueden marcar la diferencia entre los mejores equipos (concursantes) y los que están en la media, como las que determinan si un polígono es cóncavo o convexo, que deciden si un punto está dentro o fuera del polígono, que cortan un polígono con una línea recta, que encuentran la envolvente convexa de un conjunto de puntos, etc.

La implementación de las fórmulas y algoritmos de geometría computacional, que incluimos en este capítulo, utiliza las siguientes técnicas para aumentar la probabilidad de obtener un veredicto de aceptado:

1. Resaltamos los casos potencialmente especiales, que pueden surgir, y/o elegimos la implementación que reduce el número de estos casos especiales.
2. Tratamos de evitar operaciones de coma flotante (es decir, división, raíz cuadrada y cualquier otra operación que pueda provocar errores numéricos) y trabajamos con enteros precisos, siempre que es posible (es decir, suma, resta y multiplicación de enteros).
3. Si, necesariamente, tenemos que trabajar con coma flotante, realizamos comprobaciones de igualdad de coma flotante de la siguiente manera: `fabs(a-b) < EPS`, donde EPS es un

<sup>2</sup>Como referencia, el código de biblioteca sobre puntos, líneas, círculos, triángulos y polígonos, mostrado en este capítulo, requiere de varias iteraciones de corrección de errores, para asegurar que muchos de ellos (normalmente muy sutiles), así como casos especiales, son tratados correctamente.

<sup>3</sup>Los concursantes del ICPC y de la IOI proceden de muy diversas nacionalidades y culturas, por lo que es importante que todos compartan una terminología común.

número pequeño<sup>4</sup> como  $1e-9$  (es decir,  $10^{-9}$  o  $0.000000001$ ), en vez de comprobar si  $a == b$ . Cuando tengamos que verificar si un número de coma flotante  $x \geq 0,0$ , utilizaremos  $x > -\text{EPS}$  (igualmente, para comprobar si  $x \leq 0,0$ , utilizaremos  $x < \text{EPS}$ ). También utilizamos tipos de datos de precisión doble, en vez de los de precisión sencilla.

## 7.2 Objetos de geometría básicos con bibliotecas

### 7.2.1 Objetos sin dimensión: puntos

- El **punto** es el elemento básico de construcción de objetos geométricos de más dimensiones. En el espacio euclídeo bidimensional<sup>5</sup>, los puntos se suelen representar como una estructura en C/C++ (o una clase en Java) con dos<sup>6</sup> miembros: las coordenadas  $(x, y)$ , en relación al origen, es decir, la coordenada  $(0, 0)$ . Si el enunciado del problema utiliza coordenadas con números enteros, utilizaremos **int**, en caso contrario, será **double**. Para no alejarnos del generalismo, en este libro utilizaremos la versión de coma flotante de **struct point**.

```

1 // struct point_i { int x, y; }; // forma básica, modo minimalista
2 struct point_i { int x, y; // cuando sea posible, trabajar con point_i
3 point_i() { x = y = 0; } // constructor predeterminado
4 point_i(int _x, int _y) : x(_x), y(_y) {} }; // definido por usuario
5
6 struct point { double x, y; // usar si la coma flotante es inevitable
7 point() { x = y = 0.0; } // constructor predeterminado
8 point(double _x, double _y) : x(_x), y(_y) {} }; // def. por usuario

```

- En ocasiones, necesitamos ordenar los puntos. Podemos hacerlo fácilmente, sobrecargando el operador ‘menor que’ dentro de **struct point** y utilizando la biblioteca de ordenación.

```

1 struct point { double x, y;
2 point() { x = y = 0.0; }
3 point(double _x, double _y) : x(_x), y(_y) {}
4 bool operator < (point other) const //sobrecarga operador menor que
5 if (fabs(x-other.x) > EPS) // útil para ordenación
6 return x < other.x; // primer criterio, por coordenada x
7 return y < other.y; }; // segundo criterio, por coordenada y
8 // en int main(), asumiendo que vector<point> P ha sido poblado
9 sort(P.begin(), P.end()); // operador de comparación definido antes

```

- Otra veces, necesitaremos comprobar si dos puntos son iguales. También es fácil de lograr, sobrecargando el operador ‘igual que’, dentro de **struct point**. Esta prueba es más sencilla en la versión para enteros (**struct point\_i**).

<sup>4</sup>Salvo que se indique lo contrario, el mencionado  $1e-9$  será el valor por defecto que utilizaremos en este capítulo para EPS(ilon).

<sup>5</sup>Por simplificar, los espacios euclídeos de dos y tres dimensiones se corresponden al mundo bidimensional y tridimensional de la vida real.

<sup>6</sup>Añadiremos un tercer miembro,  $z$ , si trabajamos sobre un espacio euclídeo tridimensional.

```

1 struct point { double x, y;
2 .. // igual que antes
3 // usar EPS (1e-9) para igualdad entre dos números de coma flotante
4 bool operator == (point other) const {
5 return (fabs(x-other.x) < EPS && (fabs(y-other.y) < EPS)); } };
6
7 // en int main()
8 point P1(0, 0), P2(0, 0), P3(0, 1);
9 printf("%d\n", P1 == P2); // verdadero
10 printf("%d\n", P1 == P3); // falso

```

4. Podemos medir la distancia euclídea<sup>7</sup> entre dos puntos, utilizando la siguiente función:

```

1 double dist(point p1, point p2) { // distancia euclídea
2 // hypot(dx, dy) devuelve sqrt(dx*dx + dy*dy)
3 return hypot(p1.x-p2.x, p1.y-p2.y); } // devuelve double

```

5. Podemos rotar un punto por un ángulo<sup>8</sup>  $\theta$  en sentido inverso a las agujas del reloj, alrededor del origen, mediante una matriz de rotación:

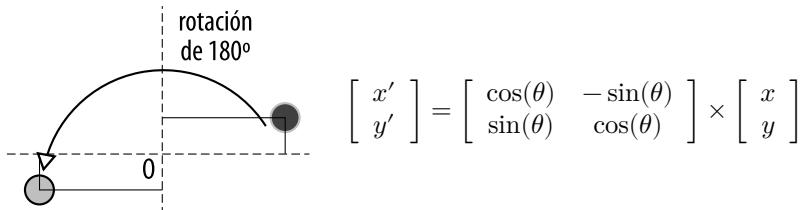


Figura 7.1: Rotación de (10, 3) 180 grados contra las agujas del reloj, alrededor de (0, 0)

```

1 // rotar p por theta grados contra el reloj respecto al origen (0, 0)
2 point rotate(point p, double theta) {
3 double rad = DEG_to_RAD(theta); // multiplicar theta por PI/180.0
4 return point(p.x*cos(rad) - p.y*sin(rad),
5 p.x*sin(rad) + p.y*cos(rad)); }

```

### Ejercicio 7.2.1.1

Calcula la distancia euclídea entre los puntos (2, 2) y (6, 5).

<sup>7</sup>La distancia euclídea entre dos puntos no es más que la distancia que se podría medir con una regla. En términos de algorítmica, se puede hallar utilizando el teorema de Pitágoras, que volveremos a ver en la subsección sobre triángulos. Aquí, simplemente, utilizamos una función de la biblioteca.

<sup>8</sup>Los humanos, normalmente, trabajamos con grados, pero muchas funciones matemáticas, en los lenguajes de programación más comunes (C/C++/Java), lo hacen con radianes. Para convertir de grados a radianes, multiplicamos el ángulo por  $\frac{\pi}{180.0}$ . Para convertir de radianes a grados, multiplicamos el ángulo por  $\frac{180.0}{\pi}$ .

### Ejercicio 7.2.1.2

Rota 90 grados en sentido contrario a las agujas del reloj un punto  $(10, 3)$ , alrededor del origen. ¿Cuál es la nueva coordenada del punto, una vez rotado?

### Ejercicio 7.2.1.3

Rota el mismo punto  $(10, 3)$  77 grados en sentido contrario a las agujas del reloj, alrededor del origen. ¿Cuál es la nueva coordenada del punto, una vez rotado? (Esta vez, tendrás que utilizar una calculadora y la matriz de rotación).

## 7.2.2 Objetos unidimensionales: líneas

- La **línea**, en el espacio euclídeo bidimensional, es un conjunto de puntos cuyas coordenadas satisfacen la ecuación lineal  $ax + by + c = 0$ . Las siguientes funciones de esta subsección dan por echo que esta ecuación lineal tiene  $b = 1$ , para las líneas no verticales, y  $b = 0$ , para las verticales, salvo que se indique lo contrario. Las líneas se suelen representar con una estructura en C/C++ (o una clase en Java) de tres miembros: los coeficientes  $a$ ,  $b$  y  $c$  de la ecuación.

```
1 struct line { double a, b, c; }; // representación de una línea
```

- Podemos calcular la ecuación de la línea correspondiente, si nos dan *al menos* dos de los puntos que pertenecen a esa línea, mediante la siguiente función:

```
1 // la respuesta se almacena en el tercer parámetro (pasa por referencia)
2 void pointsToLine(point p1, point p2, line &l) {
3 if (fabs(p1.x-p2.x) < EPS) { // una línea vertical es correcta
4 l.a = 1.0; l.b = 0.0; l.c = -p1.x; // valores predeterminados
5 } else {
6 l.a = -(double)(p1.y-p2.y) / (p1.x-p2.x);
7 l.b = 1.0; // IMPORTANTE: fijamos el valor de b a 1.0
8 l.c = -(double)(l.a*p1.x) - p1.y;
9 } }
```

- Podemos comprobar si dos líneas son *paralelas* verificando si sus coeficientes  $a$  y  $b$  son iguales. También podemos ir más allá y comprobar si dos líneas son *la misma*, verificando si son paralelas y su coeficiente  $c$  es igual (es decir, los tres coeficientes  $a$ ,  $b$  y  $c$  son iguales). Recuerda que, en nuestra implementación, hemos establecido que el valor del coeficiente  $b$  sea 0,0, para todas las líneas verticales, y 1,0 para todas las que *no lo son*.

```
1 bool areParallel(line l1, line l2) { // comprobar coeficientes a & b
2 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
```

```

1 bool areSame(line l1, line l2) { // comprobar también coeficiente c
2 return areParallel(l1, l2) && (fabs(l1.c-l2.c) < EPS); }

```

4. Si dos líneas<sup>9</sup> no son paralelas (y no son la misma), tendrán una intersección en un punto. Es posible hallar ese punto de intersección  $(x, y)$  resolviendo un sistema de dos ecuaciones algebraicas lineales<sup>10</sup> con dos incógnitas:  $a_1x + b_1y + c_1 = 0$  y  $a_2x + b_2y + c_2 = 0$ .

```

1 // devuelve verdadero y punto intersección p si 2 líneas interseccionan
2 bool areIntersect(line l1, line l2, point &p) {
3 if (areParallel(l1, l2)) return false; // no hay intersección
4 // resolver sistema de 2 ecuaciones algebraicas con 2 incognitas
5 p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);
6 // caso especial: comprobar línea vertical para evitar división por 0
7 if (fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1.c);
8 else p.y = -(l2.a*p.x + l2.c);
9 return true; }

```

5. Un **segmento de una línea** es una línea con dos extremos y *longitud finita*.
6. Un **vector**<sup>11</sup> es un segmento (por lo que tiene dos extremos y una longitud/magnitud) con una *dirección*. Normalmente<sup>12</sup>, los vectores se representan con una estructura de C/C++ (o clase en Java) con dos miembros: las magnitudes  $x$  e  $y$  del vector. Es posible aplicar un escalar a la magnitud de un vector, si es necesario.
7. Podemos trasladar (mover) un punto en relación a un vector, pues los vectores describen magnitudes de desplazamiento en los ejes  $x$  e  $y$ .

```

1 struct vec { double x, y; // 'vec' es diferente del vector de la STL
2 vec(double _x, double _y) : x(_x), y(_y) {} };
3
4 vec toVec(point a, point b) { // convertir 2 puntos a vector a->b
5 return vec(b.x-a.x, b.y-a.y); }
6
7 vec scale(vec v, double s) { // s no negativo = [<1 .. 1 .. >1]
8 return vec(v.x*s, v.y*s); } // más corto, igual, más largo
9
10 point translate(point p, vec v) { // trasladar p según v
11 return point(p.x+v.x, p.y+v.y); }

```

8. Dados un punto  $p$  y una línea  $l$  (descrita por dos puntos  $a$  y  $b$ ), podemos obtener la distancia mínima de  $p$  a  $l$ , calculando primero la ubicación del punto  $c$  de  $l$  más cercano al punto  $p$  (ver la parte izquierda de la figura 7.2) y, después, obtener la distancia euclídea entre  $p$  y  $c$ . Se puede entender el punto  $c$  como el punto  $a$ , trasladado por una

<sup>9</sup>Para evitar confusiones, debemos diferenciar entre intersección de líneas (infinitas) e intersección de *segmentos* (finitos), que veremos después.

<sup>10</sup>Consulta la sección 9.9 para ver la solución generalista a un sistema de ecuaciones lineales.

<sup>11</sup>No confundir con el vector de la STL de C++ o el Vector de Java.

<sup>12</sup>Otra estrategia de diseño potencial consiste en combinar `struct point` con `struct vec`, ya que son similares.

magnitud modificada por un escalar  $u$ , del vector  $ab$ , o  $c = a + u \times ab$ . Para obtener  $u$ , realizamos la proyección escalar del vector  $ap$  sobre el vector  $ab$ , utilizando el producto escalar (ver el vector punteado  $ac = u \times ab$ , en la parte izquierda de la figura 7.2). La breve implementación de esta solución se reproduce a continuación.

```

1 double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }
2
3 double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }
4
5 // devuelve la distancia de p a la línea definida por
6 // dos puntos a y b (que deben ser diferentes)
7 // el punto más cercano se almacena en el cuarto parámetro (por ref.)
8 double distToLine(point p, point a, point b, point &c) {
9 // fórmula: c = a + u*ab
10 vec ap = toVec(a, p), ab = toVec(a, b);
11 double u = dot(ap, ab) / norm_sq(ab);
12 c = translate(a, scale(ab, u)); // trasladar a hasta c
13 return dist(p, c); } // distancia euclídea entre p y c

```

Ten en cuenta que esta no es la única forma de obtener la respuesta pedida. Resuelve el **ejercicio 7.2.2.11**, para conocer un método alternativo.

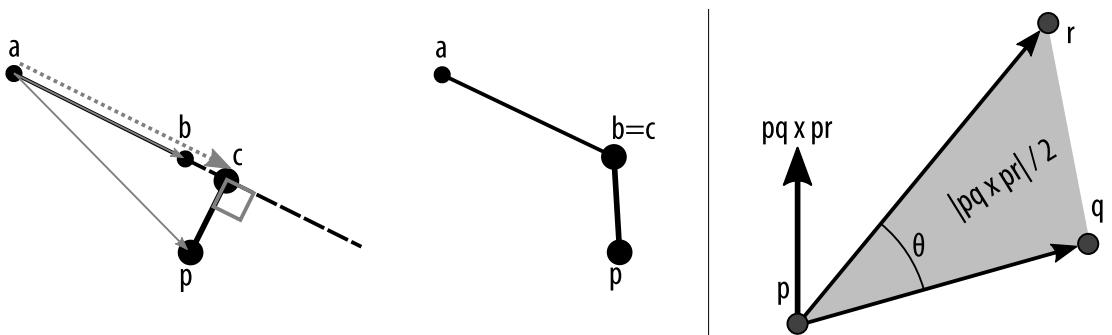


Figura 7.2: Distancia a la línea (izquierda) y al segmento (centro); producto vectorial (derecha)

9. Si, en su lugar, nos dan un *segmento* (definido por dos *extremos*  $a$  y  $b$ ), la distancia mínima desde el punto  $p$  al segmento  $ab$  deberá considerar también dos casos especiales, los extremos  $a$  y  $b$  de ese segmento (ver la parte central de la figura 7.2). La implementación es muy similar a la anterior de `distToLine`.

```

1 // devuelve la distancia de p al segmento ab definido por
2 // dos puntos a y b (técnicamente, deben ser diferentes)
3 // el punto más cercano se almacena en el cuarto parámetro (por ref.)
4 double distToLineSegment(point p, point a, point b, point &c) {
5 vec ap = toVec(a, p), ab = toVec(a, b);
6 double u = dot(ap, ab) / norm_sq(ab);
7 if (u < 0.0) { c = point(a.x, a.y); // más cerca de a

```

```

8 return dist(p, a); } // distancia euclídea entre p y a
9 if (u > 1.0) { c = point(b.x, b.y); } // más cerca de b
10 return dist(p, b); } // distancia euclídea entre p y b
11 return distToLine(p, a, b, c); } // ejecutar distToLine como antes

```

10. Podemos calcular el ángulo  $aob$ , dados tres puntos:  $a$ ,  $o$  y  $b$ , utilizando el producto escalar. Como  $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$ , tenemos<sup>13</sup>  $\theta = \arccos(oa \cdot ob / (|oa| \times |ob|))$ .

```

1 double angle(point a, point o, point b) { // devuelve ángulo aob en rad
2 vec oa = toVector(o, a), ob = toVector(o, b);
3 return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

```

11. Dada una línea, definida por dos puntos  $p$  y  $q$ , podemos determinar si un punto  $r$  se encuentra a la izquierda o a la derecha de dicha línea, o si los tres puntos  $p$ ,  $q$  y  $r$  son colineales. Esto se puede determinar mediante el *producto vectorial*. Digamos que  $pq$  y  $pr$  son dos vectores, obtenidos de estos tres puntos. El producto vectorial  $pq \times pr$  resulta en otro vector, que es perpendicular a ambos  $pq$  y  $pr$ . La magnitud de este vector es igual al área del *paralelogramo* que forman los vectores<sup>14</sup>. Si la magnitud es positiva/cero/negativa, sabremos que  $p \rightarrow q \rightarrow r$  es un giro a la izquierda/colineal/un giro a la derecha, respectivamente (ver la parte derecha de la figura 7.2).

```

1 double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }
2
3 // nota: para aceptar puntos colineales hay que cambiar el '> 0'
4 // devuelve verdadero si el punto r está a la izquierda de la línea pq
5 bool ccw(point p, point q, point r) {
6 return cross(toVec(p, q), toVec(p, r)) > 0; }
7
8 // devuelve verdadero si el punto r está en la misma línea que pq
9 bool collinear(point p, point q, point r) {
10 return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

```



ch7\_01\_points\_lines.cpp



ch7\_01\_points\_lines.java

### Ejercicio 7.2.2.1

Una línea también se puede describir mediante la siguiente ecuación:  $y = mx + c$ , donde  $m$  es el ‘gradiente’/‘pendiente’ de la línea y  $c$  la constante ‘intersección con  $y$ ’. ¿Qué método es mejor ( $ax + by + c = 0$  o el de pendiente-intersección  $y = mx + c$ )? ¿Por qué?

<sup>13</sup>acos es el nombre de la función de C/C++ para la función matemática arccos.

<sup>14</sup>El área del triángulo  $pqr$  será, por tanto, la mitad del área de este paralelogramo.

### Ejercicio 7.2.2.2

Calcula la ecuación de la línea que pasa por los dos puntos  $(2, 2)$  y  $(4, 3)$ .

### Ejercicio 7.2.2.3

Calcula la ecuación de la línea que pasa por los dos puntos  $(2, 2)$  y  $(2, 4)$ .

### Ejercicio 7.2.2.4

Supón que insistimos en utilizar la otra ecuación de la línea:  $y = mx + c$ . Muestra cómo calcular esa ecuación, dados dos puntos que formen parte de la línea. Prueba con  $(2, 2)$  y  $(2, 4)$ , como en el **ejercicio 7.2.2.3**. ¿Algún problema?

### Ejercicio 7.2.2.5

También podemos calcular la ecuación de una línea si nos dan *un* punto y la pendiente. Muestra cómo calcular la ecuación de una línea a partir de esos datos.

### Ejercicio 7.2.2.6

Traslada un punto  $c (3, 2)$ , según un vector  $ab$  definido por dos puntos:  $a (2, 2)$  y  $b (4, 3)$ . ¿Cuál es la nueva coordenada del punto?

### Ejercicio 7.2.2.7

Igual que el **ejercicio 7.2.2.6**, pero ahora la magnitud del vector  $ab$  se reduce a la mitad. ¿Cuál es la nueva coordenada del punto?

### Ejercicio 7.2.2.8

Igual que el **ejercicio 7.2.2.6**, pero después rota el punto 90 grados en sentido contrario a las agujas del reloj, alrededor de origen. ¿Cuál es la nueva coordenada del punto?

### Ejercicio 7.2.2.9

Rota un punto  $c$   $(3, 2)$  90 grados en sentido contrario a las agujas del reloj, alrededor del origen, después traslada el punto resultante según un vector  $ab$ . Ese vector  $ab$  es el mismo que el del **ejercicio 7.2.2.6**. ¿Cuál es la nueva coordenada del punto? ¿Es el resultado similar al del **ejercicio 7.2.2.8**? ¿Qué podemos aprender de este fenómeno?

### Ejercicio 7.2.2.10

Rota un punto  $c$   $(3, 2)$  90 grados en sentido contrario a las agujas del reloj, pero alrededor del punto  $p$   $(2, 1)$  (ten en cuenta que el punto  $p$  *no es* el origen). Consejo: necesitas trasladar el punto.

### Ejercicio 7.2.2.11

Podemos calcular la ubicación del punto  $c$  en la línea  $l$  que esté más cerca del punto  $p$ , buscando la otra línea  $l'$  que sea perpendicular a  $l$  y pase por el punto  $p$ . El punto más cercano  $c$  es el de intersección entre las líneas  $l$  y  $l'$ . Pero, ¿cómo logramos una línea perpendicular a  $l$ ? ¿Hay algún caso especial con el que debamos ser cuidadosos?

### Ejercicio 7.2.2.12

Dados un punto  $p$  y una línea  $l$  (descrita por dos puntos  $a$  y  $b$ ), calcula la ubicación del punto simétrico  $r$  de  $p$ , cuando se refleja a lo largo de la línea  $l$ .

### Ejercicio 7.2.2.13

Dados tres puntos:  $a$   $(2, 2)$ ,  $o$   $(2, 4)$  y  $b$   $(4, 3)$ , calcula el ángulo  $aob$  en grados.

### Ejercicio 7.2.2.14

Determina si el punto  $r$   $(35, 30)$  está a la izquierda, es colineal, o está a la derecha de una línea que pasa por los puntos  $p$   $(3, 7)$  y  $q$   $(11, 13)$ .

### 7.2.3 Objetos bidimensionales: círculos

1. Un **círculo** centrado en la coordenada  $(a, b)$  en un espacio euclídeo bidimensional, con **radio**  $r$ , es el conjunto de todos los puntos  $(x, y)$  de forma que  $(x - a)^2 + (y - b)^2 = r^2$ .
2. Para comprobar si un punto está dentro, fuera, o exactamente en la circunferencia de un círculo, podemos utilizar la siguiente función. Modifícalo un poco para tener una versión de coma flotante.

```

1 int insideCircle(point_i p, point_i c, int r) { // versión para enteros
2 int dx = p.x-c.x, dy = p.y-c.y;
3 int Euc = dx*dx + dy*dy, rSq = r*r; // todos son enteros
4 return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } // dentro/borde/fuera

```

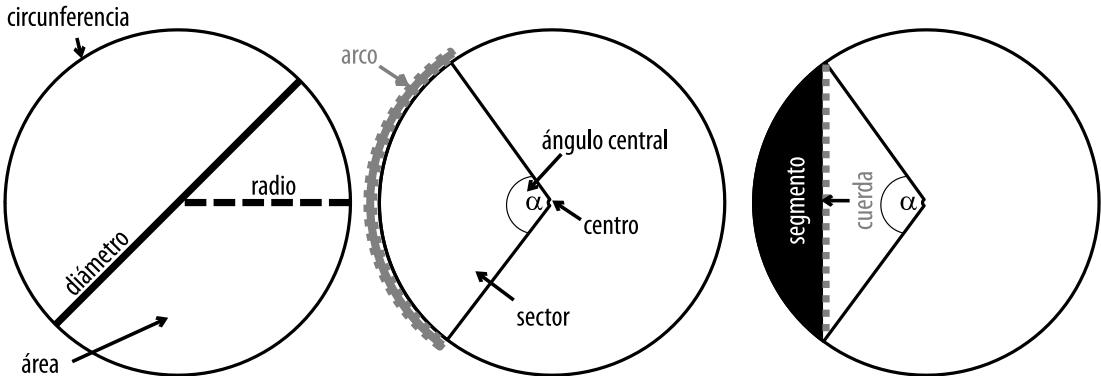


Figura 7.3: Círculos

3. La constante **Pi** ( $\pi$ ) es la razón entre la circunferencia de *cualquier* círculo y su diámetro. Para evitar errores de precisión, el valor más seguro en un concurso de programación, si la constante  $\pi$  no queda definida en el enunciado del problema, es  $\text{pi} = \text{acos}(-1.0)$  o  $\text{pi} = 2 * \text{acos}(0.0)$ .
4. Un círculo, con radio  $r$ , tiene un **diámetro**  $d = 2 \times r$  y una **circunferencia** (o **perímetro**)  $c = 2 \times \pi \times r$ .
5. Un círculo de radio  $r$  tiene un **área**  $A = \pi \times r^2$ .
6. El **arco** de un círculo se define como una sección conexa de la circunferencia  $c$  del círculo. Dado el ángulo central  $\alpha$  (un ángulo cuyo vértice está en el centro del círculo, ver la parte central de la figura 7.3) en grados, podemos calcular la longitud del arco correspondiente como  $\frac{\alpha}{360,0} \times c$ .

7. La **cuerda** de un círculo se define como el segmento cuyos extremos están dentro del círculo<sup>15</sup>. Un círculo con radio  $r$  y un ángulo central  $\alpha$  en grados (ver la parte derecha de la figura 7.3), tiene su cuerda correspondiente de longitud  $\sqrt{2 \times r^2 \times (1 - \cos(\alpha))}$ . Esto se puede deducir del teorema del coseno (ver la explicación de este teorema, en la sección que trata sobre triángulos). Otro método para calcular la longitud de la cuerda, dados  $r$  y  $\alpha$ , es utilizando trigonometría:  $2 \times r \times \sin(\alpha/2)$ . También hablaremos de trigonometría más adelante.
8. El **sector** de un círculo se define como una región del mismo, delimitada por dos radios y un arco situado entre ellos. Un círculo de área  $A$  y un ángulo central  $\alpha$  en grados (ver la parte central de la figura 7.3), tiene un sector de área  $\frac{\alpha}{360,0} \times A$ .
9. El **segmento** de un círculo se define como una región del círculo delimitada por una cuerda y un arco entre los extremos de ella (ver la parte derecha de la figura 7.3). El área de un segmento se puede calcular restando el área del sector correspondiente al área de un triángulo isósceles de lados  $r$ ,  $r$  y longitud de la cuerda.
10. Dados 2 puntos de un círculo ( $p1$  y  $p2$ ) y el radio  $r$  correspondiente, podemos determinar la ubicación de los centros ( $c1$  y  $c2$ ) de los dos posibles círculos (ver la figura 7.4). El código se incluye en el **ejercicio 7.2.3.1**.

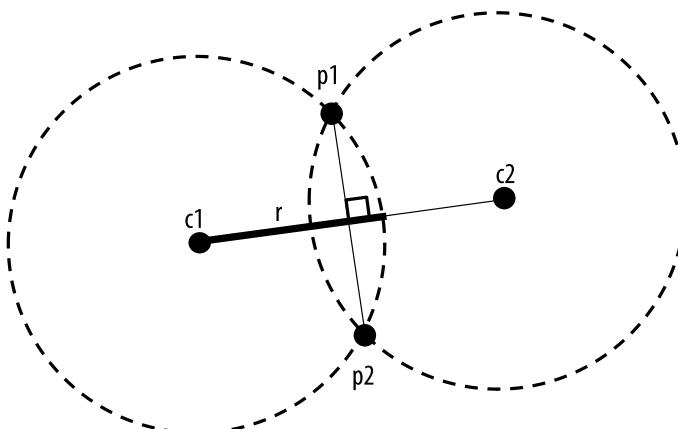


Figura 7.4: Un círculo por dos puntos y un radio



`ch7_02_circles.cpp`



`ch7_02_circles.java`

---

<sup>15</sup>El diámetro es la cuerda más larga de un círculo.

### Ejercicio 7.2.3.1

Explica lo que calcula el siguiente código.

```
bool circle2PtsRad(point p1, point p2, double r, point &c) {
 double d2 = (p1.x-p2.x) * (p1.x-p2.x) + (p1.y-p2.y) * (p1.y-p2.y);
 double det = r*r/d2 - 0.25;
 if (det < 0.0) return false;
 double h = sqrt(det);
 c.x = (p1.x+p2.x) * 0.5 + (p1.y-p2.y) * h;
 c.y = (p1.y+p2.y) * 0.5 + (p2.x-p1.x) * h;
 return true; } // para obtener el otro centro, invertir p1 y p2
```

#### 7.2.4 Objetos bidimensionales: triángulos

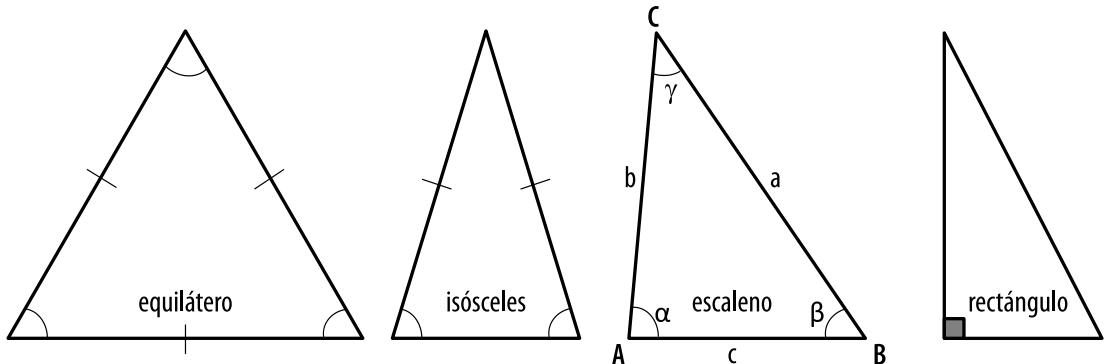


Figura 7.5: Triángulos

1. El **triángulo** (tres ángulos) es un polígono formado por tres vértices y tres aristas. Existen varios tipos de triángulos:
  - a. **Equilátero:** tres aristas de igual longitud y todos los ángulos interiores de 60 grados.
  - b. **Isósceles:** dos aristas tienen la misma longitud y dos ángulos interiores son iguales.
  - c. **Escaleno:** todas las aristas son de diferente longitud.
  - d. **Rectángulo:** uno de sus ángulos interiores tiene 90 grados (es un **ángulo recto**).
2. Un triángulo con base  $b$  y altura  $h$  tiene un **área**  $A = \frac{b \times h}{2}$ .
3. Un triángulo tiene un **perímetro**  $p = a + b + c$  y un **semiperímetro**  $s = \frac{p}{2}$ .
4. Un triángulo con 3 lados  $a, b, c$ , y un semiperímetro  $s$ , tiene un área determinada por la **fórmula de Herón**:  $A = \sqrt{s \times (s - a) \times (s - b) \times (s - c)}$ .
5. Un triángulo de área  $A$  y semiperímetro  $s$ , tiene una **circunferencia inscrita** de  $r = \frac{A}{s}$ .

```

1 double rInCircle(double ab, double bc, double ca) {
2 return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }
3
4 double rInCircle(point a, point b, point c) {
5 return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

```

6. El centro de la circunferencia inscrita es el punto de encuentro de las *bisectrices de los ángulos* del triángulo (ver la parte izquierda de la figura 7.6). Podemos obtener el centro si tenemos dos bisectrices y encontramos su punto de intersección. La implementación:

```

1 // asumimos que se han escrito las funciones de puntos/líneas
2 // devuelve verdadero si hay un centro inCircle, si no, falso
3 // si devuelve verdadero, ctr será el centro de la inscrita
4 // y r será igual a rInCircle
5 bool inCircle(point p1, point p2, point p3, point &ctr, double &r) {
6 r = rInCircle(p1, p2, p3);
7 if (fabs(r) < EPS) return false; // no hay centro de la inscrita
8
9 line l1, l2; // calcular las dos bisectrices
10 double ratio = dist(p1, p2) / dist(p1, p3);
11 point p = translate(p2, scale(toVec(p2, p3), ratio / (1+ratio)));
12 pointsToLine(p1, p, l1);
13
14 ratio = dist(p2, p1) / dist(p2, p3);
15 p = translate(p1, scale(toVec(p1, p3), ratio / (1+ratio)));
16 pointsToLine(p2, p, l2);
17
18 areIntersect(l1, l2, ctr); // obtener su punto de intersección
19 return true; }

```

7. Un triángulo de área  $A$ , tiene una **circunferencia exinscrita** de radio  $R = \frac{a \times b \times c}{4 \times A}$ .

```

1 double rCircumCircle(double ab, double bc, double ca) {
2 return ab * bc * ca / (4.0 * area(ab, bc, ca)); }
3
4 double rCircumCircle(point a, point b, point c) {
5 return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

```

8. El centro de la circunferencia exinscrita, es el punto de encuentro de las *mediatrices* del triángulo (ver la parte derecha de la figura 7.6).
9. Para comprobar si tres segmentos, de longitudes  $a$ ,  $b$  y  $c$ , pueden formar un triángulo, basta con comprobar estas *desigualdades triangulares*:  $(a + b > c) \&& (a + c > b) \&& (b + c > a)$ . Si el resultado es falso, entonces los tres segmentos no pueden formar un triángulo. Si las tres longitudes están ordenadas, siendo  $a$  la menor y  $c$  la mayor, podemos simplificar la comprobación a  $(a + b > c)$ .

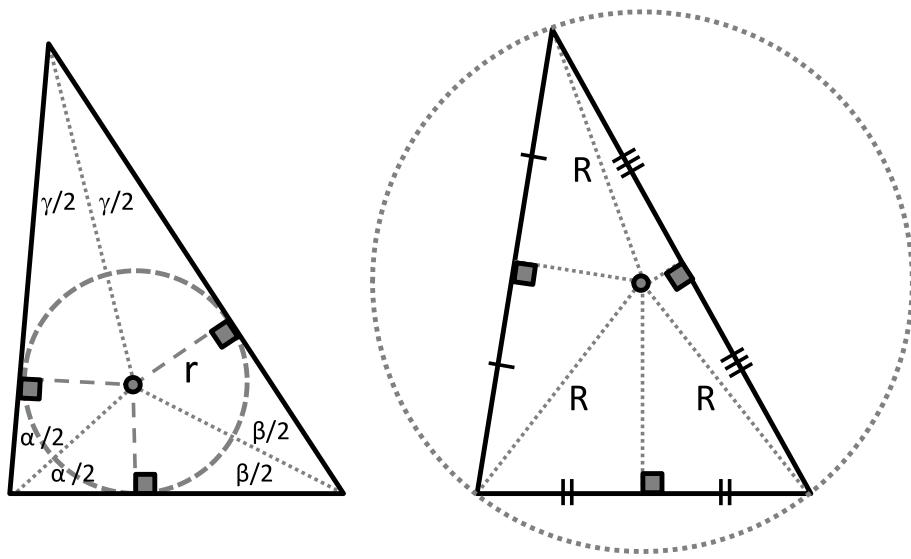


Figura 7.6: Circunferencias inscrita y exinscrita de un triángulo

10. Cuando estudiamos el triángulo, no podemos olvidarnos de la **trigonometría**, el estudio de las relaciones entre los lados y los ángulos de un triángulo. En la trigonometría, el **teorema de cosenos** (llamado también **ley de cosenos**), es una afirmación sobre un triángulo generalista, que relaciona las longitudes de sus lados con el coseno de uno de sus ángulos. Veamos el triángulo escaleno (central) de la figura 7.5. Con las notaciones descritas ahí, tenemos que  $c^2 = a^2 + b^2 - 2 \times a \times b \times \cos(\gamma)$  o  $\gamma = \arccos\left(\frac{a^2+b^2-c^2}{2 \times a \times b}\right)$ . La fórmula para los otros dos ángulos,  $\alpha$  y  $\beta$ , se define de forma similar.
11. En trigonometría, el **teorema de los senos** (también conocido como **ley de los senos**), es una ecuación que relaciona la longitud de los lados de un triángulo arbitrario con los senos de sus ángulos. Veamos el triángulo escaleno (central) de la figura 7.5. Con las notaciones descritas ahí y siendo  $R$  el radio de su circunferencia exinscrita, tenemos que  $\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)} = 2R$ .
12. El **teorema de Pitágoras** especializa el teorema de cosenos. Solo se aplica a los triángulos rectángulos. Si el ángulo  $\gamma$  es recto (de  $90^\circ$  o  $\frac{\pi}{2}$  radianes), entonces  $\cos(\gamma) = 0$ , por lo que el teorema de cosenos se reduce a  $c^2 = a^2 + b^2$ . El teorema de Pitágoras se utiliza para hallar la distancia euclídea entre dos puntos, como hemos visto antes.
13. La **terna pitagórica** se compone de tres enteros positivos  $a$ ,  $b$  y  $c$  (expresada como  $(a, b, c)$ ) tal que  $a^2 + b^2 = c^2$ . Un ejemplo famoso es  $(3, 4, 5)$ . Si  $(a, b, c)$  es una terna pitagórica, entonces también lo será  $(ka, kb, kc)$ , para cualquier entero positivo  $k$ . Una terna pitagórica describe las longitudes enteras de los tres lados de un triángulo rectángulo.



ch7\_03\_triangles.cpp



ch7\_03\_triangles.java

### Ejercicio 7.2.4.1

Digamos que los  $a$ ,  $b$  y  $c$  de un triángulo son  $2^{18}$ ,  $2^{18}$  y  $2^{18}$ . ¿Es posible calcular el área de este triángulo, utilizando la fórmula de Herón que hemos visto en el punto 4, sin provocar un desbordamiento (damos por hecho que utilizamos enteros de 64 bits)? ¿Qué deberíamos hacer para evitar este problema?

### Ejercicio 7.2.4.2\*

Implementa el código para hallar el centro de la circunferencia exinscrita de tres puntos  $a$ ,  $b$  y  $c$ . La estructura de la función es similar a la de `inCircle`, vista en esta sección.

### Ejercicio 7.2.4.3\*

Implementa otro código, que compruebe si un punto  $d$  se encuentra dentro de la circunferencia exinscrita de tres puntos  $a$ ,  $b$  y  $c$ .

## 7.2.5 Objetos bidimensionales: cuadriláteros

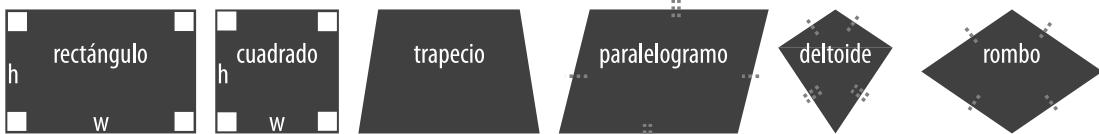


Figura 7.7: Cuadriláteros

1. Un **cuadrilátero** o **cuadrángulo**, es un polígono con cuatro aristas (y cuatro vértices). El término ‘polígono’ está descrito con más detalle en la sección 7.3. La figura 7.7 muestra algunos ejemplos de cuadriláteros.
2. Un **rectángulo**, es un polígono con cuatro aristas, cuatro vértices y cuatro ángulos rectos.
3. Un rectángulo, con anchura  $w$  y altura  $h$ , tiene un **área**  $A = w \times h$ , y un **perímetro**  $p = 2 \times (w + h)$ .
4. El **cuadrado** es un caso especial de rectángulo, en el que  $w = h$ .
5. El **trapezio** es un polígono con cuatro vértices y cuatro aristas, dos de ellas paralelas. Si los dos lados no paralelos tienen la misma longitud, hablamos de un **trapezio isósceles**.
6. Un trapezio con un par de aristas paralelas de longitudes  $w_1$  y  $w_2$ , y con una altura  $h$  entre ambas aristas paralelas, tiene un área  $A = \frac{(w_1+w_2) \times h}{2}$ .

- El **paralelogramo** es un polígono con cuatro aristas y cuatro vértices. Además, los lados opuestos deben ser paralelos.
- El **deltoides** es un cuadrilátero, que tiene dos parejas de lados de la misma longitud, que son adyacentes entre sí. El área de un deltoide es  $\frac{diagonal_1 \times diagonal_2}{2}$ .
- El **rombo** es un paralelogramo especial, en el que todos los lados tienen la misma longitud. También se puede definir como un caso específico de deltoide.

## Comentarios sobre objetos tridimensionales

Los problemas de concursos de programación que implican objetos tridimensionales son poco habituales. Pero cuando aparece uno de ellos en un conjunto de problemas, puede ser de los más difíciles. En la lista de ejercicios de programación que presentamos a continuación, hemos incluido una serie elemental de problemas relacionados con objetos tridimensionales.

## Ejercicios de programación

### Ejercicios de programación relativos a la geometría básica:

#### Puntos y líneas

- |                                            |                                                             |
|--------------------------------------------|-------------------------------------------------------------|
| 1. UVa 00152 - Tree's a Crowd              | (ordenar primero los puntos tridimensionales)               |
| 2. UVa 00191 - Intersection                | (intersección de segmentos)                                 |
| 3. UVa 00378 - Intersecting Lines          | (rutinas de biblioteca: areParallel, areSame, areIntersect) |
| 4. UVa 00587 - There's treasure everywhere | (dist euclídea)                                             |
| 5. UVa 00833 - Water Falls                 | (comprobación recursiva; usar las pruebas ccw)              |
| 6. UVa 00837 - Light and Transparencies    | (segmentos; ordenar primero coordenadas <i>x</i> )          |
| 7. <b>UVa 00920 - Sunny Mountains *</b>    | (dist euclídea)                                             |
| 8. UVa 01249 - Euclid                      | (LA 4601 - SoutheastUSA09; vector)                          |
| 9. UVa 10242 - Fourth Point                | (toVector; translate puntos en relación al vector)          |
| 10. UVa 10250 - The Other Two Trees        | (vector; rotación)                                          |
| 11. <b>UVa 10263 - Railway *</b>           | (usar distToLineSegment)                                    |
| 12. UVa 10357 - Playball                   | (dist euclídea; simulación sencilla de física)              |
| 13. UVa 10466 - How Far?                   | (dist euclídea)                                             |
| 14. UVa 10585 - Center of symmetry         | (ordenar los puntos)                                        |
| 15. UVa 10832 - Yoyodyne Propulsion ...    | (dist euclídea tridimensional; simulación)                  |
| 16. UVa 10865 - Brownie Points             | (puntos y cuadrantes; sencillo)                             |
| 17. UVa 10902 - Pick-up sticks             | (intersección de segmentos)                                 |
| 18. <b>UVa 10927 - Bright Lights *</b>     | (ordenar puntos por gradiente; dist euclídea)               |
| 19. UVa 11068 - An Easy Task               | (2 ecuaciones lineales sencillas con 2 incógnitas)          |
| 20. UVa 11343 - Isolated Segments          | (intersección de segmentos)                                 |
| 21. UVa 11505 - Logo                       | (dist euclídea)                                             |
| 22. UVa 11519 - Logo 2                     | (vector y ángulos)                                          |
| 23. UVa 11894 - Genius MJ                  | (rotación y traslación de puntos)                           |

## Círculos (solo)

1. UVa 01388 - *Graveyard*
  2. **UVa 10005 - Packing polygons \***
  3. UVa 10136 - Chocolate Chip Cookies
  4. UVa 10180 - Rope Crisis in Ropeland
  5. UVa 10209 - Is This Integration?
  6. UVa 10221 - Satellites
  7. UVa 10283 - *The Kissing Circles*
  8. UVa 10432 - Polygon Inside A Circle
  9. UVa 10451 - Ancient ...
  10. UVa 10573 - Geometry Paradox
  11. **UVa 10589 - Area \***
  12. **UVa 10678 - The Grazing Cows \***
  13. UVa 12578 - 10:6:2
- (LA 3708 - NortheasternEurope06; dividir primero el círculo en  $n$  sectores y después en  $(n + m)$ )  
(búsqueda completa; usar `circle2PtsRad`)  
(búsqueda completa; usar `circle2PtsRad`; similar a UVa 10005)  
(punto más cercano de  $AB$  al origen; arco)  
(cuadrado; arcos; similar a UVa 10589)  
(encontrar arco y longitud de la cuerda de un círculo)  
(deducir la fórmula)  
(sencillo: área de un polígono regular de  $n$  lados en un círculo)  
(círculo inscrito/exinscrito de un polígono regular de  $n$  lados)  
(no hay caso ‘imposible’)  
(comprobar si un punto está dentro de la intersección de 4 círculos)  
(área de una *ellipse*; generalización de la fórmula del área de un círculo)  
(área de rectángulo y círculo)

## Triángulos (y círculos)

1. UVa 00121 - Pipe Fitters
  2. UVa 00143 - Orchard Trees ...
  3. UVa 00190 - Circle Through Three ...
  4. UVa 00375 - Inscribed Circles and ...
  5. UVa 00438 - The Circumference of ...
  6. UVa 10195 - The Knights Of The ...
  7. UVa 10210 - Romeo & Juliet
  8. UVa 10286 - The Trouble with a ...
  9. UVa 10347 - Medians
  10. UVa 10387 - Billiard
  11. UVa 10522 - Height to Area
  12. **UVa 10577 - Bounding box \***
  13. UVa 10792 - *The Laurel-Hardy Story*
  14. UVa 10991 - Region
  15. **UVa 11152 - Colourful ... \***
  16. UVa 11164 - *Kingdom Division*
  17. UVa 11281 - *Triangular Pegs in ...*
  18. UVa 11326 - *Laser Pointer*
  19. UVa 11437 - *Triangle Fun*
  20. UVa 11479 - Is this the easiest problem?
  21. UVa 11579 - Triangle Trouble
  22. UVa 11854 - Egypt
  23. **UVa 11909 - Soya Milk \***
  24. UVa 11936 - *The Lazy Lumberjacks*
- (usar el teorema de Pitágoras; rejilla)  
(contar los puntos enteros de un triángulo; cuidado con la precisión)  
(circunferencia de un triángulo)  
(círculo inscrito de un triángulo)  
(circunferencia de un triángulo)  
(círculo inscrito de un triángulo; fórmula de Herón)  
(trigonometría básica)  
(ley de los senos)  
(dadas las 3 medianas de un triángulo, hallar su área)  
(superficie expansiva; trigonometría)  
(deducir la fórmula; fórmula de Herón)  
(obtener centro y radio de un círculo exinscrito a partir de 3 puntos; obtener todos los vértices; obtener los  $x$  e  $y$  mínimos y máximos del polígono)  
(deducir las fórmulas trigonométricas)  
(fórmula de Herón; ley de cosenos; área del sector)  
(círculo inscrito y circunferencia de un triángulo; fórmula de Herón)  
(usar las propiedades del triángulo)  
(el círculo mínimo de un triángulo no obtuso es su circunferencia; si el triángulo es obtuso, el radio del círculo mínimo es su lado más largo)  
(trigonometría; tangentes; truco de reflexión)  
(pista:  $\frac{1}{7}$ )  
(comprobación de propiedades)  
(ordenar; comprobar vorazmente si tres lados sucesivos satisfacen la desigualdad del triángulo y si es el triángulo más grande encontrado)  
(teorema/terna de Pitágoras)  
(ley de los senos (o tangente); dos casos posibles)  
(ver si 3 lados forman un triángulo válido)

## Cuadriláteros

1. UVa 00155 - All Squares  
(conteo recursivo)
2. UVa 00460 - Overlapping Rectangles  
(intersección rectángulo-rectángulo)
3. Entry Level: UVa 00476 - Points in Figures: ...  
(problema básico; ver los problemas relacionados UVa 477 y 478)
4. UVa 00477 - Points in Figures: ...  
(similar a UVa 476 y 478)
5. UVa 11207 - The Easiest Way  
(cortar rectángulo en 4 cuadrados iguales)
6. UVa 11345 - Rectangles  
(intersección rectángulo-rectángulo)
7. UVa 11455 - Behold My Quadrangle  
(comprobación de propiedades)
8. UVa 11639 - Guard the Land  
(intersección rectángulo-rectángulo; usar *array* de etiquetas)
9. **UVa 11800 - Determine the Shape \***  
(usar *next\_permutation* ayuda a probar las  $4! = 24$  permutaciones posibles de 4 puntos; comprobar si se puede satisfacer el cuadrado, rectángulo, rombo, paralelogramo y trapecio, en ese orden)  
(empaquetar dos círculos en un rectángulo)
10. UVa 11834 - Elevator  
(LA 5001 - KualaLumpur 10; comenzar con tres lados de 1, 1, 1; después, el cuarto y siguientes deben ser la suma de los tres anteriores para formar una línea; repetir hasta llegar al lado  $n$ -ésimo)
11. **UVa 12256 - Making Quadrilaterals \***

## Objetos tridimensionales

1. **UVa 00737 - Gleaming the Cubes \***  
(cubo e intersección de cubos)
2. **UVa 00815 - Flooded \***  
(volumen; voraz; ordenar por altura; simulación)
3. **UVa 10297 - Beavergnaw \***  
(conos; cilindros; volúmenes)

## Perfiles de los inventores de algoritmos

**Pitágoras de Samos** ( $\approx$  500 a.C.) fue un matemático y filósofo griego, nacido en la isla de Samos. Es conocido principalmente por el teorema de Pitágoras para triángulos rectángulos.

**Euclides de Alejandría** ( $\approx$  300 a.C.) fue un matemático griego, el ‘padre de la geometría’. Provenía de la ciudad de Alejandría. Su trabajo más influyente en el campo de las matemáticas (especialmente la geometría) es los ‘Elementos’. En la obra, Euclides dedujo los principios de lo que conocemos como geometría euclídea, a partir de un pequeño conjunto de axiomas.

**Herón de Alejandría** ( $\approx$  10-70 d.C.) fue un matemático de la antigua Grecia, de la ciudad de Alejandría, en el Egipto romano (la misma ciudad que Euclides). Su nombre está muy ligado a su fórmula para el cálculo del área de un triángulo, a partir de la longitud de sus lados.

**Ronald Lewis Graham** (nacido en 1935) es un matemático estadounidense. En 1972, inventó el método de Graham para encontrar la envolvente convexa de un conjunto finito de puntos en el plano. En la actualidad, hay muchas otras variantes y mejoras del algoritmo para encontrar la envolvente convexa.

## 7.3 Algoritmos en polígonos con bibliotecas

Un **polígono** es una figura plana, que está limitada por un camino cerrado (que comienza y termina en el mismo vértice), compuesta de una secuencia finita de segmentos rectos. Estos segmentos reciben el nombre de aristas o lados. El punto en el que se encuentran dos segmentos es un vértice o esquina del polígono. El polígono es el origen de muchos problemas de geometría (computacional), ya que permite al autor del problema presentar objetos más realistas que los tratados en la sección 7.2.

### 7.3.1 Representación de polígonos

La forma normalizada de representar un polígono, consiste simplemente en enumerar los vértices del mismo, en orden, a favor o en contra de las agujas del reloj, siendo el primer vértice igual al último (algunas de las funciones que mencionaremos más adelante en esta sección, requieren esta disposición para simplificar la implementación, ver el [ejercicio 7.3.4.2\\*](#)). En este libro, utilizaremos la ordenación de vértices en contra de las agujas del reloj. En la parte derecha de la figura 7.8, se muestra el polígono resultante de ejecutar el siguiente código.

```
1 // 6 puntos, en orden contrario a las agujas del reloj, índice desde 0
2 vector<point> P;
3 P.push_back(point(1, 1)); // P0
4 P.push_back(point(3, 3)); // P1
5 P.push_back(point(9, 1)); // P2
6 P.push_back(point(12, 4)); // P3
7 P.push_back(point(9, 7)); // P4
8 P.push_back(point(1, 7)); // P5
9 P.push_back(P[0]); // importante: cerrar el polígono
```

### 7.3.2 Perímetro de un polígono

El perímetro de un polígono (sea cóncavo o convexo) con  $n$  vértices dados, en alguna forma ordenada (a favor o en contra de las agujas del reloj), se puede calcular con la sencilla función que reproducimos a continuación.

```
1 // devuelve el perímetro, que es la suma de las distancias euclídeas,
2 // de los segmentos consecutivos (aristas de un polígono)
3 double perimeter(const vector<point> &P) {
4 double result = 0.0;
5 for (int i = 0; i < (int)P.size()-1; i++) // recuerda que P[n-1] = P[0]
6 result += dist(P[i], P[i+1]); // porque duplicamos P[0]
7 return result; }
```

### 7.3.3 Área de un polígono

El área, con signo  $A$ , de un polígono (cónvexo o convexo) con  $n$  vértices dados, en alguna forma ordenada (a favor o en contra de las agujas del reloj), se puede hallar calculando el determinante de la matriz, como se muestra a continuación. Esta fórmula debería estar entre las incluidas en el código de biblioteca.

$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{bmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \dots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \dots - x_0 \times y_{n-1})$$

```
1 // devuelve el área, que es la mitad del determinante
2 double area(const vector<point> &P) {
3 double result = 0.0, x1, y1, x2, y2;
4 for (int i = 0; i < (int)P.size()-1; i++) {
5 x1 = P[i].x; x2 = P[i+1].x;
6 y1 = P[i].y; y2 = P[i+1].y;
7 result += (x1*y2 - x2*y1);
8 }
9 return fabs(result)/2.0; }
```

### 7.3.4 Comprobación de si un polígono es convexo

Se dice que un polígono es **convexo** si cualquier segmento trazado dentro del mismo no produce una intersección con alguna de sus aristas. En caso contrario, el polígono será **cónvexo**.

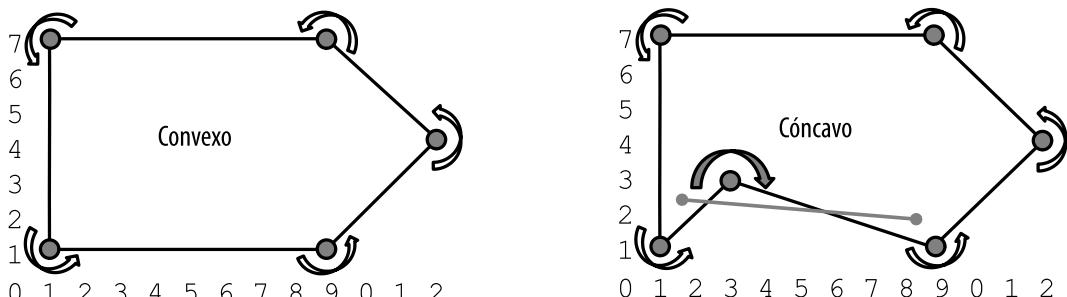


Figura 7.8: Izquierda: polígono convexo, derecha: polígono cónvexo

Sin embargo, para comprobar si un polígono es convexo, existe una técnica computacional más sencilla que “intentar comprobar todos los segmentos que se pueden dibujar dentro del polígono”. Basta con que comprobemos si todos grupos de tres vértices consecutivos del polígono giran hacia el mismo lado (a la izquierda, o en contra de las agujas del reloj, si así es como están ordenados, como en todos los ejemplos de este libro, o a la derecha en otro caso). Si podemos encontrar, al menos, una 3-tupla donde esto sea falso, entonces es cónvexo (figura 7.8).

```

1 bool isConvex(const vector<point> &P) { // devuelve verdadero si los tres
2 int sz = (int)P.size(); // vértices consecutivos de P giran igual
3 if (sz <= 3) return false; //un punto/sz=2 o una línea/sz=3 no es convexa
4 bool isLeft = ccw(P[0], P[1], P[2]); // recordar un resultado
5 for (int i = 1; i < sz-1; i++) // y compararlo con los otros
6 if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
7 return false; // signo diferente -> el polígono es cóncavo
8 return true; } // el polígono es convexo

```

### Ejercicio 7.3.4.1\*

¿Qué parte del código anterior deberías modificar para aceptar puntos colineales? Por ejemplo, el polígono  $\{(0,0), (2,0), (4,0), (2,2), (0,0)\}$  debería ser tratado como convexo.

### Ejercicio 7.3.4.2\*

Si el primer vértice no se repite como último vértice, ¿trabajaránd correctamente las funciones `perimeter`, `area` e `isConvex` vistas antes?

## 7.3.5 Comprobación de si un punto está dentro de un polígono

Otra comprobación muy común sobre un polígono  $P$ , es verificar si un punto  $pt$  está dentro o fuera del mismo. La siguiente función, que implementa el ‘algoritmo del número de cuerda’, permite realizar esa comprobación tanto para polígonos cóncavos como convexos. Funciona calculando la suma de los ángulos entre tres puntos  $\{P[i], pt, P[i + 1]\}$ , donde  $(P[i]-P[i + 1])$  son lados consecutivos del polígono  $P$ , teniendo cuidado en los giros a la izquierda (se suma el ángulo) y a la derecha (se resta el ángulo). Si la suma final es  $2\pi$  (360 grados), entonces  $pt$  está dentro del polígono  $P$  (ver la figura 7.9).

```

1 // verdadero si el punto p está dentro del polígono P cóncavo/convexo
2 bool inPolygon(point pt, const vector<point> &P) {
3 if ((int)P.size() == 0) return false;
4 double sum = 0; // asumir que el primer vértice es igual al último
5 for (int i = 0; i < (int)P.size()-1; i++) {
6 if (ccw(pt, P[i], P[i+1]))
7 sum += angle(P[i], pt, P[i+1]); // izquierda/contra agujas reloj
8 else sum -= angle(P[i], pt, P[i+1]); } // derecha/a favor agujas reloj
9 return fabs(fabs(sum) - 2*PI) < EPS; }

```

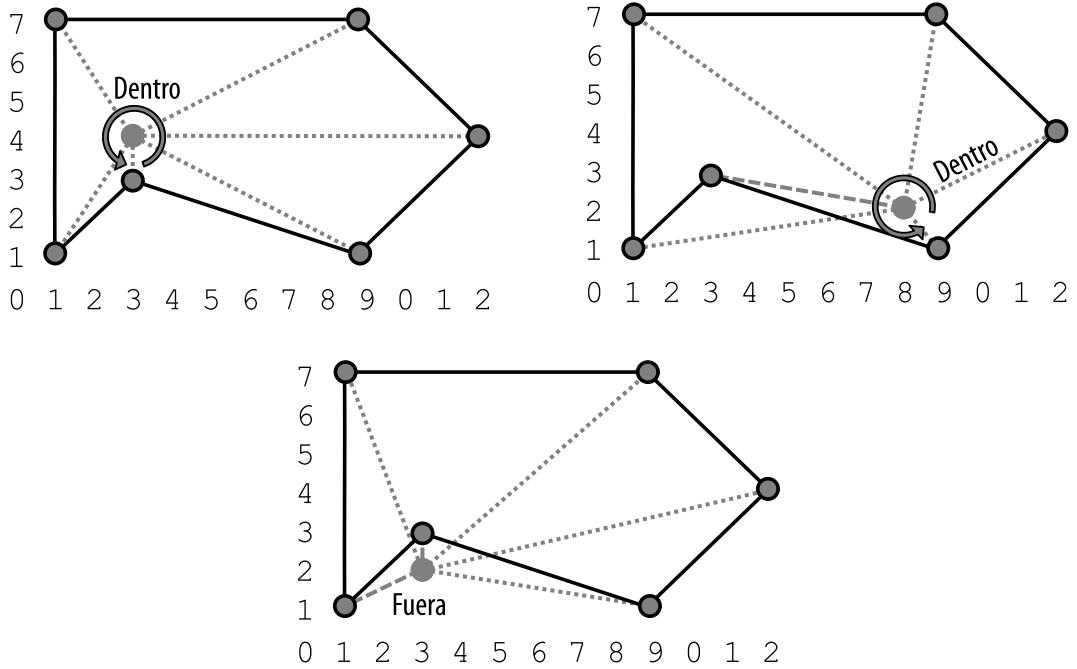


Figura 7.9: Arriba-izquierda: dentro, arriba-derecha: también dentro, abajo: fuera

### Ejercicio 7.3.5.1\*

¿Qué le ocurre a la rutina `inPolygon` si el punto `pt` se encuentra en una de las aristas del polígono `P`, por ejemplo, `pt = P[0]` o `pt` es el punto medio entre `P[0]` y `P[1]`, etc.? ¿Qué podemos hacer para solventar esa situación?

### Ejercicio 7.3.5.2\*

Comenta los pros y los contras de los siguientes métodos alternativos de comprobación de si un punto está dentro de un polígono:

1. Dividir un polígono convexo en triángulos y comprobar si la suma de las áreas de los mismos es igual al área del polígono convexo. ¿Se puede utilizar este método con un polígono cóncavo?
2. Algoritmo de *ray casting*: trazamos un rayo desde el punto a cualquier dirección fija, de forma que se produzca una intersección con alguna arista del polígono. Si hay un número impar/par de intersecciones, el punto está dentro/fuera, respectivamente.

### 7.3.6 Corte de un polígono con una línea recta

Otra cosa interesante que podemos hacer con un polígono *convexo* (ver el **ejercicio 7.3.6.2\*** para un polígono cóncavo), es dividirlo en dos subpolígonos convexos, mediante una línea recta, definida por dos puntos  $a$  y  $b$ . Más adelante, enumeraremos algunos ejercicios de programación que utilizan esta función.

La idea básica de la siguiente rutina `cutPolygon`, es iterar por los vértices del polígono original  $Q$ , de uno en uno. Si la línea  $ab$  y el vértice  $v$  forman un giro a la izquierda (lo que implica que  $v$  está en el lado izquierdo de la línea  $ab$ ), colocamos  $v$  dentro del nuevo polígono  $P$ . Una vez que encontramos una arista del polígono, que forme una intersección con la línea  $ab$ , utilizamos ese punto como parte del nuevo polígono  $p$  (ver la parte izquierda de la figura 7.10, punto 'C'). Obviamos los siguientes vértices de  $Q$ , que quedan a la derecha de la línea  $ab$ . Antes o después, volveremos a encontrar una arista que forme otra intersección con la línea  $ab$  (ver la parte izquierda de la figura 7.10, punto 'D', que resulta ser uno de los vértices originales del polígono  $Q$ ). Continuamos añadiendo vértices de  $Q$  a  $P$ , porque volvemos a estar en el lado izquierdo de la línea  $ab$ . Nos detenemos cuando lleguemos al vértice inicial y devolvemos el polígono resultante  $P$  (ver la parte derecha de la figura 7.10).

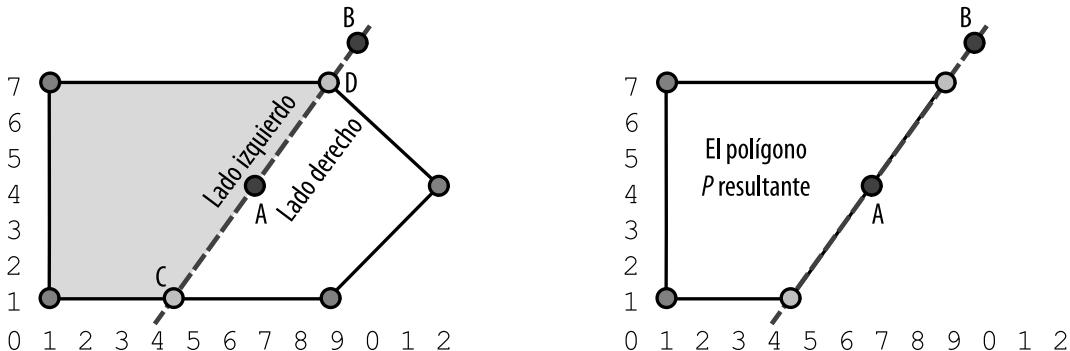


Figura 7.10: Izquierda: antes del corte, derecha: después del corte

```

1 // segmento p-1 intersecciona con línea A-B
2 point lineIntersectSeg(point p, point q, point A, point B) {
3 double a = B.y-A.y;
4 double b = A.x-B.x;
5 double c = B.x*A.y - A.x*B.y;
6 double u = fabs(a*p.x + b*p.y + c);
7 double v = fabs(a*q.x + b*q.y + c);
8 return point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v)); }
9
10 // corta polígono Q por la línea formada por puntos a -> b
11 // (nota: el último punto debe ser el mismo que el primero)
12 vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
13 vector<point> P;
14 for (int i = 0; i < (int)Q.size(); i++) {
15 double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
```

```

16 if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
17 if (left1 > -EPS) P.push_back(Q[i]); // Q[i] está a la izquierda de ab
18 if (left1 * left2 < -EPS) // la arista (Q[i], Q[i+1]) cruza la línea ab
19 P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
20 }
21 if (!P.empty() && !(P.back() == P.front()))
22 P.push_back(P[0]); // hacer que primer punto de P = último punto
23 return P;

```

Para ayudar al lector a comprender estos algoritmos sobre polígonos, hemos construido una herramienta de visualización que aporte más información al contenido del libro. Es posible dibujar nuestro propio polígono y pedirle a la herramienta una explicación visual del algoritmo tratado en esta sección.



[visualgo.net/geometry](http://visualgo.net/geometry)

### Ejercicio 7.3.6.1

Esta función `cutPolygon` solo devuelve el lado izquierdo del polígono  $Q$  después del corte con la línea  $ab$ . ¿Qué deberíamos hacer si, en su lugar, queremos el lado derecho?

### Ejercicio 7.3.6.2\*

¿Qué ocurre si ejecutamos la función `cutPolygon` sobre un polígono *cónvexo*?

## 7.3.7 Búsqueda de la envolvente convexa de un conjunto de puntos

La **envolvente convexa** de un conjunto de puntos  $P$ , es el polígono convexo  $CH(P)$  más pequeño, en el que cada punto de  $P$  está, o en el límite, o dentro de  $CH(P)$ . Imagina que los puntos son clavos en una superficie bidimensional plana y que tenemos una goma elástica lo suficientemente grande como para rodearlos a todos. Si liberamos esa goma elástica, se ajustará al área más pequeña posible. Ese es el área de la envolvente convexa de ese conjunto de puntos/clavos (ver la figura 7.11). Encontrar la envolvente convexa de un conjunto de puntos tiene aplicaciones prácticas en problemas de *empaquetado*.

Como cada vértice de  $CH(P)$  es un vértice del conjunto de puntos  $P$ , el algoritmo de búsqueda de la envolvente convexa es, esencialmente, un algoritmo que decide qué puntos de  $P$  forman parte de ella. Existen varios algoritmos de búsqueda de la envolvente convexa. En esta sección hemos elegido el *método de Ronald Graham*, con una complejidad de tiempo de  $O(n \log n)$ .

El método de Graham comienza ordenando los  $n$  puntos de  $P$ , donde el primero no necesita ser replicado también como el último (ver la figura 7.12.A), en base a sus ángulos en relación a

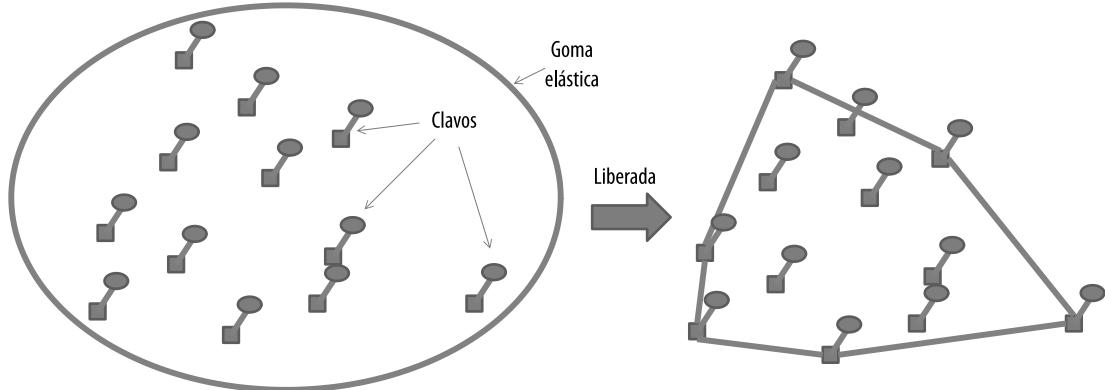


Figura 7.11: Analogía de la goma elástica para encontrar la envolvente convexa

un punto llamado pivote. En nuestro ejemplo, hemos elegido como pivote el punto más abajo y a la derecha de  $P$ . Después de la ordenación en base a los ángulos con relación a este pivote, podemos ver que las aristas 0-1, 0-2, 0-3, ..., 0-10 y 0-11 están ordenadas en sentido contrario a las agujas del reloj (ver los puntos 1 a 11 en relación al punto 0, en la figura 7.12.B).

```

1 point pivot(0, 0);
2 bool angleCmp(point a, point b) { // función de ordenación de ángulos
3 if (collinear(pivot, a, b)) // caso especial
4 return dist(pivot, a) < dist(pivot, b); // comprobar el más cercano
5 double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
6 double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
7 return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; } // comparar dos ángulos
8 vector<point> CH(vector<point> P) { // se puede reordenar el contenido de P
9 int i, j, n = (int)P.size();
10 if (n <= 3) {
11 if (!(P[0] == P[n-1])) P.push_back(P[0]); // protección para caso límite
12 return P; } // caso especial, la CH es el propio P
13
14 // encontrar P_0 = punto con Y menor y, con empate, X más a la derecha
15 int P0 = 0;
16 for (i = 1; i < n; i++) // $O(n)$
17 if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
18 P0 = i;
19
20 point temp = P[0]; P[0] = P[P0]; P[P0] = temp; // cambiar $P[P_0]$ con $P[0]$
21
22 // segundo, ordenar puntos por ángulo en relación al pivote P_0
23 pivot = P[0]; // usar esta variable global como referencia
24 sort(++P.begin(), P.end(), angleCmp); // no ordenamos $P[0]$, $O(n \log n)$
25 // continuará

```

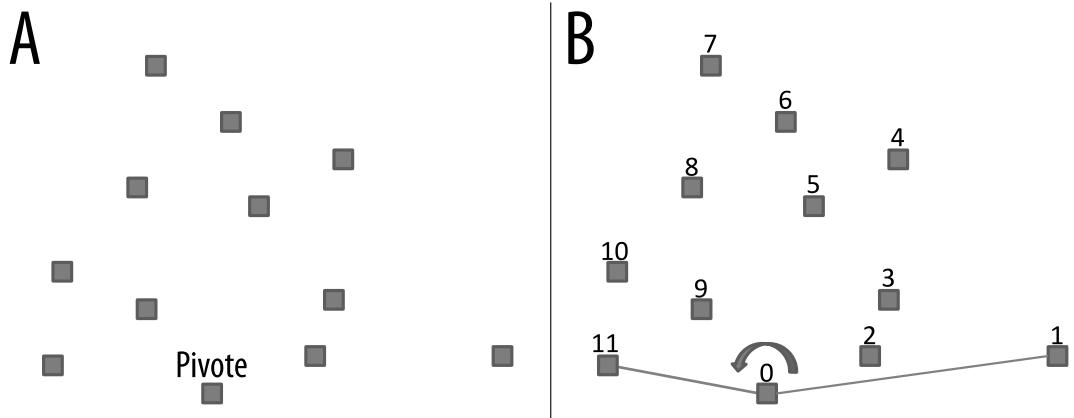


Figura 7.12: Orden de un conjunto de 12 puntos por ángulo en relación al pivote (punto 0)

Después, el algoritmo mantiene una pila  $S$  de puntos candidatos. Cada punto de  $P$  se añade *una vez* a  $S$ , de donde se acabarán eliminando los que no vayan a formar parte de  $CH(P)$ . El método de Graham mantiene este aspecto invariable: los tres elementos superiores de la pila  $S$  siempre deben formar un giro a la izquierda (propiedad básica de un polígono convexo).

Inicialmente, insertamos los tres puntos  $N-1$ ,  $0$  y  $1$ . En nuestro ejemplo, la pila comienza conteniendo (abajo) 11-0-1 (arriba). Esto siempre forma un giro a la izquierda.

Ahora, examinemos la figura 7.13.C. Aquí, intentamos insertar el punto  $2$ , y  $0-1-2$  es un giro a la izquierda, por lo que lo aceptamos. La pila  $S$  será ahora (abajo) 11-0-1-2 (arriba).

Seguimos con la figura 7.13.D. En este caso, intentamos insertar el punto  $2$  y  $1-2-3$  conforma un giro a la *derecha*. Esto significa que, si aceptamos el punto anterior a  $3$ , que es  $2$ , no tendremos un polígono convexo. Así que tenemos que extraer el punto  $2$  de la pila. Ahora  $S$  es, nuevamente, (abajo) 11-0-1 (arriba). Después, volvemos a intentar insertar el punto  $3$ . Como los tres elementos superiores *actuales* de la pila  $S$ ,  $0-1-3$ , forman un giro a la izquierda, podemos aceptar el punto  $3$ . La pila  $S$  pasa a ser (abajo) 11-0-1-3 (arriba).

Repetimos este proceso hasta terminar con todos los vértices (figura 7.13.E-F-G-H). Al finalizar la ejecución del método de Graham, lo que quede en  $S$  es el conjunto de puntos de  $CH(P)$  (ver la figura 7.13.H, la pila contiene (abajo) 11-0-1-4-7-10-11 (arriba)). El método de Graham elimina todos los giros a la derecha. Como tres vértices consecutivos cualquiera de  $S$  siempre implican un giro a la izquierda, tenemos un polígono convexo.

A continuación, incluimos la implementación del método de Graham. Usamos `vector<point>S`, que se comporta como una pila, en lugar de `stack<point>S`. La primera parte del método de Graham (la búsqueda del pivote) se ejecuta en solo  $O(n)$ . La tercera parte (las comprobaciones de giros), también lo hace en  $O(n)$ . Podemos determinar esto a partir del hecho de que cada uno de los  $n$  vértices solo puede entrar y salir de la pila una vez. La segunda parte (ordenación de los puntos por su ángulo, en relación al pivote  $P[0]$ ) es el *grueso* del algoritmo, y requiere  $O(n \log n)$ . El tiempo total de ejecución del método de Graham es de  $O(n \log n)$ .

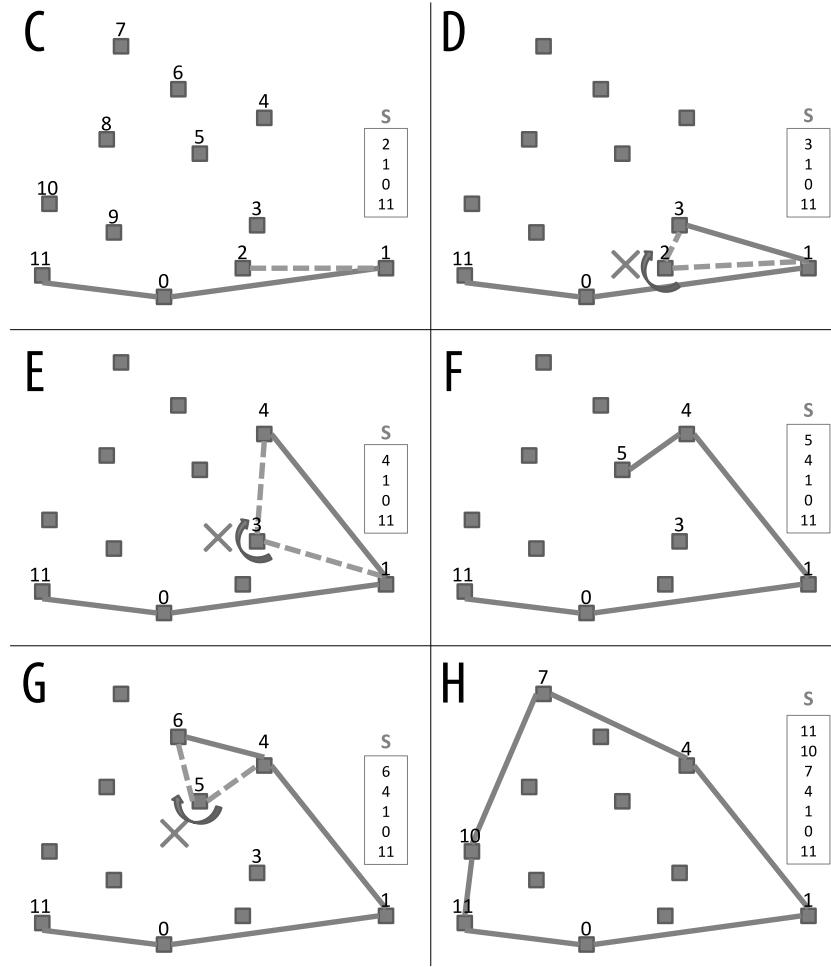


Figura 7.13: La parte principal del método de Graham

```

1 // continuación del código anterior
2 // tercero, las comprobaciones ccw
3 vector<point> S;
4 S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]); // S inicial
5 i = 2; // después, comprobamos el resto
6 while (i < n) { // n debe ser >= 3 para que este método funcione, O(n)
7 j = (int)S.size()-1;
8 if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]); //izquierda, aceptado
9 else S.pop_back(); } // o eliminar de S hasta que haya giro izquierda
10 return S; } // devolver el resultado

```

Finalizamos esta sección, y este capítulo, recomendando al lector la herramienta de visualización que hemos construido para mejorar este libro. En ella, se pueden encontrar varios algoritmos de envolvente convexa, incluyendo el método de Graham, el algoritmo de la cadena monótona de

Andrew (ver el [ejercicio 7.3.7.4\\*](#)), y el algoritmo de la marcha de Jarvis. También animamos al lector a explorar nuestro código, fuente para resolver los ejercicios de programación que incluimos en esta sección.



[visualgo.net/geometry](http://visualgo.net/geometry)



ch7\_04\_polygon.cpp



ch7\_04\_polygon.java

### Ejercicio 7.3.7.1

Supongamos que tenemos 5 puntos,  $P = \{(0, 0), (1, 0), (2, 0), (2, 2), (0, 2)\}$ . La envolvente convexa de los 5 puntos está formada, de hecho, por ellos mismos (más la vuelta al vértice  $(0, 0)$ ). Sin embargo, nuestra implementación del método de Graham elimina el punto  $(1, 0)$ , pues  $(0, 0)-(1, 0)-(2, 0)$  son colineales. ¿Qué parte de la implementación del algoritmo de Graham debemos modificar para aceptar puntos colineales?

### Ejercicio 7.3.7.2

Dentro de la función `angleCmp`, hay una llamada a otra, `atan2`. Esta se utiliza para comparar los dos ángulos, pero, ¿qué devuelve exactamente `atan2`? Investígalo.

### Ejercicio 7.3.7.3\*

Prueba el código del método de Graham y obtén la  $CH(P)$  de los siguientes casos límite. ¿Cuál es su envolvente convexa?

1. Un solo punto, por ejemplo,  $P_1 = \{(0, 0)\}$ .
2. Dos puntos (una línea), por ejemplo,  $P_2 = \{(0, 0), (1, 0)\}$ .
3. Tres puntos (un triángulo), por ejemplo,  $P_3 = \{(0, 0), (1, 0), (1, 1)\}$ .
4. Tres puntos (colineales), por ejemplo,  $P_4 = \{(0, 0), (1, 0), (2, 0)\}$ .
5. Cuatro puntos (colineales), por ejemplo,  $P_5 = \{(0, 0), (1, 0), (2, 0), (3, 0)\}$ .

### Ejercicio 7.3.7.4\*

La implementación del método de Graham que hemos presentado, puede ser poco eficiente para un  $n$  grande, pues se recalcula  $\text{atan2}$  cada vez que se realiza una comparación de ángulos (y esto puede ser un problema cuando el ángulo sea cercano a los 90 grados). En realidad, la misma idea básica del método de Graham también funciona si la entrada está ordenada en base a la coordenada  $x$  (y, en caso de igualdad, por la coordenada  $y$ ), en vez de por el ángulo. De esa forma, la envolvente se calcula en 2 pasos, generando por separado las partes *superior* e *inferior*. Esta modificación fue desarrollada por A. M. Andrew y es conocida como el algoritmo de la cadena monótona de Andrew. Tiene las mismas propiedades básicas que el método de Graham, pero evita las costosas comparaciones entre ángulos [9]. Investiga este algoritmo e impleméntalo.

A continuación, incluimos una lista de ejercicios de programación relacionados con polígonos. Sin el código de biblioteca escrito previamente, que hemos visto en esta sección, muchos de estos problemas parecen ‘difíciles’. Con el código de biblioteca disponible, se vuelven asumibles, ya que se pueden descomponer en una pocas rutinas. Dedica tiempo a resolverlos, especialmente aquellos señalados como obligatorios \*.

### Ejercicios de programación

#### Ejercicios de programación relacionados con polígonos:

1. UVa 00109 - Scud Busters  
(encontrar CH; comprobar si un punto `inPolygon`; calcular `area` del polígono)
2. UVa 00137 - Polygons  
(intersección de polígonos convexos; intersección de segmentos; `inPolygon`, CH, `area`; principio de inclusión-exclusión)
3. UVa 00218 - Moth Eradication  
(LA 5157 - WorldFinals KansasCity92; encontrar CH; `perimeter` de un polígono)
4. UVa 00361 - Cops and Robbers  
(comprobar si un punto está dentro de la CH del policía/ladrón; si un punto `pt` está en la envolvente convexa, seguro que se formará un triángulo usando tres vértices de la misma que contiene `pt`)  
(`inPolygon`/triángulo/círculo/rectángulo)
5. UVa 00478 - Points in Figures: ...  
(problema básico de CH; pero el formato de la salida es un poco tedioso)
6. UVa 00596 - The Incredible Hull  
(rutina básica `inPolygon`; el polígono de entrada puede ser cóncavo o convexo)
7. UVa 00634 - Polygon  
(CH; con formato en la salida)
8. UVa 00681 - Convex Hull Finding  
(intersección de línea vertical y polígono; ordenar; segmentos alternos)
10. **UVa 01111 - Trash Removal\***  
(LA 5138 - WorldFinals Orlando11; CH; distancia de cada lado de la CH, que es paralela al lado, a cada vértice de la CH)  
(`area` del polígono; área del círculo)
11. UVa 01206 - Boundary Points  
(LA 3169 - Manila06; envolvente convexa CH)
12. UVa 10002 - Center of Mass?  
(centroide; centro de la CH; `area` del polígono)
13. UVa 10060 - A Hole to Catch a Man  
(`area` del polígono; área del círculo)
14. UVa 10065 - Useless Tile Packers  
(encontrar `area` del polígono, la CH, y el `area` de la CH)
15. UVa 10112 - Myacm Triangles  
(comprobar si el punto `inPolygon` en el triángulo; similar a UVa 478)
16. UVa 10406 - Cutting tabletops  
(vector; `rotate`; `translate`; después `cutPolygon`)

|                                        |                                                                           |
|----------------------------------------|---------------------------------------------------------------------------|
| 17. UVa 10652 - Board Wrapping *       | (rotate; translate; CH; area)                                             |
| 18. UVa 11096 - Nails                  | (problema clásico de CH; perimeter de un polígono; empieza con este)      |
| 19. UVa 11265 - The Sultan's Problem * | (parece complicado, pero en realidad es solo cutPolygon; inPolygon; area) |
| 20. UVa 11447 - Reservoir Logs         | (area de un polígono)                                                     |
| 21. UVa 11473 - Campus Roads           | (perimeter modificado de un polígono)                                     |
| 22. UVa 11626 - Convex Hull            | (encontrar CH; cuidado con los puntos colineales)                         |

## 7.4 Soluciones a los ejercicios no resaltados

**Ejercicio 7.2.1.1:** 5,0.

**Ejercicio 7.2.1.2:** (-3,0, 10,0).

**Ejercicio 7.2.1.3:** (-0,674, 10,419).

**Ejercicio 7.2.2.1:** la ecuación de la línea  $y = mx + c$  no puede manejar todos los casos: las líneas verticales tienen una pendiente ‘infinita’ y las ‘casi verticales’ también pueden ser problemáticas. Si utilizamos esta ecuación, tenemos que tratar las líneas verticales por separado en nuestro código, lo que reduce las posibilidades de un veredicto de aceptado. Por suerte, podemos evitar esto utilizando una ecuación mejor para la línea:  $ax + by + c = 0$ .

**Ejercicio 7.2.2.2:**  $-0,5 \times x + 1,0 \times y - 1,0 = 0,0$ . Recuerda el valor fijo de 1,0 para  $b$ .

**Ejercicio 7.2.2.3:**  $1,0 \times x + 0,0 \times y - 2,0 = 0,0$ . Si utilizas la ecuación de la línea  $y = mx + c$ , tendrás  $x = 2,0$ , pero no se puede representar una línea vertical utilizando la forma  $y = ?$ .

**Ejercicio 7.2.2.4:** dados 2 puntos  $(x_1, y_1)$  y  $(x_2, y_2)$ , es posible calcular la pendiente con  $m = \frac{y_2 - y_1}{x_2 - x_1}$ . A partir de ahí, se puede calcular la ‘intersección con  $y$ ’,  $c$ , mediante la sustitución de los valores de un punto (cualquiera de ellos) y la pendiente  $m$ . El código tendrá este aspecto (puedes ver que tenemos que tratar la línea vertical por separado y de forma poco práctica):

```

1 struct line2 { double m, c; }; // otra forma de representar una línea
2
3 int pointsToLine2(point p1, point p2, line2 &l) {
4 if (p1.x == p2.x) { // caso especial: línea vertical
5 l.m = INF; // l contiene m = INF y c = valor_x
6 l.c = p1.x; // para identificar línea vertical x = valor_x
7 return 0; // necesitamos esto para diferenciar el resultado
8 }
9 else {
10 l.m = (double)(p1.y-p2.y) / (p1.x-p2.x);
11 l.c = p1.y - l.m*p1.x;
12 return 1; // l contiene m y c de la ecuación de la línea y = mx + c
13 }
}

```

### Ejercicio 7.2.2.5:

```
1 // convertir punto y gradiente/pendiente a linea
2 void pointSlopeToLine(point p, double m, line &l) {
3 l.a = -m; // siempre -m
4 l.b = 1; // siempre 1
5 l.c = -(l.a * p.x) + (l.b * p.y)); } // calcular esto
```

Ejercicio 7.2.2.6: (5,0,3,0).

Ejercicio 7.2.2.7: (4,0,2,5).

Ejercicio 7.2.2.8: (-3,0,5,0).

Ejercicio 7.2.2.9: (0,0,4,0). El resultado es diferente al del **Ejercicio 7.2.2.8**. ‘Trasladar y rotar’ es distinto a ‘rotar y trasladar’. Cuidado con el orden en el que se realizan estas acciones.

Ejercicio 7.2.2.10: (1,0,2,0). Si no rotamos sobre el origen, trasladamos el punto de entrada (3, 2) por un vector descrito por  $-p$ , (-2, -1), al punto  $c'$  (1, 1). Despu s, realizamos el giro de 90 grados, contra las agujas del reloj, alrededor del punto origen, para obtener  $c''$  (-1, 1). Por  ltimo, trasladamos  $c''$ , por un vector descrito por  $p$ , al punto (1, 2), para obtener la respuesta.

Ejercicio 7.2.2.11: esta es la soluci n:

```
1 void closestPoint(line l, point p, point &ans) {
2 line perpendicular; // perpendicular a l y pasa por p
3 if (fabs(l.b) < EPS) { // caso especial 1: lnea vertical
4 ans.x = -(l.c); ans.y = p.y; return; }
5
6 if (fabs(l.a) < EPS) { // caso especial 2: lnea horizontal
7 ans.x = p.x; ans.y = -(l.c); return; }
8
9 pointSlopeToLine(p, 1 / l.a, perpendicular); // lnea normal
10 // intersecci n de la lnea l con esta lnea perpendicular
11 // el punto de intersecci n es el m s cercano
12 areIntersect(l, perpendicular, ans); }
```

Ejercicio 7.2.2.12: a continuaci n, se incluye una soluci n, pero existen otras:

```
1 // devuelve el reflejo de un punto en una lnea
2 void reflectionPoint(line l, point p, point &ans) {
3 point b;
4 closestPoint(l, p, b); // similar a distToLine
5 vec v = toVector(p, b); // crear un vector
6 ans = translate(translate(p, v), v); // trasladar p dos veces
```

Ejercicio 7.2.2.13: 63,43 grados.

**Ejercicio 7.2.2.14:** los puntos  $p$  (3,7) →  $q$  (11,13) →  $r$  (35,30) forman un giro a la derecha. Por lo tanto, el punto  $p$  se encuentra a la derecha de la línea que pasa por los puntos  $p$  y  $r$ . Si el punto  $r$  está en (35, 31), entonces  $p, q, r$  son colineales.

**Ejercicio 7.2.3.1:** ver la figura 7.14.

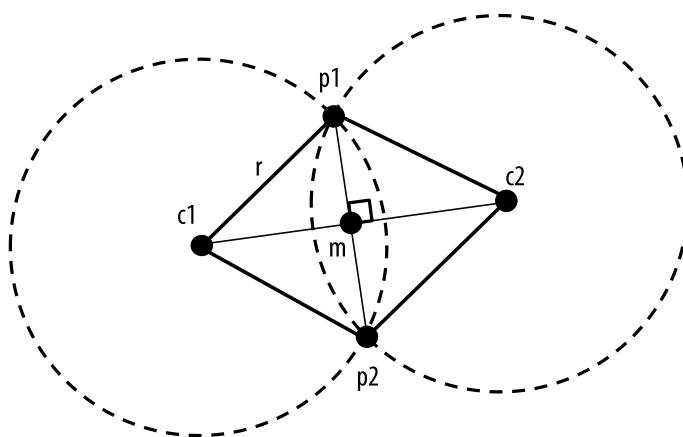


Figura 7.14: Explicación de un círculo que pasa por 2 puntos y un radio

Digamos que  $c_1$  y  $c_2$  son los centros de dos posibles círculos que pasan por dos puntos dados  $p_1$  y  $p_2$ , y tienen un radio  $r$ . El cuadrilátero  $p_1 - c_2 - p_2 - c_1$  es un rombo, ya que sus cuatro lados son iguales. Digamos que  $m$  es la intersección de las dos diagonales del rombo  $p_1 - c_2 - p_2 - c_1$ . Según la propiedad de un rombo,  $m$  es el bisector de las dos diagonales, y estas son perpendiculares entre ellas. Nos damos cuenta de que  $c_1$  y  $c_2$  se pueden calcular mediante un escalar a los vectores  $mp_1$  y  $mp_2$ , utilizando la razón apropiada ( $mc_1/mp_1$ ), para conseguir la misma magnitud que  $mc_1$  y, después, rotando los puntos  $p_1$  y  $p_2$  90 grados alrededor de  $m$ . En la implementación incluida en el **ejercicio 7.2.3.1**, la variable  $h$  es la *mitad* de la razón  $mc_1/mp_1$  (se puede verificar sobre el papel por qué  $h$  se puede calcular así). En las dos líneas que calculan las coordenadas de uno de los centros, los primeros operandos de las sumas son las coordenadas de  $m$ , mientras que los segundos operandos son el resultado de la escala y la rotación del vector  $mp_2$  alrededor de  $m$ .

**Ejercicio 7.2.4.1:** podemos utilizar tipos de datos `double`, que nos proporcionan el mayor rango. Sin embargo, para reducir el riesgo de un desbordamiento, también podemos reescribir la fórmula de Herón como  $A = \sqrt{s} \times \sqrt{s-a} \times \sqrt{s-b} \times \sqrt{s-c}$ . Sin embargo, el resultado será un poco menos preciso, ya que calculamos cuatro raíces cuadradas en lugar de una.

**Ejercicio 7.3.6.1:** intercambiar los puntos  $a$  y  $b$  al llamar a `cutPolygon(a, b, Q)`.

**Ejercicio 7.3.7.1:** editar la función `ccw` para que acepte puntos colineales.

**Ejercicio 7.3.7.2:** la función `atan2` calcula la tangente inversa de  $\frac{y}{x}$ , utilizando los signos de los argumentos para determinar correctamente el cuadrante.

## 7.5 Notas del capítulo

Algunos materiales de este capítulo están derivados de los que, cortésmente, nos ha cedido el **Dr. Cheng Holun, Alan**, de la Escuela de Informática de la Universidad Nacional de Singapur. Otras de las funciones de biblioteca provienen de las creadas por **Igor Naverniouk** (<http://shygypsy.com/tools/>) y han sido modificadas y extendidas, para incluir muchas otras funciones de geometría interesantes.

En comparación a la primera edición de este libro, este capítulo, al igual que los capítulos 5 y 6, ha crecido hasta aproximadamente el doble de su extensión original. Sin embargo, el material incluido dista mucho de estar completo, especialmente para los concursantes del ICPC. Si te estás preparando para este concurso, es una buena idea que dediques a una persona de tu equipo a estudiar este tema en profundidad. Este miembro debería dominar las fórmulas geométricas básicas y las técnicas avanzadas de geometría computacional, quizá leyendo los capítulo relativos a ello en los libros [47, 9, 7]. Pero no basta solo con la teoría, también debe prepararse para programar soluciones de geometría *robustas*, que puedan tratar con los casos especiales y los errores de precisión.

Las otras técnicas de geometría computacional que no hemos tratado en este capítulo, incluyen la técnica del **plano de barrer**, la intersección de otros **objetos geométricos**, incluyendo intersecciones segmento-segmento, varias soluciones de divide y vencerás para problemas de geometría clásicos, como el **problema del par más cercano**, el **problema del par más alejado**, el algoritmo de los **calibres giratorios**, etc. Tratamos algunos de estos problemas en el capítulo 9.

# Capítulo 8

---

## Materias más avanzadas

*El genio es un uno por ciento de inspiración  
y un noventa y nueve por ciento de esfuerzo.*

— Thomas Alva Edison

### 8.1 Introducción y motivación

La razón principal de la existencia de este capítulo es por organización. Las dos primeras secciones, contienen las materias más difíciles de los capítulos 3 y 4. En las secciones 8.2 y 8.3, trataremos las variantes y técnicas más complejas de los dos paradigmas de resolución de problemas más populares: la búsqueda completa y la programación dinámica. Si hubiésemos introducido este material en sus capítulos correspondientes, probablemente habríamos asustado a algunos lectores del libro.

La sección 8.4 nos introduce a problemas complejos, que requieren *más de un* algoritmo y/o estructura de datos para su solución. Igualmente, haber incluido estos aspectos en los capítulos anteriores, podría haber generado una cierta confusión. Consideramos más oportuno incorporarlos al presente capítulo, cuando ya hemos visto varias estructuras de datos y algoritmos (más fáciles). Por ello, es una buena idea comenzar leyendo los capítulos anteriores.

Volvemos a insistir al lector que trate de evitar la simple memorización de las soluciones pero, incluso más importante, que trate de entender las ideas clave que se introducen y que podrían ser de aplicación a otros problemas.

### 8.2 Técnicas de búsqueda más avanzadas

En la sección 3.2, hemos tratado varias técnicas (sencillas) de búsqueda completa iterativa y recursiva (*backtracking*). Sin embargo, algunos de los problemas más difíciles necesitan soluciones de búsqueda completa *más inteligentes*, para evitar un veredicto de tiempo límite superado (TLE). A continuación, veremos algunas de ellas con varios ejemplos.

## 8.2.1 Backtracking con máscara de bits

En la sección 2.2, hemos aprendido que se puede utilizar una máscara de bits para modelar un conjunto pequeño de booleanos. Las operaciones con máscaras de bits son muy ligeras y, por ello, siempre que tengamos la necesidad de utilizar un conjunto pequeño de booleanos, podemos considerar utilizarlas para acelerar nuestra solución (de búsqueda completa). En esta subsección, encontraremos dos ejemplos.

### El problema N-Queens, revisitado

En la sección 3.2.2, hemos visto el UVa 11195 - Another n-Queen Problem. Pero, incluso después de mejorar las comprobaciones de las diagonales izquierda y derecha, almacenando la disponibilidad de las  $n$  filas y las  $2 \times n - 1$  diagonales izquierda/derecha en tres `bitset`, seguimos recibiendo un veredicto TLE. Convertir los tres `bitset` a máscaras de bits puede ayudar un poco, pero seguirá siendo TLE.

Por suerte, hay una forma mejor de utilizar estas comprobaciones de fila, diagonal izquierda y diagonal derecha, como se describe a continuación. Esta formulación<sup>1</sup> nos permite un *backtracking* con máscara de bits eficiente. Utilizaremos directamente tres máscaras de bits para `rw`, `ld` y `rd`, para representar el estado de la búsqueda. Los bits activos en las máscaras de bits `rw`, `ld` y `rd` describen qué *filas* son atacadas en la *siguiente columna*, debido a los ataques en *fila*, *diagonal izquierda* o *diagonal derecha*, respectivamente, de las reinas colocadas anteriormente. Como valoramos una columna cada vez, solo habrá  $n$  diagonales izquierda/derecha posibles, por lo que podemos operar con tres máscaras de bits de la misma longitud de  $n$  bits (en comparación con los  $2 \times n - 1$  necesarios para las diagonales izquierda/derecha de la formulación que vimos en la sección 3.2.2).

Aunque ambas soluciones (la de la sección 3.2.2 y esta) utilizan la misma estructura de datos, tres máscaras de bits, la descrita aquí es mucho más eficiente. Esto pone de relieve la necesidad de que quien resuelva el problema, debe abordarlo desde diferentes puntos de vista.

Comenzamos mostrando el breve código de este *backtracking* recursivo con máscara de bits, para el problema N-queens (general), con  $n = 5$  y, después, explicamos cómo funciona.

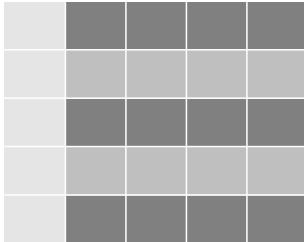
```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int ans = 0, OK = (1<<5) - 1; // comprobación para n = 5 reinas
5
6 void backtrack(int rw, int ld, int rd) {
7 if (rw == OK) { ans++; return; } // si todos los bits en 'rw' activos
8 int pos = OK & (~rw | ld | rd); // están disponibles los 1 en 'pos'
9 while (pos) { // este bucle es más rápido que 0(n)
10 int p = pos & -pos; // el menos significativo, esto es rápido
11 pos -= p; // desactivar ese bit
12 backtrack(rw|p, (ld|p)<<1, (rd|p)>>1); // inteligente
13 }
}
```

<sup>1</sup>Aunque esta solución está adaptada a este problema concreto, seguramente podremos utilizar partes de la misma en otros problemas.

```

14
15 int main() {
16 backtrack(0, 0, 0); // el punto de inicio
17 printf("%d\n", ans); // la respuesta debe ser 10 para n = 5
18 } // return 0;

```



`pos = 11111 & ~00000 = 11111 (p = 1)`

Figura 8.1: Problema 5 Queens: el estado inicial

Para  $n = 5$ , comenzamos con el estado  $(rw, ld, rd) = (0, 0, 0) = (00000, 00000, 00000)_2$ , que mostramos en la figura 8.1. La variable  $OK = (1 << 5) - 1 = (11111)_2$ , se utiliza como comprobación de la condición de finalización, y para ayudar a decidir qué filas están disponibles para una columna determinada. La operación  $pos = OK \& (\sim(rw \mid ld \mid rd))$ , *combina* la información de qué filas de la siguiente columna son atacadas por las reinas ya colocadas (con ataques por fila, diagonal izquierda o diagonal derecha), *niega* el resultado, y lo *combina* con  $OK$ , para obtener las filas *disponibles* para la siguiente columna. Inicialmente, todas las filas de la columna 0 están disponibles.

La búsqueda completa (el *backtracking* recursivo) probará con todas las filas posibles (esto es, todos los *bits activados* en la variable `pos`) de una columna determinada, de una en una. Antes, en la sección 3.2.1, hemos visto un método para explorar todos los bits activos en una máscara de bits en  $O(n)$ :

```

1 for (p = 0; p < n; p++) // O(n)
2 if (pos & (1<<p)) // si el bit 'p' está activo en 'pos'
3 // procesar p

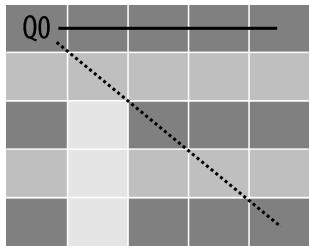
```

Sin embargo, este no es el método más eficiente. A medida que el *backtracking* recursivo gana en profundidad, cada vez habrá menos filas disponibles para seleccionar. En vez de probar con todas las  $n$  filas, podemos acelerar el bucle probando con todos los bits activos de la variable `pos`. El siguiente bucle se ejecuta en  $O(k)$ :

```

1 while (pos) { // O(k), donde k es el número de bits activos en 'pos'
2 int p = pos & -pos; // determinar el bit menos significativo de 'pos'
3 pos -= p; // desactivar ese bit
4 // procesar p
5 }

```



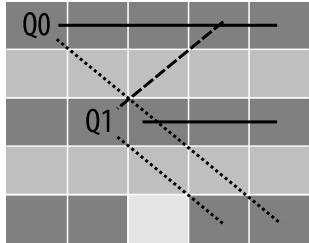
```

rw = 00001 (1)
1d = 00001 << 1 = 00010 (2)
rd = 00001 >> 1 = 00000|1 (0, LSB eliminado)
----- OR
00011 -> NEGADO -> 11100
pos = 11111 & 11100 = 11100 (p = 4)

```

Figura 8.2: Problema 5 Queens: después de colocar la primera reina

Volviendo a nuestra explicación, para  $\text{pos} = (11111)_2$ , comenzaremos con  $p = \text{pos} \& \neg\text{pos} = 1$ , o fila 0. Después de colocar la primera reina (reina 0) en la fila 0 de la columna 0, la fila 0 ya no estará disponible en la siguiente columna 1, y esto es reflejado rápidamente por la operación de bits  $\text{rw} \mid p$  (también  $1d \mid p$  y  $rd \mid p$ ). Aquí se encuentra la belleza de esta solución. Una diagonal izquierda/derecha aumenta/disminuye en uno, respectivamente, el número de fila que ataca, al cambiar a la siguiente columna. Una operación desplazamiento a izquierda y derecha,  $(1d \mid p) << 1$  y  $(rd \mid p) >> 1$ , puede gestionar este comportamiento con eficiencia. En la figura 8.2, vemos que para la siguiente columna 1, la fila 1 no está disponible, debido al ataque diagonal de la reina 0. Así que solo tendremos disponibles las filas 2, 3 y 4 para esa columna 1. Comenzaremos con la fila 2.



```

rw = 00101 (5)
1d = 00110 << 1 = 01100 (12)
rd = 00100 >> 1 = 00010 (2)
----- OR
01111 -> NEGADO -> 10000
pos = 11111 & 10000 = 10000 (p = 16)

```

Figura 8.3: Problema 5 Queens: después de colocar la segunda reina

Después de colocar la segunda reina (reina 1) en la fila 2 de la columna 1, las filas 0 (debido a la reina 0) y, ahora, 2, ya no están disponibles para la siguiente columna 2. La operación de desplazamiento a la izquierda, para la limitación de la diagonal izquierda, provoca que las filas 2 (debido a la reina 0) y, ahora, 3, no estén disponibles para la columna 2. Por lo tanto, solo nos queda libre la fila 4 para la columna 2, y esa es la que tendremos que elegir (ver la figura 8.3).

Después de colocar la tercera reina (reina 2) en la fila 4 de la columna 2, las filas 0 (por la reina 0), 2 (por la reina 1) y, ahora, 4, ya no están disponibles para la columna 3. La operación de desplazamiento a la izquierda, por la limitación de la diagonal izquierda, provoca que las filas 3 (por la reina 0) y 4 (por la reina 1) no estén disponibles para la columna 3 (no hay fila 5, el bit más significativo de  $1d$  no se utiliza). La operación de desplazamiento a la derecha, por la limitación de la diagonal derecha, provoca que las filas 0 (debido a la reina 1) y, ahora, 3 no estén disponibles para la columna 3. Al combinar toda esta información, resulta que solo la fila 1 está disponible para la columna 3, y es la que tendremos que elegir (ver la figura 8.4).

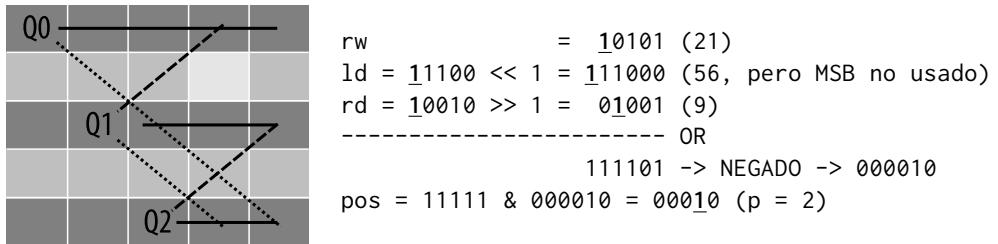


Figura 8.4: Problema 5 Queens: después de colocar la tercera reina

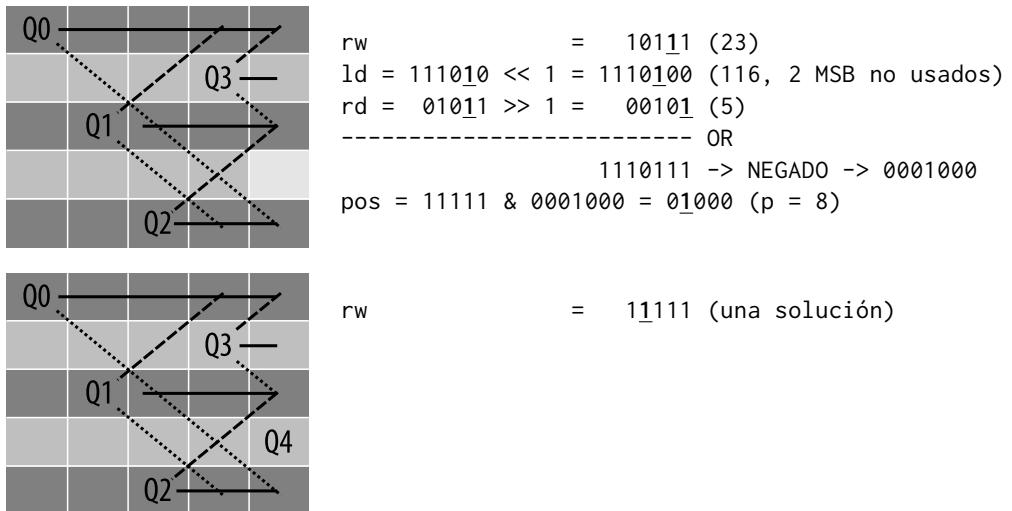


Figura 8.5: Problema 5 Queens: después de colocar las cuarta y quinta reinas

Se puede aplicar la misma explicación para las cuarta y quinta reinas (reinas 3 y 4), como se muestra en la figura 8.5. Podemos continuar el proceso para obtener las otras nueve soluciones para  $n = 5$ .

Mediante esta técnica, podemos resolver el problema UVa 11195. Basta con modificar el código anterior, para tomar en consideración las casillas prohibidas (que también se pueden modelar como máscaras de bits). Vamos a analizar brevemente el peor caso para un tablero  $n \times n$ , sin casillas prohibidas. Asumiendo que este *backtracking* recursivo con máscara de bits tiene, aproximadamente, dos filas disponibles menos en cada paso, tendremos una complejidad de tiempo de  $O(n!!)$ , donde  $n!!$  es la notación de multifactorial. Para  $n = 14$ , sin casillas prohibidas, la solución de *backtracking* recursivo de la sección 3.2.2 necesita hasta  $O(14!) \approx 87178M$  operaciones, mientras que el *backtracking* recursivo con máscara de bits presentado aquí, solo requiere unas  $O(14!!) = 14 \times 12 \times 10 \times \dots \times 2 = 645120$  operaciones.

## Estructura de datos de grafos de matriz de adyacencia compacta

El problema UVa 11065 - Gentlemen Agreement se reduce al cálculo de dos enteros, el número de conjuntos independientes y el tamaño del conjunto independiente máximo (MIS, ver la definición del problema en la sección 4.7.4) de un grafo *general* dado, con  $V \leq 60$ . Buscar el MIS de un grafo general es un problema NP-complejo. Por lo tanto, parece poco probable que exista un algoritmo polinómico para este problema.

Una solución es el siguiente *backtracking* recursivo inteligente. El estado de la búsqueda es una 3-tupla,  $(i, usado, profundidad)$ . El parámetro  $i$  implica que podemos considerar que los vértices en  $[i..V-1]$  están incluidos en el conjunto independiente. El segundo parámetro, *usado*, es una máscara de longitud  $V$  bits, que indica qué vértices ya no están disponibles para el conjunto independiente actual, porque al menos uno de sus vecinos ya ha sido incluido en el conjunto independiente. El tercer parámetro, *profundidad*, guarda la profundidad de la recursión, que también es el tamaño del conjunto independiente actual.

Hay un truco muy audaz con máscaras de bits, que podemos utilizar para acelerar significativamente la solución para este problema. Vemos que el grafo de entrada es pequeño,  $V \leq 60$ . Por lo tanto, podemos almacenar este grafo en una matriz de adyacencia de tamaño  $V \times V$  (para este problema, establecemos como verdaderas todas las celdas a lo largo de la diagonal principal de esa matriz). Sin embargo, podemos comprimir *una fila* de  $V$  booleanos ( $V \leq 60$ ) en una máscara de bits, utilizando un entero de 64 bits con signo.

Con esta matriz de adyacencia compacta *AdjMat* (compuesta de solo  $V$  filas de enteros de 64 bits con signo), podemos utilizar una operación rápida de máscara de bits para etiquetar eficientemente a los vecinos de los vértices. Si decidimos tomar un vértice libre  $i$ , es decir,  $(usado \& (1 << i)) == 0$ , aumentamos la *profundidad* en uno y utilizamos una operación de máscara de bits en  $O(1)$ , *usado | AdjMat[i]*, para etiquetar a *todos* los vecinos de  $i$ , incluyendo al propio  $i$  (*AdjMat[i]* también es una máscara de bits de longitud  $V$ , con el bit  $i$ -ésimo activo).

Cuando todos los bits de la máscara *usado* estén activos, habremos encontrado otro conjunto independiente. También guardamos el valor más grande de *profundidad* a lo largo del proceso, ya que ese será el tamaño del conjunto independiente máximo del grafo de entrada. A continuación, las secciones clave del código:

```
1 void rec(int i, long long used, int depth) {
2 if (used == (1<<V) - 1) { // se visitan todas las intersecciones
3 nS++; // otro conjunto posible
4 mxS = max(mxS, depth); // tamaño del conjunto
5 }
6 else {
7 for (int j = i; j < V; j++)
8 if (!(used & (1<<j))) // si la intersección j no se ha utilizado
9 rec(j+1, used|AdjMat[j], depth+1); // operación de bits rápida
10 }
11 }
12
13 // dentro de int main()
14 // una matriz de adyacencia más potente, con bits (para mayor velocidad)
15 for (int i = 0; i < V; i++)
```

```

16 AdjMat[i] = (1<<i); // i a sí mismo
17 for (int i = 0; i < E; i++) {
18 scanf("%d %d", &a, &b);
19 AdjMat[a] |= (1<<b);
20 AdjMat[b] |= (1<<a);
21 }

```

### Ejercicio 8.2.1.1\*

El Sudoku es otro problema NP-completo. El *backtracking* recursivo para encontrar una solución a un Sudoku estándar de  $9 \times 9$  ( $n = 3$ ), se puede acelerar utilizando una máscara de bits. Para cada celda vacía  $(r, c)$ , intentamos poner un dígito  $[1 \dots n^2]$ , de uno en uno, si es un movimiento válido. Las comprobaciones de  $n^2$  filas,  $n^2$  columnas y  $n^2$  cuadrados, se pueden hacer con tres máscaras de longitud  $n^2$  bits. Resuelve dos problemas similares, UVa 989 y UVa 10957, mediante esta técnica.

### 8.2.2 Backtracking con poda intensa

El problema I - ‘Robots on Ice’, de la final mundial del ICPC 2010, se puede interpretar como una ‘prueba de esfuerzo en estrategia de poda’. El enunciado es sencillo: dado un tablero de  $M \times N$  con 3 puntos de control  $\{A, B, C\}$ , encontrar un camino hamiltoniano<sup>2</sup> de longitud  $M \times N$ , desde la coordenada  $(0, 0)$  hasta la  $(0, 1)$ . El camino hamiltoniano debe tocar los tres puntos de control  $A$ ,  $B$  y  $C$  a un cuarto, la mitad y tres cuartos de su recorrido, respectivamente. Límites:  $2 \leq M, N \leq 8$ .

Ejemplo: si tenemos el siguiente tablero de  $3 \times 6$  con  $A = (\text{fila, columna}) = (2, 1)$ ,  $B = (2, 4)$  y  $C = (0, 4)$ , como en la figura 8.6, existen dos caminos posibles.

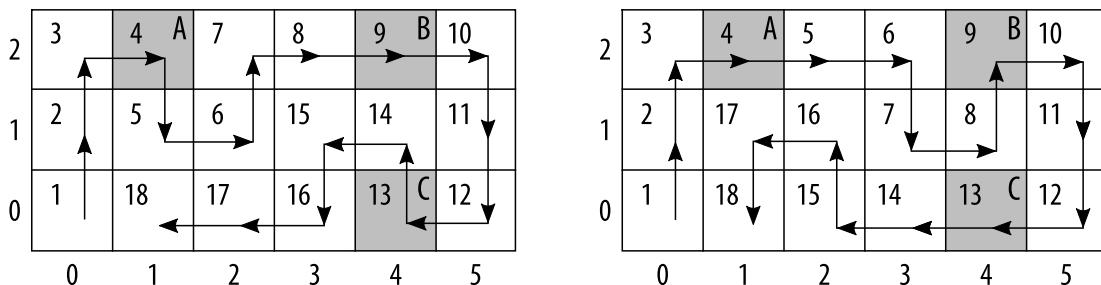


Figura 8.6: Visualización de UVa 1098 - Robots on Ice

Un algoritmo ingenuo de *backtracking* recursivo resultará en un veredicto TLE, ya que hay 4 opciones en cada paso, y la longitud máxima del camino es  $8 \times 8 = 64$ , en el caso de prueba más grande. Probar los  $4^{64}$  caminos posibles es inasumible. Para acelerar el algoritmo, debemos podar el espacio, si la búsqueda:

<sup>2</sup>Un camino hamiltoniano es aquel que, en un grafo no dirigido, visita todos los vértices una sola vez.

1. Abandona los límites de la rejilla  $M \times N$  (obviamente).
2. No toca los puntos de control correspondientes en las distancias de  $1/4$ ,  $1/2$  o  $3/4$ . En realidad, la presencia de estos tres puntos de control *reduce* el espacio de búsqueda.
3. Toca los puntos de control antes del momento oportuno.
4. No llegará a tiempo al siguiente punto de control, desde la posición actual.
5. No podrá alcanzar determinadas coordenadas, ya que el camino parcial actual bloquea el acceso a las mismas. Esto se puede comprobar con una DFS/BFS sencilla (sección 4.2). Primero, ejecutamos DFS/BFS desde la coordenada de destino  $(0, 1)$ . Si hay coordenadas en la rejilla  $M \times N$  que *no* son alcanzables desde  $(0, 1)$  y *todavía no han sido visitadas* por el camino parcial actual, podemos podar ese camino parcial.

### Ejercicio 8.2.2.1\*

Las cinco estrategias de poda que hemos mencionado en esta subsección son buenas pero, en realidad, insuficientes para cumplir con el límite de tiempo de los problemas LA 4793 y UVa 1098. Existe una solución más rápida, que utiliza la técnica de encuentro en el medio (sección 8.2.4). Este ejemplo ilustra que la elección del límite de tiempo puede determinar qué soluciones de búsqueda completa se consideran lo suficientemente rápidas. Estudia la idea de la técnica de encuentro en el medio de la sección 8.2.4 y aplícala para resolver el problema de Robots on Ice.

## 8.2.3 Búsqueda estado-espacio con BFS o Dijkstra

En las secciones 4.2.2 y 4.4.3, hemos tratado dos algoritmos de grafos estándar, utilizados para resolver el problema de los caminos más cortos de origen único (SSSP). Se puede utilizar BFS si el grafo no es ponderado, mientras que emplearemos Dijkstra si tenemos un grafo ponderado. Los problemas de SSSP enumerados en el capítulo 4 siguen siendo fáciles, en el sentido de que, en la mayoría de los casos, podemos ver fácilmente ‘el grafo’ en el enunciado del problema. Pero esto ya no es cierto en algunos de los problemas de búsqueda en grafos más difíciles, que aparecen en esta sección, donde los grafos, normalmente implícitos, no son tan evidentes y los estados/vértices pueden ser objetos complejos. En ese caso, normalmente denominaremos la búsqueda como de ‘estado-espacio’, en vez de SSSP.

Cuando el estado es un objeto complejo (como un par de datos (posición, máscara de bits) en UVa 321 - The New Villa, una 4-tupla (fila, columna, dirección, color) en UVa 10047 - The Monocycle, etc.), normalmente no usamos `vector<int> dist` para almacenar la información de distancia, como en las implementaciones estándar de BFS o Dijkstra. Esto se debe a que esos estados no se convierten bien a índices enteros. Una vía es utilizar `map<TIPO-VÉRTICE, int> dist` en su lugar. Este truco añade un (pequeño) factor  $\log V$  a la complejidad de tiempo de BFS o Dijkstra. Pero para una búsqueda estado-espacio compleja, este tiempo de ejecución adicional puede ser aceptable si, con ello, reducimos la complejidad general del código. En esta subsección, mostramos un ejemplo de una de esas búsquedas estado-espacio complejas.

### Ejercicio 8.2.3.1

¿Cómo almacenar TIPO-VÉRTICE en C++ y en Java, cuando es un par, una 3-tupla o una 4-tupla de información?

### Ejercicio 8.2.3.2

Misma pregunta que en el **ejercicio 8.2.3.1**, pero cuando TIPO-VÉRTICE es un objeto mucho más complejo, como un *array*.

### Ejercicio 8.2.3.3

¿Es posible considerar la búsqueda estado-espacio como un problema de maximización?

#### UVa 11212 - Editing a Book

Enunciado resumido del problema: dados  $n$  párrafos, numerados de 1 a  $n$ , hay que colocarlos en orden 1, 2, ...,  $n$ . Con la ayuda de un portapapeles, puedes presionar Ctrl-X (cortar) y Ctrl-V (pegar), varias veces. No puedes cortar dos veces antes de pegar, pero puedes cortar varios párrafos contiguos al mismo tiempo, y luego se pegarán en el mismo orden. ¿Cuál es el número mínimo de pasos necesarios?

Ejemplo 1: para ordenar  $\{2, 4, (1), 5, 3, 6\}$ , cortamos el párrafo (1) y lo pegamos antes del 2, para obtener  $\{1, 2, 4, 5, (3), 6\}$ . Despues, cortarmos el párrafo (3) y lo pegamos antes del 4, para llegar a  $\{1, 2, 3, 4, 5, 6\}$ . La respuesta es dos pasos.

Ejemplo 2: para ordenar  $\{(3, 4, 5), 1, 2\}$ , cortamos tres párrafos al mismo tiempo, (3, 4, 5), y los pegamos después del 2, para obtener  $\{1, 2, 3, 4, 5\}$ . Aquí lo hemos resuelto con un solo paso. La solución no es única, pues existe la alternativa de cortar simultáneamente (1, 2) y pegarlos antes del párrafo 3, para obtener igualmente  $\{1, 2, 3, 4, 5\}$ , también en un solo paso.

El límite superior aproximado del número de pasos necesarios para reordenar estos  $n$  párrafos es  $O(k)$ , donde  $k$  es el número de párrafos que están descolocados inicialmente. Esto es debido a que podemos utilizar el siguiente algoritmo ‘trivial’ (e incorrecto): cortar un solo párrafo que esté descolocado y pegarlo en su posición correcta. Después de  $k$  operaciones de cortar y pegar, seguro que tendremos el texto bien colocado. Pero este podría no ser el camino más corto.

Por ejemplo, el algoritmo ‘trivial’ mencionado procesará  $\{5, 4, 3, 2, 1\}$  de la siguiente manera:  $\{(5), 4, 3, 2, 1\} \rightarrow \{(4), 3, 2, 1, 5\} \rightarrow \{(3), 2, 1, 4, 5\} \rightarrow \{(2), 1, 3, 4, 5\} \rightarrow \{1, 2, 3, 4, 5\}$ , para un total de 4 operaciones cortar-pegar. Esto no es óptimo, pues podemos resolver este caso en solo 3 pasos:  $\{5, 4, (3, 2), 1\} \rightarrow \{3, (2, 5), 4, 1\} \rightarrow \{3, 4, (1, 2), 5\} \rightarrow \{1, 2, 3, 4, 5\}$ .

Este problema tiene un espacio de búsqueda tan *enorme* que, incluso para una instancia con un pequeño  $n = 9$ , es casi imposible resolverlo a mano. Casi seguro que ni siquiera intentaremos dibujar el árbol de recursiones para verificar que necesitamos, al menos, 4 pasos para ordenar  $\{5, 4, 9, 8, 7, 3, 2, 1, 6\}$  y, al menos 5, para hacerlo con  $\{9, 8, 7, 6, 5, 4, 3, 2, 1\}$ .

El estado de este problema es una *permutación* de párrafos. Hay, como mucho,  $O(n!)$  permutaciones posibles. Con un  $n = 9$  máximo, según el enunciado del problema, esto es  $9!$  o 362880. Por lo tanto, el número de vértices del grafo estado-espacio tampoco es tan grande.

La dificultad del problema reside en el número de *arestas* del grafo estado-espacio. Dada una permutación de longitud  $n$  (un vértice), hay  $nC_2$  puntos de corte posibles (índice  $i, j \in [1..n]$ ) y  $n$  puntos posibles para pegar (índice  $k \in [1..(n - (j - i + 1))]$ ). Por lo tanto, cada uno de los  $O(n!)$  vértices tiene conectadas, aproximadamente,  $O(n^3)$  aristas.

En realidad, el problema pedía el camino más corto desde el vértice de origen/estado (la permutación de entrada) al vértice de destino (una permutación ordenada), en este grafo estado-espacio no ponderado, pero enorme. El peor caso, si ejecutamos una BFS sencilla en  $O(V + E)$  en este grafo, es  $O(n! + (n! \times n^3)) = O(n! \times n^3)$ . Para  $n = 9$ , esto es  $9! \times 9^3 = 264539520 \approx 265M$  operaciones. Casi con toda certeza la solución recibirá un veredicto TLE (o MLE). Necesitamos una solución mejor, que encontraremos en la sección 8.2.4.

### 8.2.4 Encuentro en el medio (búsqueda bidireccional)

En algunos problemas de SSSP (pero normalmente de búsqueda estado-espacio) en grafos enormes de los que conocemos dos vértices, el vértice/estado de origen  $s$  y el vértice/estado de destino  $t$ , podemos reducir de forma *significativa* la complejidad de tiempo de búsqueda, si realizamos esta desde *ambas direcciones* y esperamos *encontrarnos en el medio*. Ilustraremos esta técnica continuando con el difícil problema UVa 11212.

Antes de seguir, debemos poner de relieve que la técnica de encuentro en el medio no se refiere siempre a una BFS bidireccional. Se trata de una estrategia de resolución de problemas de ‘búsqueda desde dos direcciones/lugares’, que puede aparecer en diferentes formas en otros problemas de búsqueda complejos, ver el **ejercicio 3.2.1.4\***.

#### UVa 11212 - Editing a Book (revisitado)

Aunque la complejidad de tiempo en el peor caso de búsqueda estado-espacio de este problema es mala, la respuesta más grande posible es pequeña. Cuando ejecutamos BFS en el caso de prueba más grande, con  $n = 9$ , desde el estado de destino  $t$  (la permutación ordenada  $\{1, 2, \dots, 9\}$ ), para alcanzar todos los otros estados, encontramos que, en este caso, la profundidad máxima de la BFS para  $n = 9$  es solo de 5 (después de ejecutarla durante *unos minutos*, lo que significa TLE en un concurso).

Esta importante información nos permite realizar una BFS bidireccional, eligiendo llegar hasta solo la profundidad 2 desde cada dirección. Aunque conocer este dato no es necesario para ejecutar una BFS bidireccional, ayuda a reducir el espacio de búsqueda.

A continuación, tratamos tres casos posibles:

- Caso 1: el vértice  $s$  está a dos pasos del vértice  $t$  (ver la figura 8.7).

Comenzamos ejecutando BFS (profundidad máxima de la BFS = 2) desde el vértice de destino  $t$ , para poblar la información de distancia desde  $t$ : `dist_t`. Si ya hemos encontrado el vértice de origen  $s$ , es decir, si `dist_t[s]` no es INF, devolvemos este valor. Las respuestas posibles son 0 (si  $s = t$ ), 1 o 2 pasos.

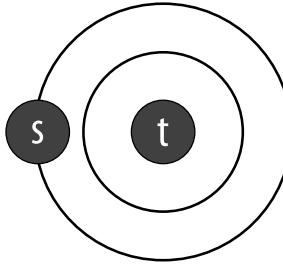


Figura 8.7: Caso 1: ejemplo cuando  $s$  está a dos pasos de  $t$

- Caso 2: el vértice  $s$  está a tres o cuatro pasos del vértice  $t$  (ver la figura 8.8).

Si no hemos sido capaces de encontrar el vértice de origen  $s$  después del caso 1 anterior, es decir,  $\text{dist}_t[s] = \text{INF}$ , sabremos que  $s$  se encuentra más alejado del vértice  $t$ . Ahora, ejecutamos BFS desde el vértice de origen  $s$  (también con profundidad máxima de la BFS = 2), para poblar la información de distancia desde  $s$ :  $\text{dist}_s$ . Si encontramos un vértice común  $v$  ‘en el medio’, durante la ejecución de esta segunda BFS, sabremos que el vértice  $v$  está alejado dos capas de los vértices  $t$  y  $s$ . Entonces, la respuesta será  $\text{dist}_s[v] + \text{dist}_t[v]$  pasos. Las respuestas posibles son 3 o 4 pasos.

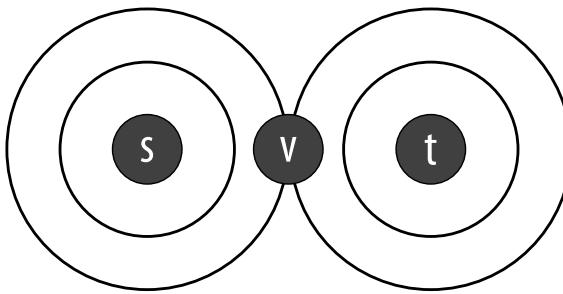


Figura 8.8: Caso 2: ejemplo cuando  $s$  está a cuatro pasos de  $t$

- Caso 3: el vértice  $s$  está a, exactamente, cinco pasos del vértice  $t$  (ver la figura 8.9).

Si no somos capaces de encontrar un vértice  $v$  común, después de ejecutar la segunda BFS en el caso 2, la respuesta serán, claramente, los 5 pasos que ya conocíamos, pues sabemos que  $s$  y  $t$  siempre son alcanzables. Detenernos en la profundidad 2 nos evitará calcular la 3, que requiere *mucho más tiempo*.

Hemos visto que, dada una permutación de longitud  $n$  (un vértice), tendremos unas  $O(n^3)$  ramas, en este enorme grafo estado-espacio. Sin embargo, si ejecutamos cada BFS con una profundidad máxima de 2, solo deberemos procesar un máximo de  $O((n^3)^2) = O(n^6)$  operaciones por BFS. Con  $n = 9$ , esto son  $9^6 = 531441$  operaciones (mayor que  $9!$ , pues hay algunas superposiciones). Como el vértice de destino  $t$  no cambia durante la búsqueda estado-espacio, podemos comenzar calculando la BFS desde el vértice de destino  $t$  una sola vez. Después, calculamos la segunda BFS desde el vértice  $s$  para cada consulta. Nuestra implementación de la BFS tendrá un factor log adicional, debido al uso de una estructura de datos de tabla (como `map`), para almacenar `dist_t` y `dist_s`. Ahora la solución será aceptada.

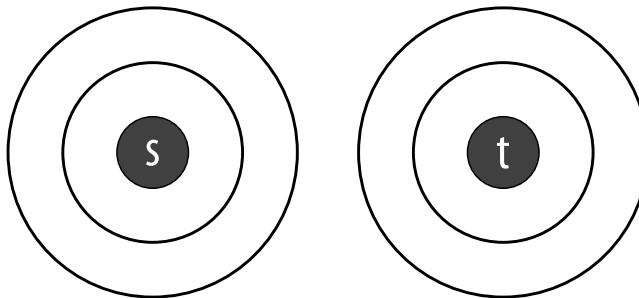


Figura 8.9: Caso 3: ejemplo cuando  $s$  está a cinco pasos de  $t$

### 8.2.5 Búsqueda informada: A\* e IDA\*

#### Conceptos básicos de A\*

Los algoritmos de búsqueda completa que hemos visto en los capítulos 3 y 4, y en las subsecciones anteriores son ‘no informados’, es decir, todos los estados posibles alcanzables desde el estado actual son *igualmente válidos*. En algunos problemas, tenemos acceso a más información (de ahí el nombre de ‘búsqueda informada’), y podemos utilizar la búsqueda A\*, que utiliza métodos heurísticos, para ‘orientar’ la dirección de la búsqueda.

Ilustraremos esta búsqueda A\* utilizando el conocido problema del juego del 15. Tenemos 15 piezas deslizables en el juego, numeradas del 1 al 15. Estas 15 piezas están encerradas en un marco de  $4 \times 4$ , con un hueco vacío. Los movimientos posibles consisten en deslizar las piezas adyacentes al hueco a ese espacio vacío. Otra forma de ver estos movimientos es: “deslizar el *hueco vacío* a derecha, arriba, izquierda o abajo”. El fin del juego es ordenar las piezas, para que queden igual que en la figura 8.10, el estado ‘objetivo’.

Este pequeño juego es un auténtico quebradero de cabeza para varios algoritmos, debido a su gigantesco espacio de búsqueda. Podemos representar un estado del juego mediante una lista de los números de las piezas, fila a fila, de izquierda a derecha, en un *array* de 16 enteros. Para simplificarlo, le asignaremos el valor 0 a la casilla vacía, por lo que el estado objetivo es {1, 2, 3, ..., 14, 15, 0}. Desde un estado donde puede haber hasta 4 estados alcanzables, dependiendo de la posición del hueco, hay  $2/3/4$  acciones posibles si la casilla vacía está en alguna de las 4 esquinas/8 casillas que no son esquinas/4 casillas del medio, respectivamente. El espacio de búsqueda es enorme.

Sin embargo, no todos los estados son igual de buenos. Hay una interesante característica heurística en este problema, que puede ayudar a guiar el algoritmo de búsqueda, que es la suma de las distancias Manhattan<sup>3</sup> entre cada pieza (no vacía), desde estado actual y su ubicación en el estado objetivo. Esta información establece el límite inferior de pasos necesarios para resolver el juego. Al combinar el coste alcanzado (indicado por  $g(s)$ ) y el valor heurístico (indicado por  $h(s)$ ) de un estado  $s$ , podemos tener una idea mejor de cuál será el siguiente movimiento. Lo ilustraremos con el juego en el siguiente estado inicial  $A$ :



Figura 8.10: Juego del 15

<sup>3</sup>La distancia Manhattan entre dos puntos es la suma de las diferencias absolutas de sus coordenadas.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{0}} \\ 13 & 14 & 15 & \underline{\mathbf{12}} \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \underline{\mathbf{0}} \\ 9 & 10 & 11 & \underline{\mathbf{8}} \\ 13 & 14 & 15 & 12 \end{bmatrix} C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \underline{\mathbf{0}} & \underline{\mathbf{11}} \\ 13 & 14 & 15 & 12 \end{bmatrix} D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{12}} \\ 13 & 14 & 15 & \underline{\mathbf{0}} \end{bmatrix}$$

El coste del estado inicial  $A$  es  $g(s) = 0$ , porque no ha habido movimientos. Hay tres estados  $\{B, C, D\}$  alcanzables desde  $A$ , con  $g(B) = g(C) = g(D) = 1$ , es decir, un movimiento. Pero estos tres estados *no son* igual de buenos:

1. El valor heurístico, si deslizamos la pieza 0 hacia arriba, es  $h(B) = 2$ , ya que las piezas 8 y 12 están ambas desplazadas por 1. Esto provoca  $g(B) + h(B) = 1 + 2 = 3$ .
2. El valor heurístico, si deslizamos la pieza 0 a la izquierda, es  $h(C) = 2$ , ya que las piezas 11 y 12 están ambas desplazadas por 1. Esto provoca  $g(C) + h(C) = 1 + 2 = 3$ .
3. Pero si deslizamos la pieza 0 hacia abajo, tenemos  $h(D) = 0$ , ya que todas las piezas están en su posición correcta. Esto provoca  $g(D) + h(D) = 1 + 0 = 1$ , la combinación más baja.

Si visitamos los estados, en orden ascendente, de los valores  $g(s) + h(s)$ , exploraremos primero los estados con menor coste esperado, es decir, el estado  $D$  en este ejemplo, que es el estado objetivo. Esta es la esencia del algoritmo de búsqueda A\*.

Normalmente implementamos esta ordenación de estados con la ayuda de una cola de prioridad, lo que hace que la implementación de la búsqueda A\* sea muy similar a la del algoritmo de Dijkstra, que vimos en la sección 4.4. Si establecemos  $h(s)$  a 0 para todos los estados, A\* degenera al algoritmo de Dijkstra.

Mientras la función heurística  $h(s)$  no sobreestime la distancia real al estado objetivo (también conocida como **heurística admisible**), este algoritmo de búsqueda A\* es óptimo. La parte más difícil en la resolución de problemas de búsqueda utilizando A\* es encontrar esa heurística.

### Limitaciones de A\*

El problema con A\* (y también con BFS y Dijkstra, cuando se utilizan en grafos estado-espacio grandes) al usar una cola (de prioridad), es que los requisitos de memoria pueden ser immensos, si el estado objetivo está alejado del inicial. En algunos problemas de búsqueda difíciles, tendremos que recurrir a las siguientes técnicas relacionadas.

### Búsqueda de profundidad limitada

En la sección 3.2.2 hemos visto el algoritmo de *backtracking* recursivo. El principal problema con el *backtracking* puro es que se puede ver atrapado en la exploración de un camino muy profundo, que no llevará a la solución, antes de volver a la posición correcta, habiendo malgastado un tiempo precioso.

La búsqueda de profundidad limitada (DLS), pone límite a la profundidad a la que puede llegar el *backtracking*. DLS impide avanzar cuando la profundidad de la búsqueda supera el límite establecido. Si ese límite resulta ser igual al de la profundidad del estado objetivo más superficial, DLS será más rápida que la versión general del *backtracking*. Sin embargo, si el límite es demasiado pequeño, el estado objetivo será inalcanzable. Si el problema indica que el estado objetivo está ‘como mucho a  $d$  pasos’ del estado inicial, es mejor utilizar DLS que *backtracking*.

### Búsqueda de profundidad iterativa

Si la DLS no se usa bien, el estado objetivo será inalcanzable, incluso con la solución. En general, la DLS no se utiliza sola, sino como parte de una búsqueda de profundidad iterativa (IDS).

La IDS llama a la DLS con un *límite creciente*, hasta que se encuentra el estado objetivo. La IDS es, por lo tanto, completa y óptima. Es una buena estrategia que deja a un lado el problema de determinar el mejor límite de profundidad, al probarlos todos de forma creciente. Comienza con profundidad 0 (el propio estado inicial), sigue con 1 (los que son alcanzables a un solo paso del estado inicial), después 2, etc. Al hacerlo, la IDS combina, esencialmente, los beneficios de poca carga de trabajo y uso de memoria de la DFS y la capacidad de la BFS de visitar todos los estados vecinos, capa a capa (ver la tabla 4.2 en la sección 4.2).

Aunque la IDS llama a la DLS muchas veces, la complejidad de tiempo sigue siendo de  $O(b^d)$ , donde  $b$  es el factor de ramificación y  $d$  la profundidad del estado objetivo más superficial. La razón:  $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + \dots + (b^0 + b^1 + b^2 + \dots + b^d)) \leq O(c \times b^d) = O(b^d)$ .

### A\* de profundidad iterativa (IDA\*)

Para resolver más rápidamente el juego del 15, podemos utilizar el algoritmo IDA\* (A\* de profundidad iterativa) que es, esencialmente, una IDS con DLS modificada. IDA\* llama a la DLS modificada, para probar todos los estados vecinos, en un orden predefinido (es decir, deslizar la pieza 0 a derecha, arriba, izquierda y, finalmente, abajo, siempre en ese orden y sin usar una cola de prioridad). Esta DLS modificada no se detiene cuando ha superado el límite de profundidad, sino cuando su  $g(s) + h(s)$  supera la mejor solución conocida hasta el momento. IDA\* expande el límite gradualmente, hasta que llega al estado objetivo.

La implementación de IDA\* no es sencilla e invitamos al lector a que desentrañe nuestro código.



### Ejercicio 8.2.5.1\*

Una de las partes más difíciles de resolver problemas de búsqueda utilizando A\*, es encontrar la heurística admisible correcta y calcularla de forma eficiente, ya que debe ser repetida muchas veces. Haz una lista de las heurísticas admisibles que se utilizan habitualmente en problemas de búsqueda complejos que utilicen el algoritmo A\*, y muestra cómo calcularlas eficientemente. Una de ellas es la distancia Manhattan, como hemos visto en esta sección.

## Ejercicio 8.2.5.2\*

Resuelve el problema UVa 11212 - Editing a Book, que hemos tratado en profundidad en las secciones 8.2.3 y 8.2.4, con A\* en vez de con BFS bidireccional. Consejo: empieza por determinar cuál es una heurística apropiada para este problema.

## Ejercicios de programación

Ejercicios de programación que se resuelven con técnicas de búsqueda más avanzadas:

### Problemas de *backtracking* más complicados

1. *UVa 00131 - The Psychic Poker Player*

(*backtracking* con máscara de bits de  $2^5$ , para decidir qué carta se queda en la mano/se intercambia con la superior del mazo; usar  $S!$  permutaciones para mezclar las 5 cartas de la mano y lograr el mejor valor)

2. *UVa 00710 - The Game*

(*backtracking* con *memoización/poda*)

3. *UVa 00711 - Dividing up*

(reduce el espacio de búsqueda; *backtracking*)

4. *UVa 00989 - Su Doku*

(Sudoku clásico; este problema es NP-completo, pero este caso se puede resolver mediante *backtracking* con poda; usa una máscara de bits para acelerar la comprobación de dígitos disponibles)

(LA 3565 - WorldFinals SanAntonio06; *backtracking* con algún tipo de máscara de bits)

5. *UVa 01052 - Bit Compression*

(la primera fila por fuerza bruta en  $2^{10}$ , el resto va seguido)

6. ***UVa 10309 - Turn the Lights Off*** \*

(el orden no es importante, por lo que podemos probar a pulsar los botones en orden creciente, fila a fila, columna a columna; pulsar un botón solo afecta al cuadrado  $3 \times 3$  a su alrededor; por lo tanto, después de pulsar el botón  $(i, j)$ , la luz  $(i - 1, j - 1)$  estará encendida (ya que ningún botón posterior le afectará); esta comprobación se puede usar para podar el *backtracking*)

7. *UVa 10318 - Security Panel*

(parece DP, pero el estado, con máscaras de bits, no se puede *memoizar*; por suerte la rejilla es pequeña)

(muy similar a UVa 989; si puedes resolver ese, solo tienes que modificar un poco el código para este)

10. ***UVa 11065 - A Gentlemen's Agreement*** \*

(conjunto independiente; la máscara de bits ayuda a acelerar la solución; consultar la sección 8.2.1)

11. *UVa 11127 - Triple-Free Binary Strings*

(*backtracking* con máscara de bits)

12. ***UVa 11195 - Another n-Queen Problem*** \*

(ver el ***ejercicio 3.2.1.3\**** y la sección 8.2.1)

13. *UVa 11464 - Even Parity*

(primera fila por fuerza bruta en  $2^{15}$ ; el resto va seguido)

14. *UVa 11471 - Arrange the Tiles*

(reducir el espacio de búsqueda agrupando títulos del mismo tipo; *backtracking* recursivo)

### Búsqueda estado-espacio con BFS o Dijkstra más complicada

1. *UVa 00321 - The New Villa*

(s: (posición, máscara de bits  $2^{10}$ ); mostar el camino)

2. UVa 00658 - It's not a Bug ...
3. UVa 00928 - Eternal Truths
4. **UVa 00985 - Round and Round ... \***  
 (s: máscara de bits indicando si el error existe o no; usar Dijkstra, ya que el grafo estado-espacio es ponderado)  
 (s: (fila, columna, dirección, paso))  
 (4 rotaciones es lo mismo que ninguna; s: (fila, columna, rotacion = [0..3]); encontrar el camino más corto desde el estado [1][1][0] al [R][C][x], donde  $0 \leq x \leq 3$ )  
 (LA 3570 - WorldFinals SanAntonio06; usar Floyd Warshall para obtener la información APSP; después modelar el problema original como un problema de SSSP ponderados, que se resuelve con Dijkstra)
5. UVa 01057 - Routing  
 (LA 4637 - Tokyo09; SSSP que se resuelve con BFS)
6. UVa 01251 - Repeated Substitution ...
7. UVa 01253 - Infected Land
8. UVa 10047 - The Monocycle
9. **UVa 10097 - The Color game**
10. UVa 10923 - Seven Seas
11. **UVa 11198 - Dancing Digits \***
12. **UVa 11329 - Curious Fleas \***  
 (s: (fila, columna, dirección, color); BFS)  
 (s: (N1, N2); grafo no ponderado implícito: BFS)  
 (s: (posición\_barco, ubicación\_de\_enemigos, ubicación\_de\_obstáculos, pasos\_realizados); grafo ponderado implícito: Dijkstra)  
 (s: (permutación); BFS; difícil de programar)  
 (s: (máscara de 26 bits); 4 para la posición del dado en la rejilla  $4 \times 4$ , 16 para determinar si hay una pulga en la casilla, 6 para las caras del dado que tienen una pulga; usar map; programación tediosa)  
 (invertir los roles de origen y destino)  
 (BFS en grafo no ponderado implícito)  
 (LA 4201 - Dhaka08; similar a UVa 11974)
13. UVa 11513 - 9 Puzzle
14. UVa 11974 - Switch The Lights
15. UVa 12135 - Switch Bulbs

#### Encuentro en el medio/A\*/IDA\*

1. UVa 00652 - Eight
2. **UVa 01098 - Robots on Ice \***  
 (juego clásico de 8 bloques deslizantes; IDA\*)  
 (LA 4793 - WorldFinals Harbin10; ver la sección 8.2.2; sin embargo, tiene una solución 'encuentro en el medio', más rápida)
3. UVa 01217 - Route Planning  
 (LA 3681 - Kaohsiung06; se resuelve con A\*/IDA\*; el caso de prueba probablemente cuenta con solo 15 paradas, incluyendo la de origen y la de destino de la ruta)
4. **UVa 10181 - 15-Puzzle Problem \***  
 (similar a UVa 652, pero más grande; podemos usar IDA\*)
5. UVa 11163 - Jaguar King  
 (otro juego que se resuelve con IDA\*)
6. **UVa 11212 - Editing a Book \***  
 (encuentro en el medio; ver sección 8.2.4)

Ver también otros problemas de búsqueda completa en la sección 8.4.

## 8.3 Técnicas de DP más avanzadas

En las secciones 3.5, 4.7.1, 5.4, 5.6 y 6.5, hemos visto una introducción a la técnica de la programación dinámica (DP), varios problemas clásicos de DP y sus soluciones, y una breve introducción a problemas de DP no clásicos. Existen otras técnicas de DP más avanzadas, que no han quedado cubiertas en esas secciones. A continuación, presentamos algunas de ellas.

### 8.3.1 DP con máscara de bits

Algunos de los problemas de DP modernos, necesitan un (pequeño) conjunto de booleanos como parámetro del estado de DP. Nos encontramos ante otra situación en la que las máscaras de bits puede resultar útiles (ver también la sección 8.2.1). Esta técnica es válida para la DP, ya que el entero, representado por la máscara de bits, se puede utilizar como índice de la tabla de DP. Nos hemos acercado a este método al tratar el problema del viajante con DP (sección 3.5.2). Aquí mostramos otro ejemplo.

#### UVa 10911 - Forming Quiz Teams

Para leer el enunciado resumido del problema y el código que lo soluciona, consulta el primer problema mencionado en el capítulo 1. El grandilocuente nombre de este problema es “emparejamiento perfecto de peso mínimo en un grafo ponderado general pequeño”. En su variante general, este problema es difícil. Sin embargo, si el tamaño de la entrada es pequeño, de hasta  $M \leq 20$ , es posible utilizar una solución de DP con máscara de bits.

La solución de DP con máscara de bits de este problema es sencilla. El estado de emparejamiento se representa con un **bitmask**. Lo ilustraremos con un pequeño ejemplo, cuando  $M = 6$ . Comenzamos en un estado en el que todavía no ha habido emparejamientos, es decir, **bitmask=000000**. Si los elementos 0 y 2 se emparejan, podemos activar dos bits (0 y 2) al mismo tiempo, mediante la sencilla operación **bitmask | (1<<0) | (1<<2)**, para que el estado pase a ser **bitmask=000101**. Recordemos que el índice empieza desde 0 y se cuenta por la derecha. Si, desde este estado, el siguiente emparejamiento es entre los elementos 1 y 5, el nuevo estado será **bitmask=100111**. El emparejamiento perfecto se producirá cuando el estado solo esté compuesto por unos, en este caso: **bitmask=111111**.

Aunque hay muchas formas de llegar a un estado determinado, solo existen  $O(2^M)$  estados diferentes. Para cada uno de ellos, registramos el peso mínimo de los emparejamientos anteriores que hay que realizar para llegar a ese punto. Buscamos el emparejamiento perfecto. En primer lugar, buscar un bit ‘inactivo’  $i$ , utilizando un bucle en  $O(M)$ . Después, buscamos el siguiente mejor bit ‘inactivo’  $j$  desde  $[i+1..M-1]$ , utilizando otro bucle en  $O(M)$  y, recursivamente, emparejamos  $i$  y  $j$ . Esta comprobación se vuelve a realizar utilizando operaciones de bits, es decir, podemos verificar **if (! (bitmask & (1<<i)))** (igual para  $j$ ). El algoritmo se ejecuta en  $O(M \times 2^M)$ . En el problema UVa 10911,  $M = 2N$  y  $2 \leq N \leq 8$ , por lo que esta solución de DP con máscara de bits es viable. Estudia el código para conocer el resto de detalles.



ch8\_02\_UVa10911.cpp



ch8\_02\_UVa10911.java

En esta subsección, hemos visto que es posible utilizar DP con máscara de bits para resolver instancias pequeñas ( $M \leq 20$ ) de emparejamientos en un grafo general. Como norma genérica, la técnica de máscara de bits nos permite representar un conjunto pequeño, de hasta  $\approx 20$  elementos. Los ejercicios de programación de esta sección contienen más ejemplos en los que se utiliza una máscara de bits como *uno de los parámetros* del estado de DP.

### Ejercicio 8.3.1.1

Muestra la solución de DP con máscara de bits necesaria, si tenemos que tratar con “coincidencia de cardinalidad máxima en un grafo general pequeño ( $V \leq 18$ )”.

### Ejercicio 8.3.1.2\*

Reescribe el código `ch8_02_UVa10911.cpp/java` con el truco `LSOne`, mostrado en la sección 8.2.1, para acelerarlo.

## 8.3.2 Recopilación de parámetros comunes (de DP)

Tras haber resuelto un buen número de problemas de DP (incluyendo *backtracking* recursivo sin *memoización*), los concursantes desarrollarán un instinto sobre los parámetros que se eligen, normalmente, para representar los estados de los problemas de DP (o *backtracking* recursivo). Estos son algunos de ellos:

1. Parámetro: el índice  $i$  es un *array*, como  $[x_0, x_1, \dots, x_i, \dots]$ .  
Transición: extender el *subarray*  $[0..i]$  (o  $[i..n-1]$ ), procesar  $i$ , tomar el elemento  $i$  o no, etc.  
Ejemplo: suma máxima unidimensional, parte de la mochila 0-1, TSP, etc., (sección 3.5.2).
2. Parámetros: índices  $(i, j)$  en dos *arrays*, como  $[x_0, x_1, \dots, x_i] + [y_0, y_1, \dots, y_j]$ .  
Transición: extender  $i, j$  o ambos, etc.  
Ejemplo: alineación de cadenas/distancia de edición, LCS, etc., (sección 6.5).
3. Parámetro: *subarray*  $(i, j)$  de un *array*  $[..., x_i, x_{i+1}, \dots, x_j, ...]$ .  
Transición: separar  $(i, j)$  en  $(i, k) + (k+1, j)$ , o en  $(i, i+k) + (i+k+1, j)$ , etc.  
Ejemplo: multiplicación de cadenas de matrices (sección 9.20), etc.
4. Parámetro: un vértice (posición) en un DAG (normalmente implícito).  
Transición: procesar los vecinos de este vértice, etc.  
Ejemplo: número de caminos o caminos más corto/largo en un DAG (sección 4.7.1), etc.
5. Parámetro: parámetro de estilo mochila.  
Transición: reducir (o aumentar) el valor actual hasta llegar a 0 (o al umbral), etc.  
Ejemplo: mochila 0-1, sumar subconjuntos, cambio de monedas (sección 3.5.2), etc.  
Nota: este parámetro no es muy adecuado para la DP, si el rango es muy amplio.  
Ver consejos en la sección 8.3.3 si el valor del parámetro puede ser negativo.
6. Parámetro: conjunto pequeño (normalmente utilizando máscaras de bits).  
Transición: etiquetar uno o más elementos del conjunto como activos (o no), etc.  
Ejemplo: TSP con DP (sección 3.5.2), DP con máscara de bits (sección 8.3.1), etc.

Los problemas más difíciles de DP combinan, normalmente, dos o más parámetros para representar distintos estados. Intenta resolver los problemas de DP que incluimos en esta sección, para mejorar tus habilidades con DP.

### 8.3.3 Gestión de valor de parámetros negativos con técnica de desplazamiento

En algunos casos, poco habituales, el rango de un parámetro utilizado en un estado de DP puede ser negativo. Esto provoca un problema en la solución de DP, ya que asignamos los valores de los parámetros a índices de una tabla de DP. Los índices de una tabla de DP no deben ser negativos. Por suerte, podemos resolver esta cuestión utilizando una técnica de desplazamiento, para hacer que los índices no sean negativos. Mostramos esta técnica con otro problema de DP complicado: Free Parentheses.

#### UVa 1238 - Free Parentheses (ACM ICPC Jakarta08, LA 4143)

Enunciado resumido del problema: recibes una expresión aritmética sencilla, que consta solo de los operadores de *suma* y *resta*, por ejemplo,  $1 - 2 + 3 - 4 - 5$ . Puedes poner *paréntesis* en la expresión, libremente y en la posición que quieras, mientras siga siendo *válida*. ¿Cuántos números *diferentes* puedes componer? La respuesta para la expresión anterior es de 6:

$$\begin{array}{lll} 1 - 2 + 3 - 4 - 5 & = & -7 \\ 1 - (2 + 3) - 4 - 5 & = & -13 \\ 1 - (2 + 3 - 4) - 5 & = & -5 \end{array} \quad \begin{array}{lll} 1 - (2 + 3 - 4 - 5) & = & 5 \\ 1 - 2 + 3 - (4 - 5) & = & 3 \\ 1 - (2 + 3) - (4 - 5) & = & -3 \end{array}$$

El problema establece los siguientes límites: la expresión consta de solo  $2 \leq N \leq 30$  números no negativos menores que 100, separados por operadores de suma o resta. No hay ningún operador antes del primer número, ni después del último.

Para resolver este problema, son necesarias tres observaciones:

1. Solo necesitamos abrir un paréntesis después de un signo ‘-’ (negativo) para invertir el significado de los siguientes operadores ‘+’ y ‘-’.
2. Solo podemos cerrar  $X$  paréntesis si ya los hemos abierto antes. Debemos llevar la cuenta, para procesar los subproblemas correctamente.
3. El valor máximo es  $100 + 100 + \dots + 100$  (100 repetido 30 veces) = 3000, y el valor mínimo es  $0 - 100 - \dots - 100$  (un 0 seguido de 29 veces -100) = -2900. También tendremos que almacenar esta información, como veremos más adelante.

Para resolver este problema utilizando DP, debemos determinar qué conjunto de parámetros del problema representarán los distintos estados. Los más fáciles de identificar son estos dos:

1. ‘idx’, la posición actual que estamos procesando, necesitamos saber dónde estamos.
2. ‘abiertos’, número de paréntesis abiertos para generar una expresión válida<sup>4</sup>.

Pero estos dos parámetros no son suficientes para identificar de forma única el estado. Por ejemplo, la expresión parcial ‘1-1+1-1...’ tiene  $idx = 3$  (los índices 0, 1, 2 y 3 han sido procesados),  $abiertos = 0$  (no se pueden cerrar paréntesis), lo que suma 0. Ocurre que ‘1-(1+1-1)...’ tiene

<sup>4</sup>En  $idx = N$  (hemos procesado el último número), es correcto que  $abiertos > 0$ , ya que podemos cerrar todos los paréntesis que sea necesario al final de la expresión, por ejemplo:  $1 - (2 + 3 - (4 - (5)))$ .

los mismos `idx = 3`, `abiertos = 0` y también suma 0. Pero ‘`1-(1+1)-1..`’ tiene los mismos `idx = 3`, `abiertos = 0` y suma -2. Así que estos dos parámetros de DP *no* identifican un estado de forma única. Necesitamos otro parámetro para distinguirlos, el valor ‘`val`’. Esta habilidad para identificar el conjunto correcto de parámetros, para representar los distintos estados, es algo que hay de desarrollar para resolver correctamente problemas de DP. El código y la explicación:

```

1 void rec(int idx, int open, int val) {
2 if (visited[idx][open][val+3000]) // ya se ha llegado a este estado
3 return; // truco de +3000 para convertir índices negativos a [200..6000]
4 // los índices negativos no son adecuados para un array estático
5 visited[idx][open][val+3000] = true; // fijar estado al que llegar
6
7 if (idx == N) // último número, el valor actual es un resultado
8 used[val+3000] = true, return; // posible de la expresión
9
10 int nval = val + num[idx] * sig[idx] * ((open%2 == 0) ? 1 : -1);
11 if (sig[idx] == -1) // opción 1: abrir paréntesis solo si el signo es -
12 rec(idx+1, open+1, nval); // sin efecto si el signo es +
13 if (open > 0) // opción 2: cerrar paréntesis, solo podemos hacerlo
14 rec(idx+1, open-1, nval); // si ya tenemos paréntesis abiertos
15 rec(idx+1, open, nval); // opción 3: normal, no hacer nada
16 }
17
18 // Procesamiento previo: crear un array 'used' que, inicialmente, se fija
19 // como falso, ejecutar esta DP de arriba a abajo llamando a rec(0, 0, 0)
20 // La solución es el número de valores en 'used' que son verdaderos

```

Como vemos en el código anterior, es posible representar todos los estados posibles del problema con un *array* tridimensional: `bool visited[idx][open][val]`. El objeto de esta tabla `visited` es indicar si cierto estado ha sido visitado, o no. Como el rango de ‘`val`’ se encuentra entre -2900 y 3000 (5901 valores diferentes), debemos desplazarlo para evitar que sea negativo. En este ejemplo, utilizamos una constante segura de +3000. El número de estados es  $30 \times 30 \times 6001 \approx 5M$ , con un tiempo de procesamiento de  $O(1)$  por estado. Es lo bastante rápido.

### 8.3.4 ¿MLE? Prueba a utilizar un BST equilibrado como tabla recordatoria

En la sección 3.5.2, hemos visto un problema de DP, la mochila 0-1, donde el estado es  $(id, remW)$ . El parámetro `id` tiene un rango de  $[0..n-1]$  y `remW` de  $[0..S]$ . Si el autor del problema decide que  $n \times S$  sea de un tamaño suficiente, provocará que el *array* bidimensional (para la tabla de DP) de tamaño  $n \times S$ , sea demasiado grande (MLE en concursos).

Por suerte, para un problema como el de la mochila 0-1, si ejecutamos DP de arriba a abajo, nos daremos cuenta de que no todos los estados se visitan (mientras que sí se hará con la versión de DP de abajo a arriba). Por lo tanto, podemos intercambiar tiempo de ejecución por un menor espacio, utilizando un BST equilibrado (`map` en la STL de C++ o `TreeMap` en Java), como tabla recordatoria. Este BST equilibrado *solo* registrará los estados que han sido visitados por la DP de arriba a abajo. Por lo tanto, si solo hay  $k$  estados visitados, podemos utilizar un espacio de  $O(k)$ , en vez de  $n \times S$ . El tiempo de ejecución de la DP de arriba a abajo aumenta por un factor

de  $O(c \times \log k)$ . Sin embargo, hay que tener en cuenta que este truco es útil pocas veces, debido al gran factor constante  $c$  implicado.

### 8.3.5 ¿MLE/TLE? Usa una mejor representación de estados

Nuestra solución de DP ‘correcta’ (que obtiene la respuesta correcta, pero utiliza más recursos de cálculo) podría obtener un veredicto de límite de memoria superado (MLE) o límite de tiempo superado (TLE), si el autor del problema ha utilizado una mejor representación de estados y ha establecido límites mayores en la entrada. Si esto ocurre, no tendremos más remedio que buscar una representación de estados de DP mejor, para reducir el tamaño de la tabla de DP (y, en consecuencia, acelerar la complejidad de tiempo total). Ilustramos esta técnica con un ejemplo:

**UVa 1231 - ACORN (ACM ICPC Singapore07, LA 4106)**

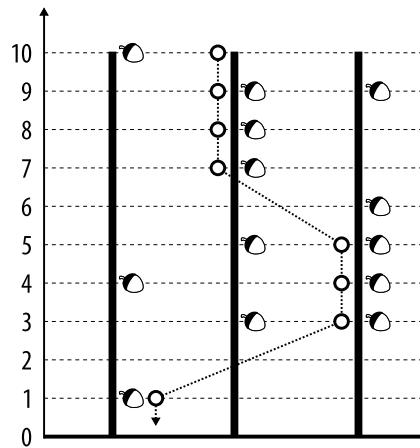


Figura 8.11: El camino de descenso

Enunciado resumido del problema: dados  $t$  robles, la altura  $h$  de *todos* los árboles, la altura  $f$  que la ardilla Jayjay pierde cuando salta de un árbol a otro,  $1 \leq t, h \leq 2000$ ,  $1 \leq f \leq 500$  y las posiciones de las bellotas en cada uno de los robles, `bellota[árbol][altura]`, determinar el número máximo de bellotas que puede recoger Jayjay en *un solo descenso*. Por ejemplo, si  $t = 3, h = 10, f = 2$  y `bellota[árbol][altura]` es como se muestra en la figura 8.11, el mejor camino de descenso tiene un total de 8 bellotas (ver la línea punteada).

Una solución ingenua de DP utiliza una tabla `total[árbol][altura]`, que almacena las mejores bellotas posibles a recoger cuando Jayjay se encuentra en un árbol determinado, a una altura determinada. Después, Jayjay intenta bajar una unidad (-1) en el *mismo* árbol o salta ( $-f$ ) unidades a  $t-1$  robles *diferentes*, desde su posición. En el caso más grande, se requieren  $2000 \times 2000 = 4M$  estados y  $4M \times 2000 = 8000M$  operaciones. Este método resultará claramente en un veredicto de TLE.

Una solución de DP mejor es la siguiente. En realidad, podemos ignorar la información: “en qué árbol está ahora Jayjay”, ya que *memoizar* el mejor de todos será suficiente. Esto es debido a que saltar a cualquier otro de los  $t-1$  robles, reduce la altura de Jayjay de la misma forma. Establecer

una tabla  $dp[altura]$ , que almacene las mejores bellotas posibles para recoger cuando Jayjay está a esa altura. El código de DP de abajo a arriba, que solo necesita  $2000 = 2K$  estados, y una complejidad de tiempo de  $2000 \times 2000 = 4M$ , es el siguiente:

```
1 for (int tree = 0; tree < t; tree++) // inicialización
2 dp[h] = max(dp[h], acorn[tree][h]);
3
4 for (int height = h-1; height >= 0; height--)
5 for (int tree = 0; tree < t; tree++) {
6 acorn[tree][height] +=
7 max(acorn[tree][height+1], // desde este árbol, +1 por encima
8 ((height+f <= h) ? dp[height+f] : 0)); // desde el árbol en altura+f
9 dp[height] = max(dp[height], acorn[tree][height]); // actualizar esto
10 }
11
12 printf("%d\n", dp[0]); // la solución se almacena aquí
```



ch8\_03\_UVa1231.cpp



ch8\_03\_UVa1231.java

Cuando el tamaño de los estados de DP ingenuos es tan grande que provoca que la complejidad de tiempo total no sea viable, hay que pensar en otra forma más eficiente (aunque menos obvia) de representarlos. Utilizar una buena representación de estados, mejora la velocidad potencial en una solución de DP. Recuerda que ningún problema en un concurso de programación es irresoluble, por lo que el autor ha tenido que utilizar algún mecanismo para ello.

### 8.3.6 ¿MLE/TLE? Abandona un parámetro, recuperándolo de los otros

Otro truco conocido para reducir el uso de memoria de una solución de DP (y, con ello, acelerar la solución), es abandonar un parámetro importante, que se pueda recuperar utilizando el resto. Utilizaremos un problema de la final mundial del ICPC para ilustrar esta técnica.

#### UVa 1099 - Sharing Chocolate (Final mundial ACM ICPC de Harbin 2010, LA 4794)

Enunciado resumido del problema: tenemos una gran tableta de chocolate, de tamaño  $1 \leq w, h \leq 100$ ,  $1 \leq n \leq 15$  amigos y el tamaño de la petición de cada amigo. ¿Podremos dividir el chocolate, utilizando cortes horizontales y verticales, de forma que cada amigo reciba *una porción* del tamaño deseado?

Por ejemplo, veamos la parte izquierda de la figura 8.12. El tamaño de la tableta de chocolate original tiene  $w = 4$  y  $h = 3$ . Si hay 4 amigos que piden porciones de chocolate de tamaños  $\{6, 3, 2, 1\}$ , respectivamente, podremos dividir el chocolate en 4 trozos, utilizando los 3 cortes que se muestran en la parte derecha de la misma figura.

Los concursantes que ya estén familiarizados con la programación dinámica, pensarán rápidamente en las siguientes ideas: en primer lugar, si la suma de todas las peticiones no es igual a

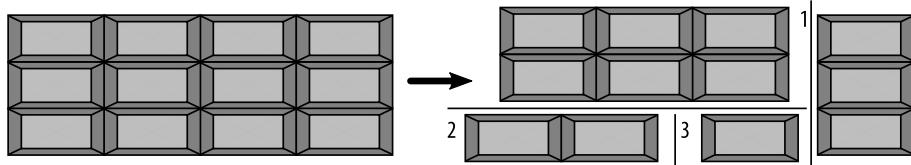


Figura 8.12: Ilustración de ACM ICPC WF2010 - J - Sharing Chocolate

$w \times h$ , no habrá solución. En caso contrario, podemos representar un estado único de este problema, utilizando tres parámetros:  $(w, h, \text{máscara\_de\_bits})$ , donde  $w$  y  $h$  son las dimensiones actuales de la tableta que estamos considerando, y  $\text{máscara\_de\_bits}$  el subconjunto de amigos que ya han recibido una porción de chocolate del tamaño deseado. Sin embargo, un análisis rápido nos mostrará que esto necesita una tabla de DP de tamaño  $100 \times 100 \times 2^{15} = 327M$ . Es demasiado para un concurso de programación.

Existe una mejor representación de estados con solo dos parámetros, o  $(w, \text{máscara\_de\_bits})$  o  $(h, \text{máscara\_de\_bits})$ . Sin perder la generalidad, elegiremos  $(w, \text{máscara\_de\_bits})$ . Gracias a esta, podemos ‘recuperar’ el valor  $h$  necesario mediante  $\text{sum}(\text{máscara\_de\_bits}) / w$ , donde  $\text{sum}(\text{máscara\_de\_bits})$  es la suma de los tamaños de las porciones solicitadas por los amigos ya satisfechos en  $\text{máscara\_de\_bits}$  (es decir, todos los bits ‘activos’ de  $\text{máscara\_de\_bits}$ ). De esta forma, tendremos todos los parámetros necesarios:  $w, h$  y  $\text{máscara\_de\_bits}$ , pero solo utilizaremos una tabla de DP de tamaño  $100 \times 2^{15} = 3M$ . Esta versión es viable.

Casos base: si  $\text{máscara\_de\_bits}$  solo contiene un bit ‘activo’ y el tamaño de la porción de chocolate solicitada por esa persona es igual a  $w \times h$ , tenemos una solución. En caso contrario, no la tendremos.

Casos generales: si tenemos una tableta de chocolate de tamaño  $w \times h$  y un conjunto de amigos satisfechos  $\text{máscara\_de\_bits} = \text{máscara\_de\_bits}_1 \cup \text{máscara\_de\_bits}_2$ , podemos realizar un corte horizontal, o vertical, para dividirlo en dos trozos que satisfagan a los amigos en  $\text{máscara\_de\_bits}_1$  y  $\text{máscara\_de\_bits}_2$ .

La complejidad de tiempo del peor caso de este problema sigue siendo enorme pero, con la poda adecuada, la solución se ejecutará dentro del límite de tiempo.

### Ejercicio 8.3.6.1\*

Resolver los problemas UVa 10482 - The Candyman Can y UVa 10626 - Buying Coke, que utilizan esta técnica. Determinar qué parámetro nos conviene más abandonar, pero podremos recuperar gracias a los otros.

A parte de varios problemas de DP en la sección 8.4, encontrarás otros pocos en el capítulo 9, que no hemos incluido aquí por considerarlos *poco habituales*. Son:

1. Sección 9.3: problema del viajante bitónico (insistimos en la técnica de ‘abandonar un parámetro y recuperarlo de otros’).
2. Sección 9.5: problema del cartero chino (otro uso de DP con máscara de bits, para resolver el emparejamiento perfecto de peso mínimo en un grafo ponderado general pequeño).

3. Sección 9.20: multiplicación de cadenas de matrices (un problema de DP clásico).
4. Sección 9.21: potencia de matrices (podemos acelerar las transiciones de DP en *algunos* problemas de DP raros de  $O(n)$  a  $O(\log n)$ , reescribiendo las recurrencias de DP como multiplicación de matrices).
5. Sección 9.22: conjunto independiente ponderado máximo (en un árbol), con DP.
6. Sección 9.33: la estructura de datos de tabla dispersa utiliza DP.

## Ejercicios de programación

**Ejercicios de programación relativos a DP más avanzada:**

**DP de nivel 2 (un poco más difícil que los ejercicios de los capítulos 3, 4, 5 y 6)**

1. **UVa 01172 - The Bridges of ... \*** (LA 3986 - SouthWesternEurope07; DP no clásica; un poco de emparejamiento de grafos pero con límites)
2. **UVa 01211 - Atomic Car Race \*** (LA 3404 - Tokyo05; calcular previamente el array  $T[L]$ , el tiempo para ejecutar un camino de longitud  $L$ ; DP con un parámetro  $i$ , donde  $i$  es el punto de control donde cambiamos la rueda; si  $i = n$ , no cambiamos la rueda) (usar BigInteger de Java)
3. UVa 10069 - Distinct Subsequences (usar double)
4. UVa 10081 - Tight Words (se puede utilizar máscara de bits)
5. UVa 10364 - Square (mostrar camino; primos)
6. *UVa 10419 - Sum-up the Primes* (modelar como máscara de bits el tablero de  $4 \times 4$  y las 48 agujas posibles; entonces se convierte en un juego sencillo de dos jugadores; ver la sección 5.8) (problema pequeño; se puede resolver con *backtracking*)
7. *UVa 10536 - Game of Euler* (suma de subconjuntos con DP; con técnica de desplazamiento de negativos; además, matemáticas sencillas)
8. UVa 10651 - Pebble Solitaire (similar a DP + máscara de bits; almacenar estado como entero)
9. *UVa 10690 - Expression Again* (visto en esta sección)
10. UVa 10898 - Combo Deal (similar a UVa 10911, pero con emparejamiento de tres personas en un equipo)
11. **UVa 10911 - Forming Quiz Teams \*** (DP interesante; s: (id, val); usar desplazamiento para números negativos; t: más o menos; mostrar solución)
12. *UVa 11088 - End up with More Teams* (se puede resolver con búsqueda completa)
13. UVa 11832 - Account Book (notar que las esferas  $> n$  son inútiles)
14. UVa 11218 - KTV
15. *UVa 12324 - Philip J. Fry Problem*

**DP de nivel 3**

1. UVa 00607 - Scheduling Lectures (devuelve par de datos)
2. *UVa 00702 - The Vindictive Coach* (el DAG implícito no es trivial)
3. *UVa 00812 - Trade on Verweggistan* (mezcla de DP y voraz)
4. UVa 00882 - The Mailbox ... (s: (bajo, alto, buzón\_restante); probar todos)
5. **UVa 01231 - ACORN \*** (LA 4106 - Singapore07; DP con reducción de dimensión; en esta sección)
6. **UVa 01238 - Free Parentheses \*** (LA 4143 - Jakarta08; autor: Felix Halim; tratado en esta sección)

7. UVa 01240 - ICPC Team Strategy  
 8. UVa 01244 - Palindromic paths  
 9. *UVa 10029 - Edit Step Ladders*  
 10. *UVa 10032 - Tug of War*  
 11. *UVa 10154 - Weights and Measures*  
 12. UVa 10163 - Storage Keepers  
 13. UVa 10164 - Number Game  
 14. UVa 10271 - Chopsticks  
 15. UVa 10304 - Optimal Binary ...  
 16. *UVa 10604 - Chemical Reaction*  
 17. *UVa 10645 - Menu*  
 18. UVa 10817 - Headmaster's Headache  
 19. *UVa 11002 - Towards Zero*  
 20. *UVa 11084 - Anagram Division*  
 21. UVa 11285 - Exchange Rates  
 22. **UVa 11391 - Blobs in the Board \***  
 23. *UVa 12030 - Help the Winners*
- (LA 4146 - Jakarta08)  
 (LA 4336 - Amritapuri08; almacenar el mejor camino entre  $i, j$ ; la tabla de DP contiene cadenas)  
 (usar map como tabla recordatoria)  
 (mochila con DP, con optimizaciones para evitar TLE)  
 (variante de LIS)  
 (probar todas las líneas seguras L posibles y ejecutar DP; s: (id, N\_restante); t: contratar/ignorar 'id' de persona para buscar en almacenamiento K)  
 (un poco de teoría de números (módulo); backtracking; memoización del estado de la DP: (suma, tomado))  
 (el tercer palillo puede ser cualquiera, debemos seleccionar el palillo adyacente de forma voraz; estado DP: (pos, k\_restante); transición: ignorar este palillo, o tomar este palillo y el inmediatamente siguiente, entonces desplazarse a pos + 2; podar estados inviables cuando no haya palillos suficientes para formar ternas)  
 (DP clásica; requiere suma de rangos unidimensional y la aceleración de Knuth-Yao para lograr una solución en  $O(n^2)$ )  
 (la mezcla de puede hacer con cualquier par de productos químicos, hasta que solo queden dos; hay que memoizar los productos restantes, con la ayuda de map; ordenar los productos restantes ayuda a aumentar el número de coincidencias en la tabla recordatoria)  
 (s: (días\_restantes, presupuesto\_restante, plato\_anterior, cuenta\_plato\_anterior); los dos primeros parámetros son del estilo de la mochila; los dos últimos se utilizan para determinar el precio del plato ya que, si se sirve como primero, segundo o posterior, tiene un coste distinto)  
 (s: (id, máscara de bits); espacio:  $100 \times 2^{2 \times 8}$ )  
 (DP sencilla; usar técnica de desplazamiento de negativos)  
 (usar next\_permutation/fuerza bruta no parece la mejor opción; se puede resolver con DP)  
 (mantener los mejores valores de CAD y USD de cada día)  
 (DP con máscara de bits en rejilla bidimensional)  
 (s: (idx, máscara de bits, todos1, tiene2); t: probar todos los zapatos no emparejados con la chica que ha elegido el vestido 'idx')

#### DP de nivel 4

1. UVa 00473 - Raucous Rockers  
 2. **UVa 01099 - Sharing Chocolate \***  
 3. **UVa 01220 - Party at Hali-Bula \***  
 4. UVa 01222 - Bribery FIPA  
 5. **UVa 01252 - Twenty Questions \***  
 6. *UVa 10149 - Yahtzee*
- (el límite de la entrada no está claro; por lo tanto, utilizar un vector redimensionable y estados compactos)  
 (LA 4794 - WorldFinals Harbin10; tratado en esta sección)  
 (LA 3794 - Tehran06; problema de conjunto independiente máximo (MIS) en un árbol; comprobar también si el MIS es único)  
 (LA 3797 - Tehran06; DP en un árbol)  
 (LA 4643 - Tokyo09; DP, s: (máscara de bits 1, máscara de bits 2) donde la máscara de bits 1 describe las características que decidimos preguntar y la 2 las respuesta obtenidas)  
 (DP con máscara de bits; usa reglas de cartas; tedioso)

- |                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7. UVa 10482 - The Candyman Can<br>8. UVa 10626 - Buying Coke<br>9. UVa 10722 - Super Lucky Numbers<br><br>10. UVa 11125 - Arrange Some Marbles<br>11. UVa 11133 - Eigensequence<br>12. UVa 11432 - Busy Programmer<br>13. UVa 11472 - Beautiful Numbers | (ver el <a href="#">ejercicio 8.3.6.1*</a> )<br>(ver el <a href="#">ejercicio 8.3.6.1*</a> )<br>(necesita BigInteger de Java; la formulación de la DP debe ser eficiente, para evitar TLE; estado: ( <i>N_dígitos_restantes</i> , <i>B</i> , primero, <i>dígito_anterior_es_uno</i> ) y utilizar un poco de combinatoria sencilla)<br>(conteo de caminos en DAG implícito; DP de 8 dimensiones)<br>(el DAG implícito no es trivial)<br>(el DAG implícito no es trivial)<br>(estado de la DP con cuatro parámetros) |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Ver también otros problemas de DP en la sección 8.4 y el capítulo 9

## 8.4 Descomposición de problemas

Aunque solo hay ‘unos pocos’ algoritmos y estructuras de datos básicos, que forman parte de los problemas de concursos de programación (creemos que hemos cubierto la mayoría en este libro), los problemas más difíciles pueden necesitar una *combinación* de dos (o más) de ellos. Para resolverlos, primero debemos descomponer esos problemas en componentes que abordaremos de forma independiente. Y, para poder hacerlo, debemos familiarizarnos con esos componentes individuales (todo el contenido desde el capítulo 1 hasta la sección 8.3).

Hay  $N C_2$  combinaciones posibles de dos algoritmos y/o estructuras de datos de entre los  $N$  totales, pero no todas tienen sentido. En esta sección hemos recopilado una lista de algunas<sup>5</sup> de las combinaciones *más comunes* de dos algoritmos y/o estructuras de datos, basándonos en nuestra experiencia resolviendo  $\approx 1675$  problemas del juez UVa. Finalizaremos la sección tratando la, casi inexistente, combinación de *tres* algoritmos y/o estructuras de datos.

### 8.4.1 Dos componentes: búsqueda binaria de la respuesta y otro

En la sección 3.3.1, hemos visto la búsqueda binaria de la respuesta en un problema de simulación (sencillo), que no depende de los preciosos algoritmos que aparecen después. En realidad, esta técnica se puede combinar con algunos algoritmos de las secciones 3.4 y 8.3. Algunas de las variantes que hemos detectado, combinan la búsqueda binaria de la respuesta con:

- Algoritmo voraz (sección 3.4), UVa 714, 11516, 11670.
- Comprobación de conectividad de un grafo (sección 4.2), UVa 295, 10876.
- Algoritmo de los SSSP (sección 4.4), UVa 10537, 10816, IOI 2009 (Mecho).
- Algoritmo de flujo máximo (sección 4.6), UVa 10983.
- Algoritmo de la MCBM (sección 4.7.4), UVa 1221, 10804, 11262.
- Operaciones con BigInteger (sección 5.3), UVa 10606.
- Fórmulas geométricas (sección 7.2), UVa 1280, 10566, 10668, 11646, 12097.

<sup>5</sup>La lista no es exhaustiva ni, probablemente, lo será nunca.

En esta sección, incluimos dos ejemplos más del uso de la técnica de búsqueda binaria de la respuesta. Su combinación con otro algoritmo se puede detectar mediante la pregunta: “Si intentamos adivinar la respuesta solicitada (en forma de búsqueda binaria), ¿se convierte el problema original en una pregunta de verdadero/falso?”.

### Búsqueda binaria de la respuesta y algoritmo voraz

Enunciado resumido del problema UVa 714 - Copying Books: se nos dan  $m \leq 500$  libros numerados  $1, 2, \dots, m$ , que pueden tener un número distinto de páginas ( $p_1, p_2, \dots, p_m$ ). Quieres hacer una copia de cada uno. La tarea consiste en asignar esos libros a  $k$  escribas,  $k \leq m$ . Cada libro se le puede asignar a un solo escriba, y cada escriba debe recibir una *secuencia continua* de libros. Eso significa que existe una sucesión creciente de números  $0 = b_0 < b_1 < b_2, \dots < b_{k-1} \leq b_k = m$ , de forma que el escriba  $i$ -ésimo ( $i > 0$ ) reciba una secuencia de libros entre  $b_{i-1} + 1$  y  $b_i$ . Todos los escribas copian las páginas a la misma velocidad. Por lo tanto, el tiempo necesario para copiar todos los libros viene determinado por el escriba al que se le asigne más trabajo. Ahora hay que determinar: “¿cuál es el mínimo de páginas que copiará el escriba con más trabajo?”.

Existe una solución de programación dinámica para este problema, pero podemos resolverlo probando a adivinar la respuesta mediante búsqueda binaria. Utilicemos un ejemplo, en el que  $m = 9$ ,  $k = 3$  y  $p_1, p_2, \dots, p_9$  son 100, 200, 300, 400, 500, 600, 700, 800 y 900, respectivamente.

Si imaginamos que la *respuesta* = 1000, el problema se hace más ‘fácil’, es decir, si el escriba con más trabajo solo puede copiar hasta 1000 páginas, ¿se puede resolver el problema? La respuesta es que no. Podemos asignar de forma voraz los trabajos del libro 1 al  $m$  de la siguiente manera: {100, 200, 300, 400} para el escriba 1, {500} para el escriba 2 y {600} para el escriba 3. Pero, si lo hacemos así, seguiremos teniendo 3 libros {700, 800, 900} sin asignar. Por lo tanto, la respuesta debe ser  $> 1000$ .

Si probamos con *respuesta* = 2000, podemos asignar de forma voraz los trabajos de la siguiente manera: {100, 200, 300, 400, 500} para el escriba 1, {600, 700} para el escriba 2 y {800, 900} para el escriba 3. Se copiarán todos los libros, pero seguimos teniendo lagunas, porque los escribas 1, 2 y 3 tendrán un potencial no utilizado de {500, 700, 300}. Por lo tanto, la respuesta debe ser  $\leq 2000$ .

Podemos buscar esta *respuesta* de forma binaria entre  $[lo..hi]$ , donde  $lo = \max(p_i), \forall i \in [1..m]$  (el número de páginas del libro más grueso) y  $hi = p_1 + p_2 + \dots + p_m$  (la suma de las páginas de todos los libros). Y, para los curiosos, la *respuesta* óptima para este caso de prueba es 1700. La complejidad de tiempo de esta solución es de  $O(m \log hi)$ . Tengamos en cuenta que ese factor log adicional suele ser despreciable, en el ámbito de un concurso de programación.

### Búsqueda binaria de la respuesta y fórmulas geométricas

Utilizaremos el problema UVa 11646 - Athletics Track, para ilustrar otra posibilidad de la técnica de búsqueda binaria de la respuesta. El enunciado resumido del problema dice: examina un campo de fútbol rectangular con una pista de atletismo alrededor, como se ve en la parte izquierda de la figura 8.13, donde los dos arcos de ambos lados (arco1 y arco2) pertenecen al mismo círculo, centrado en el campo de fútbol. Queremos que la longitud de la pista de atletismo ( $L_1 + \text{arco1} + L_2 + \text{arco2}$ ) sea de, exactamente, 400 metros. Si recibimos la razón de la longitud  $L$  y la anchura  $W$  del campo de fútbol, en forma  $a : b$ , ¿cuáles deben ser la longitud  $L$  y la anchura  $W$  del campo de fútbol, que satisfaga los límites anteriores?

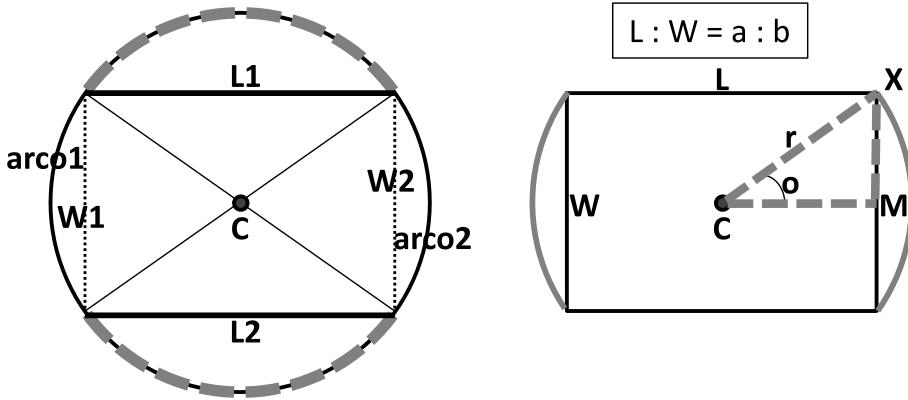


Figura 8.13: Pista de atletismo (de UVa 11646)

Es muy difícil, aunque no imposible, obtener la solución con papel y lápiz (solución analítica), pero con la ayuda de un ordenador y la búsqueda binaria de la respuesta (en realidad, el método de bisección), la respuesta es muy fácil.

Identificamos por búsqueda binaria el valor de  $L$ . Con  $L$  podemos obtener  $W = b/a \times L$ . La longitud esperada de un arco es  $(400 - 2 \times L)/2$ . Ahora, podemos utilizar trigonometría para calcular el radio  $r$  y el ángulo  $o$ , mediante el triángulo  $CMX$  (ver la parte derecha de la figura 8.13).  $CM = 0,5 \times L$  y  $MX = 0,5 \times W$ . Con  $r$  y  $o$ , podemos calcular la verdadera longitud del arco. Entonces comparamos este valor con la longitud esperada del arco, para decidir si tenemos que aumentar o reducir la longitud  $L$ . Mostramos el código relevante a continuación.

```

1 lo = 0.0; hi = 400.0; // rango posible de la respuesta
2 while (fabs(lo-hi) > 1e-9) {
3 L = (lo+hi) / 2.0; // método de bisección sobre L
4 W = b/a*L; // W se puede deducir de L y la razón a:b
5 expected_arc = (400 - 2.0*L) / 2.0; // valor de referencia
6
7 CM = 0.5*L; MX = 0.5*W; // aplicar aquí trigonometría
8 r = sqrt(CM*CM + MX*MX);
9 angle = 2.0 * atan(MX/CM) * 180.0/PI; // ángulo del arco = 2x ángulo o
10 this_arc = angle/360.0 * PI * (2.0*r); // calcular el valor del arco
11
12 if (this_arc > expected_arc) hi = L; else lo = L; // aumentar/reducir L
13 }
14 printf("Case %d: %.12lf %.12lf\n", caseNo++, L, W);

```

### Ejercicio 8.4.1.1\*

Demostrar que otras estrategias no serán mejores que la voraz mencionada, para la solución del problema UVa 714.

### Ejercicio 8.4.1.2\*

Deducir la solución analítica del problema UVa 11646, en vez de utilizar la técnica de búsqueda binaria de la respuesta.

#### 8.4.2 Dos componentes: con RSQ/RMQ estática unidimensional

Esta combinación debería ser bastante fácil de detectar. El problema implica *otro* algoritmo, para poblar el contenido de un *array estático* unidimensional (que no cambiará una vez completado) y sobre el que habrá *muchas* consultas de suma de rango/rango mínimo/rango máximo (RSQ/RMQ). La mayor parte de las veces, estas RSQ/RMQ se realizan en la fase de salida del problema. Pero, en ocasiones, se utilizan para acelerar el mecanismo interno de otros algoritmos.

En la sección 3.5.2 se ha tratado la solución para RSQ estática unidimensional con programación dinámica. Para la RMQ estática unidimensional, tenemos la estructura de datos de tabla de dispersión (que es una solución de DP), tratada en la sección 9.33. Sin esta aceleración de la DP de RSQ/RMQ, el otro algoritmo necesario para resolver el problema obtendrá, probablemente, un veredicto de TLE.

Como ejemplo sencillo, consideremos un problema simple, que pide saber cuántos primos hay en varios rangos de consulta  $[a..b]$  ( $2 \leq a \leq b \leq 1000000$ ). Este problema implica, obviamente, generación de números primos (por ejemplo, con el algoritmo de criba, sección 5.5.1). Pero, como el problema tiene  $2 \leq a \leq b \leq 1000000$ , recibiremos un veredicto de TLE si realizamos cada consulta en tiempo  $O(b - a + 1)$ , iterando desde  $a$  hasta  $b$ , especialmente si el autor ha decidido, a propósito, que  $b - a + 1$  sea cercano a 1000000 en (casi) cada consulta. Necesitamos acelerar la fase de salida hasta  $O(1)$  por consulta, utilizando una solución de programación dinámica con RSQ estática unidimensional.

#### 8.4.3 Dos componentes: procesamiento previo de un grafo y DP

En esta subsección, nos centramos en problemas en los que el procesamiento previo de un grafo es uno de los componentes, ya que implica grafos, y la DP es el otro. Mostramos esta combinación con dos ejemplos.

##### SSSP/APSP y TSP con DP

Utilizaremos el problema UVa 10937 - Blackbeard the Pirate, para ilustrar esta combinación de SSSP/APSP y TSP con DP. Los SSSP/APSP se utilizan para transformar la entrada (normalmente un grafo implícito o una rejilla) en otro grafo (generalmente más pequeño). Entonces, ejecutamos la solución de programación dinámica para TSP en el segundo grafo.

La entrada de este problema se muestra en la parte izquierda del siguiente diagrama. Se trata del ‘mapa’ de una isla. Barbanegra acaba de atracar en esta isla, en la posición indicada con una ‘@’. Ha enterrado hasta 10 tesoros en la isla. Los tesoros están etiquetados con el símbolo de exclamación ‘!’. Hay nativos furiosos indicados con un ‘\*’. Barbanegra debe mantenerse, al menos, a una casilla de distancia de los nativos furiosos, en cualquiera de las ocho direcciones. Pero Barbanegra quiere recuperar todos sus tesoros y volver al barco. Solo puede desplazarse por tierra, ‘,’ y no por agua ‘~’ o por zonas de obstáculos ‘#’.

| Entrada:<br>grafo implícito | Índices @ y !<br>Aumentar * con X | Matriz de distancias APSP de<br>un grafo completo (pequeño)                                                                                                                                                                                                                 |
|-----------------------------|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ~~~~~                       | ~~~~~                             |                                                                                                                                                                                                                                                                             |
| ~~!!!###~~                  | ~~123###~~                        |                                                                                                                                                                                                                                                                             |
| ~##...###~                  | ~##..X###~                        | $\begin{array}{ c cccccc } \hline & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 11 & 10 & 11 & 8 & 8 \\ 1 & 11 & 0 & 1 & 2 & 5 & 9 \\ 2 & 10 & 1 & 0 & 1 & 4 & 8 \\ 3 & 11 & 2 & 1 & 0 & 5 & 9 \\ 4 & 8 & 5 & 4 & 5 & 0 & 6 \\ 5 & 8 & 9 & 8 & 9 & 6 & 0 \\ \hline \end{array}$ |
| ~#. . *#~                   | ~#. . XX*#~                       |                                                                                                                                                                                                                                                                             |
| ~#! . *~~~                  | ~#4. X**~~~                       |                                                                                                                                                                                                                                                                             |
| ~~. . . ~~~                 | ~~. . XX~~~                       |                                                                                                                                                                                                                                                                             |
| ~~. . . ~~~                 | ~~. . . ~~~                       |                                                                                                                                                                                                                                                                             |
| ~~. . . @~~                 | ~~. . . 0~~                       |                                                                                                                                                                                                                                                                             |
| ~#! . ~~~~                  | ~#5. ~~~~                         |                                                                                                                                                                                                                                                                             |
| ~~~~~                       | ~~~~~                             |                                                                                                                                                                                                                                                                             |

Estamos ante un problema TSP evidente (sección 3.5.3), pero antes de que podamos utilizar la solución de TSP con DP, tenemos que hacer del grafo de entrada una matriz de distancias.

En este problema, solo nos interesan los ‘@’ y los ‘!’. Asignamos el índice 0 a ‘@’ e índices positivos a los otros ‘!’. Aumentamos el alcance de cada ‘\*’, sustituyendo el ‘\*’ alrededor del ‘\*’ con una ‘X’. Ejecutamos BFS en el grafo no ponderado implícito, empezando por ‘@’ y todos los ‘!’, accediendo solo a las celdas etiquetadas como ‘?’ (tierra). Esto dará una matriz de distancias de los caminos más cortos entre todos los pares (APSP), como vemos en el diagrama.

Ahora, una vez que tenemos la matriz de distancias APSP, podemos ejecutar TSP con DP, como vimos en la sección 3.5.3, para obtener la respuesta. Para el caso de prueba anterior, la ruta TSP óptima es: 0-5-4-1-2-3-0, con un coste =  $8+6+5+1+1+11 = 32$ .

#### Contracción de SCC y algoritmo de DP en un DAG

En algunos problemas modernos, que implican grafos *dirigidos*, hemos tenido que tratar con los componentes fuertemente conexos (SCC) del grafo (sección 4.2.9). Una de las variantes más recientes es el problema que pide *contraer* primero todos los SCC del grafo dirigido dado, para formar vértices más grandes (que llamaremos supervértices). No se garantiza que el grafo dirigido original sea acíclico, por lo que no podemos aplicar inmediatamente técnicas de DP. Pero cuando se contraen los SCC de un grafo dirigido, el grafo de supervértices resultante es un DAG (ver un ejemplo en la figura 4.9). Si recuerdas el contenido de la sección 4.7.1, un DAG resulta muy apropiado para las técnicas de DP, porque siempre es acíclico.

UVa 11324 - The Largest Clique es un de esos problemas. En resumen, trata sobre encontrar el camino más largo en el DAG de SCC contraídos. Cada supervértice tiene un peso, que representa el número de vértices que se han contraído en él.

#### 8.4.4 Dos componentes: con grafos

Este tipo de combinaciones de problemas se detecta porque uno de los componentes es, claramente, un algoritmo de grafos. Sin embargo, necesitamos otro algoritmo de apoyo que, normalmente, será alguna regla matemática o geométrica (para construir el grafo subyacente) o, incluso, otro algoritmo de grafos. En esta subsección, ilustramos uno de esos ejemplos.

Hemos mencionado, en la sección 2.4.1, que, en algunos problemas, no es necesario almacenar el grafo subyacente en ninguna estructura de datos específica de grafos (grafo implícito). Esto

es posible si podemos deducir con facilidad las aristas del grafo, o mediante algunas reglas. UVa 11730 - Number Transformation es uno de esos problemas.

Aunque el enunciado es eminentemente matemático, la cuestión principal es, en realidad, un problema de caminos más cortos de origen único (SSSP) en un grafo no ponderado, que se resuelve mediante BFS. El grafo subyacente se genera al vuelo durante la ejecución de la BFS. El origen es el número  $S$ . A partir de ahí, cada vez que la BFS procesa un vértice  $d$ , añade a una cola el vértice no visitado  $u + x$ , donde  $x$  es un factor primo de  $u$ , que no sea ni 1 ni el propio  $u$ . El número de capa de la BFS, cuando se alcanza el vértice de destino  $T$ , es el número mínimo de transformaciones necesarias para convertir  $S$  en  $T$ , según las reglas del problema.

#### 8.4.5 Dos componentes: con matemáticas

En esta combinación, uno de los componentes es, claramente, un problema matemático, pero no es el único. El otro no será, normalmente, un grafo, ya que, en ese caso, estaría clasificado en el punto anterior. Lo más probable es que sea *backtracking* recursivo o búsqueda binaria. También es posible tener dos algoritmos matemáticos diferentes en el mismo problema. A continuación, ilustramos un ejemplo de ello.

UVa 10637 - Coprimes, es el problema de partición de  $S$  ( $0 < S \leq 100$ ) en  $t$  ( $0 < t \leq 30$ ) números coprimos. Por ejemplo, para  $S = 8$  y  $t = 3$ , podemos tener  $1 + 1 + 6$ ,  $1 + 2 + 5$  o  $1 + 3 + 4$ . Después de leer el enunciado del problema, tendremos una fuerte sensación de que estamos ante un problema de matemáticas (teoría de números). Sin embargo, necesitaremos algo más que la criba de Eratóstenes para generar los primos, el GCD para comprobar si dos números son coprimos y una rutina de *backtracking* recursivo para crear todas las particiones.

#### 8.4.6 Dos componentes: búsqueda completa y geometría

Muchos problemas de geometría (computacional) se pueden resolver por fuerza bruta (aunque algunos necesitan una solución de divide y vencerás). Cuando los límites de la entrada permitan una solución de búsqueda completa, no dudes y hazlo así.

Por ejemplo, el problema UVa 11227 - The silver bullet, se reduce a lo siguiente: dados  $N$  ( $1 \leq N \leq 100$ ) puntos en un plano bidimensional, determinar el número máximo de puntos que son colineales. Podemos permitirnos utilizar la siguiente solución de búsqueda completa en  $O(N^3)$ , ya que  $N \leq 100$  (aunque hay una solución mejor). Para cada par de puntos  $i$  y  $j$ , comprobamos si los otros  $N-2$  puntos son colineales con la línea  $i - j$ . Esta solución es muy fácil, con tres bucles anidados y la función `bool collinear(point p, point q, point r)`, que aparece en la sección 7.2.2.

#### 8.4.7 Dos componentes: con estructura de datos eficiente

Esta combinación suele aparecer en algunos problemas ‘estándar’, pero con límites de entrada *grandes*, de forma que debemos utilizar una estructura de datos más eficiente, para evitar un veredicto de TLE.

Por ejemplo, el problema UVa 11967-Hic-Hac-Hoe, es una extensión del juego de las tres en raya. En vez de utilizar el pequeño tablero de  $3 \times 3$ , nos encontramos ante un tablero ‘infinito’. Por lo tanto, no hay forma de almacenarlo en un *array* bidimensional. Por suerte, podemos

almacenar las coordenadas de los ‘círculos’ y las ‘cruces’ en un BST equilibrado y referirnos a él para comprobar el estado de la partida.

### 8.4.8 Tres componentes

En las subsecciones 8.4.1-8.4.7, hemos visto varios ejemplos de problemas que implicaban dos componentes. En esta subsección, mostraremos dos ejemplos de combinaciones poco habituales de tres algoritmos y/o estructuras de datos diferentes.

#### Factores primos, DP y búsqueda binaria

El problema UVa 10856 - Recover Factorial se puede resumir de la siguiente manera: “dado  $N$ , el número de factores primos de  $X!$ , ¿cuál es el valor mínimo posible de  $X$ ? ( $N \leq 100000001$ )”. Resolver este problema es todo un reto. Para hacerlo posible, empezamos por descomponerlo.

En primer lugar, calculamos el número de factores primos de un entero  $i$ , y lo almacenamos en una tabla `NumPF[i]`, con la siguiente recurrencia: si  $i$  es primo, entonces `NumPF[i] = 1` factor primo; pero si  $i = PF \times i'$ , entonces `NumPF[i] = 1 + el número de factores primos de i'`. Calculamos este número de factores primos  $\forall i \in [1..2703665]$ . El límite superior de este rango lo obtenemos por ensayo-error, de acuerdo a los límites establecidos en el enunciado.

La segunda parte de la solución consiste en *acumular* el número de factores primos de  $N!$ , estableciendo `NumPF[i] += NumPF[i-1];  $\forall i \in [1..N]$` . Por lo tanto, `NumPF[N]` contiene el número de factores primos de  $N!$ . Esta es la solución de DP para el problema de la RSQ estática unidimensional.

Por último, la tercera parte de la solución debería ser obvia: podemos hacer una búsqueda binaria para encontrar el índice  $X$ , de forma que `NumPF[X] = N`. Si no hay respuesta, la salida indicará “Not possible.”.

#### Búsqueda completa, búsqueda binaria y voraz

A continuación, veremos un problema de la final mundial del ICPC que combina *tres* paradigmas de resolución de problemas, que aprendimos en el capítulo 3. Concretamente, búsqueda completa, divide y vencerás (búsqueda binaria) y voraz.

#### Final mundial ACM ICPC 2009 - Problema A - A Careful Approach, LA 4445

Enunciado resumido del problema: estamos ante una situación de aterrizaje de aviones, en la que hay  $2 \leq n \leq 8$  aviones. Cada uno tiene una ventana de tiempo en la que puede aterrizar con seguridad. Esta ventana viene determinada por dos enteros,  $a_i$  y  $b_i$ , que indican el principio y el final de un intervalo cerrado  $[a_i..b_i]$ , en el que debe aterrizar el avión  $i$ -ésimo. Los números  $a_i$  y  $b_i$  indican minutos y satisfacen  $0 \leq a_i \leq b_i \leq 1440$  (24 horas). En este problema puedes asumir que el tiempo que emplea el avión en aterrizar es cero. Las tareas son:

1. Calcular un **orden para el aterrizaje de todos los aviones**, que respete las ventanas de tiempo. Pista: orden = permutación = búsqueda completa?

- Además, los aterrizajes deben estar separados entre sí **todo lo que se pueda**, de forma que el espacio de tiempo transcurrido entre aterrizajes sucesivos sea lo más grande posible. Por ejemplo, si aterrizarán tres aviones a las 10:00, 10:05 y 10:15, el menor periodo entre aterrizajes es de 5 minutos, lo que sucede entre los dos primeros aviones. Los períodos no deben ser iguales, pero el menor debe ser lo más largo posible. Pista: ¿tenemos una situación similar al del problema de ‘cobertura de intervalos’ (sección 3.4.1)?
- Muestra la respuesta en minutos y segundos, redondeando al segundo más cercano.

Puedes ver una ilustración en la figura 8.14, donde las líneas indican la ventana de tiempo de un avión y las estrellas su momento óptimo de aterrizaje.

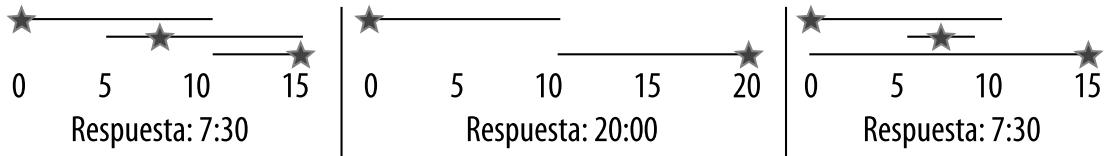


Figura 8.14: Ilustración de ACM ICPC WF2009 - A - A Careful Approach

Solución: como habrá un máximo de 8 aviones, se puede encontrar una solución óptima, probando los  $8! = 40320$  órdenes de aterrizajes posibles. Este es el componente de **búsqueda completa** del problema, que se puede implementar con facilidad utilizando `next_permutation` de `algorithm`, en la STL de C++.

Para cada orden de aterrizaje específico, queremos conocer la ventana de aterrizaje más larga posible. Supongamos que probamos a adivinar que la respuesta es una longitud  $L$  determinada. Podemos verificar vorazmente si esa  $L$  es viable, obligando al primer avión a aterrizar lo antes posible y al resto en `max(a[ese avión], hora de aterrizaje anterior + L)`. Este es el componente **voraz**.

Una ventana de longitud  $L$  demasiado larga/corta, provocará que `lastLanding` (ver el código a continuación) se quede largo/corto para `b[último avión]`, por lo que habrá que reducir o aumentar  $L$ . Podemos buscar la respuesta de forma binaria para  $L$ . Este es el componente de **divide y vencerás** de este problema. Como solo necesitamos saber la respuesta redondeada al entero más cercano, podemos detener la búsqueda binaria cuando el error  $\epsilon < 1e-3$ . Para más detalles, estudia el código fuente.

```

1 // Final mundial Estocolmo 2009, A - A Careful Approach, UVa 1079, LA 4445
2
3 #include <algorithm>
4 #include <cmath>
5 #include <cstdio>
6 using namespace std;
7
8 int i, n, caseNo = 1, order[8];
9 double a[8], b[8], L, maxL;
10
11 double greedyLanding() { // con un orden de aterrizajes y una L, intentar
12 // aterrizar los aviones y ver cuál es el espacio hasta b[order[n - 1]]}

```

```

13 double lastLanding = a[order[0]]; //voraz, el primero aterriza al instante
14 for (i = 1; i < n; i++) { // para los otros aviones
15 double targetLandingTime = lastLanding+L;
16 if (targetLandingTime <= b[order[i]])
17 // aterriza: elección voraz del máximo a[order[i]] o targetLandingTime
18 lastLanding = max(a[order[i]], targetLandingTime);
19 else
20 return 1;
21 }
22 // devolver valor positivo para reducir L en la búsqueda binaria
23 // devolver valor negativo para aumentar L en la búsqueda binaria
24 return lastLanding - b[order[n-1]];
25 }

26
27 int main() {
28 while (scanf("%d", &n), n) // 2 <= n <= 8
29 for (i = 0; i < n; i++) { // avión i aterriza en el intervalo [ai, bi]
30 scanf("%lf %lf", &a[i], &b[i]);
31 a[i] *= 60; b[i] *= 60; //originalmente minutos, convertir a segundos
32 order[i] = i;
33 }
34
35 maxL = -1.0; // variable que buscamos
36 do { // permutar orden de aterrizaje, hasta 8!
37 double lo = 0, hi = 86400; // mínimo 0 seg, máximo 1 día = 86400 seg
38 L = -1; // comenzar con una solución inviable
39 while (fabs(lo-hi) >= 1e-3) { // búsqueda binaria de L, EPS = 1e-3
40 L = (lo+hi) / 2.0; // respuesta redondeada al entero más cercano
41 double retVal = greedyLanding(); // primero redondear
42 if (retVal <= 1e-2) lo = L; // debe aumentar L
43 else hi = L; // inviable, debe reducir L
44 }
45 maxL = max(maxL, L); // obtener el máximo de todas las permutaciones
46 }
47 while (next_permutation(order, order+n)); // todas las permutaciones
48
49 // otra forma de redondear es usar el formato de printf: %.0lf:%.2lf
50 maxL = (int)(maxL + 0.5); // redondear al segundo más cercano
51 printf("Case %d: %d:%.2d\n", caseNo++, (int)(maxL/60), (int)maxL%60);
52 }

53
54 return 0;
55 }

```



ch8\_04\_UVa1079.cpp



ch8\_04\_UVa1079.java

## Ejercicio 8.4.8.1

El código anterior será aceptado, pero utiliza el tipo de datos double para `lo`, `hi` y `L`. En realidad, esto no es necesario, ya que se pueden realizar todos los cálculos con enteros. Además, en vez de usar `while (fabs(lo-hi) >= 1e-3)`, usa `for (int i = 0; i < 50; i++)`. Reescribe el código.

## Ejercicios de programación

### Ejercicios de programación relativos a descomposición de problemas:

#### Dos componentes - Búsqueda binaria de la respuesta y otro

1. UVa 00714 - Copying Books (búsqueda binaria de la respuesta y voraz)
2. UVa 01221 - Against Mammoths (LA 3795 - Tehran06; búsqueda binaria de la respuesta y MCBM (emparejamiento perfecto); usar el algoritmo de aumento de camino para calcular el MCBM, ver sección 4.7.4)
3. UVa 01280 - Curvy Little Bottles (LA 6027 - WorldFinals Warsaw12; búsqueda binaria de la respuesta y fórmula geométrica)
4. UVa 10372 - Leaps Tall Buildings ... (búsqueda binaria de la respuesta y física)
5. UVa 10566 - Crossed Ladders (método de bisección)
6. UVa 10606 - Opening Doors (la solución es el mayor número cuadrado  $\leq N$ , pero este problema implica BigInteger; podemos utilizar una técnica de búsqueda binaria de la respuesta (bastante lenta) para obtener  $\sqrt{N}$ )
7. UVa 10668 - Expanding Rods (método de bisección)
8. UVa 10804 - Gopher Strategy (similar a UVa 11262)
9. UVa 10816 - Travel in Desert (búsqueda binaria de la respuesta y Dijkstra)
10. **UVa 10983 - Buy one, get ... \*** (búsqueda binaria de la respuesta y flujo máximo)
11. **UVa 11262 - Weird Fence \*** (búsqueda binaria de la respuesta y MCBM)
12. **UVa 11516 - WiFi \*** (búsqueda binaria de la respuesta y voraz)
13. UVa 11646 - Athletics Track (el círculo está en el centro de la pista)
14. UVa 12428 - Enemy at the Gates (búsqueda binaria de la respuesta y un poco de teoría de grafos sobre puentes, vista en el capítulo 4)
15. IOI 2009 - Mecho (búsqueda binaria de la respuesta y BFS)

#### Dos componentes - DP y RSQ/RMQ unidimensional

1. UVa 00967 - Circular (similar a UVa 897; pero esta vez la salida se puede acelerar utilizando suma de rangos unidimensional con DP)
2. UVa 10200 - Prime Time (búsqueda completa; comprobar si  $\text{isPrime}(n^2 + n + 41) \forall n \in [a \dots b]$ ; esta fórmula  $n^2 + n + 41$  de generación de primos fue descubierta por Leonhard Euler; funciona para  $0 \leq n \leq 40$ ; sin embargo, es poco precisa para  $n$  mayores; por último, usar DP con RSQ unidimensional para acelerar la solución)
3. UVa 10533 - Digit Primes (criba; comprobar si un primo es un dígito primo; suma de rangos unidimensional con DP)

- |                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>4. UVa 10871 - Primed Subsequence</p> <p>5. <b><u>UVa 10891 - Game of Sum</u></b>*</p> <p>6. <b><u>UVa 11105 - Semi-prime H-numbers</u></b>*</p> <p>7. <b><u>UVa 11408 - Count DePrimes</u></b>*</p> <p>8. UVa 11491 - Erasing and Winning</p> <p>9. UVa 12028 - A Gift from ...</p> | <p>(consulta de suma de rangos unidimensional)</p> <p>(DP doble; la primera DP es la consulta de suma de rangos 1D estándar entre dos índices: <math>i, j</math>; la segunda DP evalúa el árbol de decisión con estado <math>(i, j)</math> y prueba todos los puntos de división; <i>minimax</i>)</p> <p>(consulta de suma de rangos unidimensional)</p> <p>(consulta de suma de rangos unidimensional)</p> <p>(voraz; optimizado para estructura de datos de tabla dispersa, para manejar la RMQ estática)</p> <p>(generar el <i>array</i>; ordenarlo; preparar consulta de suma de rangos unidimensional; entonces la solución es mucho más sencilla)</p> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Dos componentes - Procesamiento previo de grafos y DP

- |                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>1. <b><u>UVa 00976 - Bridge Building</u></b>*</p> <p>2. UVa 10917 - A Walk Through the Forest</p> <p>3. UVa 10937 - Blackbeard the Pirate</p> <p>4. UVa 10944 - Nuts for nuts..</p> <p>5. <b><u>UVa 11324 - The Largest Clique</u></b>*</p> <p>6. <b><u>UVa 11405 - Can U Win?</u></b>*</p> <p>7. UVa 11693 - Speedy Escape</p> <p>8. UVa 11813 - Shopping</p> | <p>(utiliza un sistema de relleno por difusión para separar las orillas norte y sur; usarlo para calcular el coste de instalar un puente en cada columna; la solución de DP debería resultar evidente tras este procesamiento previo)</p> <p>(conteo de caminos en un DAG; pero, primero, hay que construir el DAG ejecutando el algoritmo de Dijkstra desde 'casa')</p> <p>(BFS → TSP; después DP o <i>backtracking</i>; tratado en esta sección)</p> <p>(BFS → TSP; después DP; <math>n \leq 16</math>)</p> <p>(caminos más largos en un DAG; pero, primero, hay que transformar el grafo al DAG de sus SCC; ordenación topológica)</p> <p>(BFS desde 'K' y cada 'P', 9 elementos como máximo; después DP-TSP)</p> <p>(calcular información de caminos más cortos usando Floyd Warshall; después, DP)</p> <p>(Dijkstra → TSP; después, DP, <math>n \leq 10</math>)</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Dos componentes - Con grafos

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>1. UVa 00273 - Jack Straw</p> <p>2. UVa 00521 - Gossiping</p> <p>3. UVa 01039 - Simplified GSM Network</p> <p>4. <b><u>UVa 01092 - Tracking Bio-bots</u></b>*</p> <p>5. UVa 01243 - Polynomial-time Red...</p> <p>6. UVa 01263 - Mines</p> <p>7. UVa 10075 - Airlines</p> <p>8. UVa 10307 - Killing Aliens in Borg Maze</p> <p>9. UVa 11267 - The 'Hire-a-Coder' ...</p> <p>10. <b><u>UVa 11635 - Hotel Booking</u></b>*</p> <p>11. UVa 11721 - Instant View ...</p> | <p>(intersección de segmentos y algoritmo de clausura transitiva de Warshall)</p> <p>(construir un grafo; los vértices son conductores; asignar un arista entre dos conductores si pueden encontrarse; se determina mediante una regla matemática (GCD); si el grafo es conexo, la respuesta es 'sí')</p> <p>(LA 3270 - WorldFinals Shanghai05; construir el grafo con geometría simple; después usar Floyd Warshall)</p> <p>(LA 4787 - WorldFinals Harbin10; primero comprimir el grafo; recorrerlo desde la salida utilizando solo movimientos al sur y al oeste; inclusión-exclusión)</p> <p>(LA 4272 - Hefei08; clausura transitiva de Warshall; SCC; reducción transitiva de un grafo dirigido)</p> <p>(LA 4846 - Daejeon10; geometría; SCC; dos problemas relacionados: UVa 11504 y 11770)</p> <p>((<i>gcDistance</i>), sección 9.11, con APSP)</p> <p>(construir grafo SSSP con BFS; MST)</p> <p>(comprobación bipartita; MST; aceptar peso negativo)</p> <p>(Dijkstra y BFS)</p> <p>(buscar nodos que puedan llegar a SCC con ciclo negativo)</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

12. UVa 11730 - Number Transformation
  13. UVa 12070 - Invite Your Friends
  14. UVa 12101 - Prime Path
  - 15. UVa 12159 - Gun Fight \***
- (factorización de primos; ver sección 5.5.1)  
 (LA 3290 - Dhaka05; BFS y Dijkstra)  
 (BFS; implica números primos)  
 (LA 4407 - KualaLumpur08; usar comprobaciones CCW sencillas  
 (geometría), para construir el grafo bipartito; MCBM)

## Dos componentes - Con matemáticas

1. UVa 01195 - Calling Extraterrestrial ...
  2. UVa 10325 - The Lottery
  3. UVa 10427 - Naughty Sleepy ...
  - 4. UVa 10539 - Almost Prime Numbers \***
  5. UVa 10637 - Coprimes \*
  6. UVa 10717 - Mint \*
  7. UVa 11282 - Mixing Invitations
  8. UVa 11415 - Count the Factorials
  9. UVa 11428 - Cubes
- (LA 2565 - Kanazawa02; usar criba para generar la lista de primos;  
 fuerza bruta sobre cada primo  $p$ ; y búsqueda binaria para encontrar la  
 pareja  $q$  correspondiente)  
 (principio de inclusión-exclusión; fuerza bruta en los subconjuntos  
 para  $M \leq 15$  pequeños; LCM-GCD)  
 (los números en  $[10^{(k-1)} \dots 10^k - 1]$  tienen  $k$  dígitos)  
 (criba; obtener 'casi primos': números no primos que son divisibles por  
 un único número primo; podemos obtener una lista de 'casi primos'  
 mediante las potencias de los primos, por ejemplo, 3 es primo, por lo  
 que  $3^2 = 9, 3^3 = 27, 3^4 = 81$ , etc., son 'casi primos'; después  
 podemos ordenarlos y realizar búsqueda binaria)  
 (números primos y GCD)  
 (búsqueda completa y GCD/LCM, sección 5.5.2)  
 (desarreglo y coeficiente binomial; usar BigInteger de Java)  
 (contar el número de factores de cada entero; hallar el número de  
 factores de cada número factorial, y almacenarlo en un array; para  
 cada consulta, buscar en el array para encontrar el primer elemento  
 con ese valor, mediante búsqueda binaria)  
 (búsqueda completa y búsqueda binaria)

## Dos componentes - Búsqueda completa y geometría

1. UVa 00142 - Mouse Clicks
  2. UVa 00184 - Laser Lines
  3. UVa 00201 - Square
  4. UVa 00270 - Lining Up
  5. UVa 00356 - Square Pegs And Round ...
  6. UVa 00638 - Finding Rectangles
  7. UVa 00688 - Mobile Phone Coverage
  8. UVa 10012 - How Big Is It? \*
  9. UVa 10167 - Birthday Cake
  10. UVa 10301 - Rings and Glue
  11. UVa 10310 - Dog and Gopher
  12. UVa 10823 - Of Circles and Squares
  13. UVa 11227 - The silver bullet \*
  14. UVa 11515 - Cranes
- (fuerza bruta; punto en un rectángulo; dist)  
 (fuerza bruta; comprobación collinear)  
 (conteo de cuadrados de varios tamaños; probar todos)  
 (ordenación de gradienes; búsqueda completa)  
 (distancia euclídea; fuerza bruta)  
 (obtener las 4 esquinas por fuerza bruta)  
 (fuerza bruta; dividir la región en pequeños rectángulos y decidir si cada  
 uno está cubierto, o no, por una antena; si lo está, añadir el área de ese  
 rectángulo a la respuesta)  
 (probar las 8! permutaciones; dist euclídea)  
 (fuerza bruta para  $A$  y  $B$ ; comprobaciones ccw)  
 (intersección de círculos; backtracking)  
 (búsqueda completa; distancia euclídea)  
 (búsqueda completa; comprobar si un punto está dentro de  
 círculos/cuadrados)  
 (fuerza bruta; comprobación collinear)  
 (intersección de círculos; backtracking)

## 15. UVa 11574 - Colliding Traffic \*

(obtener todos los pares de botes por fuerza bruta; si un par ya ha colisionado, la respuesta es 0,0; en caso contrario deducir la ecuación de segundo grado para detectar cuándo colisionarán, si es que lo harán; elegir el tiempo mínimo global de colisión; si no hay ninguna colisión, mostrar 'No collision.')

### Mezclados con estructuras de datos eficientes

#### 1. UVa 00843 - Crypt Kicker

(*backtracking*; probar a asignar cada letra a otra del alfabeto; usar estructura de datos de *trie* (sección 6.6) para acelerar, si una cierta palabra (parcial) está en el diccionario)

#### 2. UVa 00922 - Rectangle by the Ocean

(en primer lugar, calcular el área del polígono; por cada par de puntos, definir un rectángulo con ellos; usar *set* para comprobar si un tercer punto del rectángulo está en el polígono; comprobar si es mejor que el mejor hasta ese momento)

#### 3. UVa 10734 - Triangle Partitioning

(en realidad es un problema de geometría que implica la regla del triángulo/coseno, pero usamos una estructura de datos que tolere la imprecisión de coma flotante, debido a la normalización de los lados del triángulo, para asegurarnos de que contamos cada uno una sola vez)

#### 4. UVa 11474 - Dying Tree \*

(usar búsqueda de unión; empezar conectando todas las ramas del árbol; después, conectar un árbol a otro, si cualquiera de sus ramas tiene una distancia inferior a  $k$  (un poco de geometría); entonces, conectar cualquier árbol que pueda alcanzar a cualquier doctor; por último, verificar si la primera rama del primer árbol enfermo está conectada a algún médico; el código puede ser muy largo; cuidado)

#### 5. UVa 11525 - Permutation \*

(usar un árbol de Fenwick y búsqueda binaria de la respuesta para hallar el menor índice  $i$  que tenga  $RSQ(1, i) = Si$ )

#### 6. UVa 11960 - Divisor Game \*

(criba modificada, número de divisores; consulta de rango máximo estática; usar estructura de datos de árbol de dispersión)

#### 7. UVa 11966 - Galactic Bonding

(usar búsqueda de unión para llevar la cuenta del número de conjuntos disjuntos/constelaciones; si la distancia euclídea  $\leq D$ , unir las 2 estrellas) (fuerza bruta sencilla, pero utilizando *map* de la STL de C++, ya que no podemos almacenar el verdadero tablero de las tres en raya; solo recordamos  $n$  coordenadas y comprobamos si hay  $k$  coordenadas consecutivas que pertenezcan a algún jugador)

#### 8. UVa 11967 - Hic-Hac-Hoe

(fuerza bruta con estructura de datos *set*)

#### 9. UVa 12318 - Digital Roulette

(problema de BFS, pero necesita una estructura *set* de información de cadenas para acelerar el proceso)

### Tres componentes

#### 1. UVa 00295 - Batman \*

(búsqueda binaria de la respuesta  $x$  a esta pregunta: ¿si la persona tiene diámetro  $x$ , puede ir de izquierda a derecha?; por cualquier par de obstáculos (incluyendo los muros superior e inferior), trazar una arista entre ellos, si la persona no puede moverse entre los mismos y comprobar si los muros superior e inferior están desconectados → la persona con diámetro  $x$  puede pasar; distancia euclídea)

|                                                     |                                                                                                                                                                                                                           |
|-----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2. UVa 00811 - The Fortified Forest                 | (LA 5211 - WorldFinals Eindhoven99; CH, perímetro del polígono; generar iterativamente todos los subconjuntos con máscara de bits)                                                                                        |
| 3. <b><u>UVa 01040 - The Traveling Judges *</u></b> | (LA 3271 - WorldFinals Shanghai05; búsqueda completa iterativa; probar todos los subconjuntos de $2^{20}$ ciudades; crear un MST de esas ciudades con la ayuda de conjuntos disjuntos para unión-buscar; salida compleja) |
| 4. UVa 01079 - A Careful Approach                   | (LA 4445 - WorldFinals Stockholm09; tratado en este capítulo)                                                                                                                                                             |
| 5. UVa 01093 - Castles                              | (LA 4788 - WorldFinals Harbin10; probar todas las raíces posibles; DP en un árbol)                                                                                                                                        |
| 6. UVa 01250 - Robot Challenge                      | (LA 4607 - SoutheastUSA09; geometría; SSSP en un DAG → DP; suma de rangos unidimensional con DP)                                                                                                                          |
| 7. UVa 10856 - Recover Factorial                    | (tratado en esta sección)                                                                                                                                                                                                 |
| 8. UVa 10876 - Factory Robot                        | (búsqueda binaria de la respuesta y geometría; distancia euclídea y búsqueda de unión; idea similar a UVa 295)                                                                                                            |
| 9. <b><u>UVa 11610 - Reverse Prime *</u></b>        | (primero, invertir los primos inferiores a $10^6$ ; después, añadir ceros, si es necesario; usar árbol de Fenwick y búsqueda binaria)                                                                                     |

## 8.5 Soluciones a los ejercicios no resaltados

**Ejercicio 8.2.3.1:** en C++ podemos utilizar `pair<int, int>` (forma abreviada: `ii`), para almacenar un par de datos con enteros. Para una 3-tupla, podemos utilizar `pair<int, ii>`. Para una 4-tupla, `pair<ii, ii>`. En Java no tenemos una característica similar, por lo que tendremos que crear una clase que implemente `Comparable` (para que podamos utilizar `TreeMap` para almacenar los objetos adecuadamente).

**Ejercicio 8.2.3.2:** no tenemos otra opción que utilizar una clase, tanto en C++ como en Java. En C/C++, `struct` también es una opción. Después, tendremos que implementar una función de comparación para esa clase.

**Ejercicio 8.2.3.3:** la búsqueda estado-espacio es, en esencia, una extensión del problema de los caminos *más cortos* de origen único, que es un problema de minimización. El problema del camino más largo (maximización) es NP-complejo y, normalmente, no lo utilizamos, ya que (el problema de minimización de) la búsqueda estado-espacio es lo suficientemente complicada.

**Ejercicio 8.3.1.1:** la solución es similar a la de UVa 10911, vista en la sección 1.2. Pero en el problema del “emparejamiento de cardinalidad máxima”, existe la posibilidad de que un vértice *no sea* emparejado. A continuación, incluimos la solución de DP con máscara de bits para un grafo general pequeño:

```

1 int MCM(int bitmask) {
2 if (bitmask == (1<<N) - 1) return 0; // no más emparejamientos posibles
3 if (memo[bitmask] != -1) return memo[bitmask];
4
5 int p1, p2;
6 for (p1 = 0; p1 < N; p1++) // buscar un vértice libre p1
7 if (!(bitmask & (1<<p1)))
8 break;

```

```

9
10 // Esta es la diferencia clave:
11 // tenemos la posibilidad de no emparejar el vértice libre p1 con nada
12 int ans = MCM(bitmask | (1<<p1));
13
14 // Asumimos que el grafo pequeño se almacena en una matriz de adyacencia
15 for (p2 = 0; p2 < N; p2++) // buscar vecinos de p1 que estén libres
16 if (AdjMat[p1][p2] && p2 != p1 && !(bitmask & (1<<p2)))
17 ans = max(ans, 1 + MCM(bitmask | (1<<p1) | (1<<p2)));
18
19 return memo[bitmask] = ans;
20 }

```

**Ejercicio 8.4.8.1:** encontrarás la solución en la sección 3.3.1.

## 8.6 Notas del capítulo

Tal y como prometimos en las notas del capítulo de la edición anterior, hemos mejorado significativamente este capítulo 8. En su presente forma, ha duplicado, aproximadamente, su contenido y ejercicios, debido a dos razones. En primer lugar, hemos resuelto un buen número de problemas difíciles entre ambas ediciones. En segundo, hemos traído algunos de los problemas más difíciles de otros capítulos (en la edición anterior) a este, especialmente del capítulo 7 a la sección 8.4.6.

En esta edición, el capítulo 8 ya no es el último. Hemos añadido un capítulo 9, en el que incluimos una lista de temas poco habituales, que casi nunca aparecen en los concursos, pero que pueden ser de interés para el lector entusiasta.

# Capítulo 9

---

## Temas poco habituales

*El conocimiento es un tesoro que sigue a su dueño a todas partes.*

— Proverbio chino

### Introducción y motivación

En este capítulo, enumeraremos temas ‘exóticos’, o poco habituales, dentro de las ciencias de la computación, que podrían (aunque raramente lo hacen) aparecer en un concurso de programación. Estos problemas, estructuras de datos y algoritmos, son un poco distintos de los que hemos tratados en los ocho primeros capítulos, más generales. Aprender la materia de este capítulo se podría considerar como poco eficaz pues, después de invertir un gran esfuerzo en su estudio, lo más probable es que *no esté* presente en ningún concurso. Sin embargo, en nuestra opinión, estos temas poco habituales, resultarán atractivos a quienes, sinceramente, quieren ampliar su conocimiento sobre las ciencias de la computación.

Ignorar este capítulo no causará grandes daños de cara a la preparación para un concurso del tipo del ICPC, ya que la probabilidad de que aparezca cualquiera de estos temas es bastante baja<sup>1</sup>. Sin embargo, cuando los encontramos, los concursantes con un conocimiento previo, tendrán una ventaja evidente sobre los que no lo tienen. Los buenos programadores podrán deducir la solución a partir de conceptos básicos, pero será un proceso más lento que el de aquellos que ya han tenido contacto con los planteamientos y las soluciones.

Muchos de estos temas poco habituales no están incluidos en el temario de la IOI [20]. Por lo tanto, los concursantes de la IOI podrían aplazar el aprendizaje de este capítulo hasta su ingreso en la universidad. Sin embargo, una lectura rápida siempre es una buena idea.

Hemos tratado de mantener el tratamiento de cada tema en la forma más concisa posible, dedicándole una o dos páginas. No se incluye mucho código de ejemplo, ya que el lector que domine el contenido de los capítulos 1 al 8 no debería encontrar mucha dificultad en trasladar los algoritmos mencionados a un lenguaje de programación. Además, tampoco hay un número significativo de ejercicios escritos.

Las denominaciones que hemos utilizado para cada uno de los tipos de problemas podrían no corresponder con las que espera el lector, por lo que le remitimos al índice alfabético, incluido al final del libro, para comprobar si existen términos alternativos.

---

<sup>1</sup> Algunas empresas tecnológicas podrían usarlos en entrevistas de trabajo.

## 9.1 Problema 2-SAT

### Enunciado del problema

Recibimos una conjunción de disyunciones (“y de operaciones OR”), donde cada disyunción (“operación OR”) tiene dos argumentos, que pueden ser variables o la negación de variables. Las disyunciones de pares se denominan ‘cláusulas’ y la fórmula se conoce como 2-CNF (forma normal conjuntiva). El problema 2-SAT, debe encontrar una asignación de certeza (sea verdadera o falsa) para esas variables, que verifique la fórmula 2-CNF, es decir, cada cláusula tiene, al menos, un término que se evalúa como verdadero.

Ejemplo 1:  $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ , queda satisfecha porque podemos asignar  $x_1 = \text{verdadero}$  y  $x_2 = \text{falso}$  (la asignación alternativa sería  $x_1 = \text{falso}$  y  $x_2 = \text{verdadero}$ ).

Ejemplo 2:  $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$  no se puede satisfacer. Podemos probar las 8 combinaciones posibles de valores booleanados de  $x_1$ ,  $x_2$  y  $x_3$ , para darnos cuenta de que ninguna de ellas verifica la fórmula 2-CNF.

### Soluciones

#### Búsqueda completa

Los concursantes que solo tengan un conocimiento superficial del problema de satisfacción, pueden pensar que estamos ante un problema NP-completo y, desde ahí, intentar una solución de búsqueda completa. Si la fórmula 2-CNF tiene  $n$  variables y  $m$  cláusulas, probar las  $2^n$  asignaciones posibles, y comprobar cada asignación en  $O(m)$ , tiene una complejidad de tiempo total de  $O(2^n \times m)$ . Esto resultará con todo seguridad en un veredicto TLE.

El 2-SAT es un *caso especial* del problema de satisfacción, y cuenta con una solución polinómica como la que sigue.

#### Reducción a grafo de implicación y búsqueda de SCC

En primer lugar, debemos darnos cuenta de que una cláusula de una fórmula 2-CNF ( $a \vee b$ ), se puede expresar como  $(\neg a \Rightarrow b)$  y  $(\neg b \Rightarrow a)$ . Por lo tanto, partiendo de esta fórmula, podemos construir el ‘grafo de implicación’ correspondiente. Cada variable tiene dos vértices en el grafo de implicación, ella misma y su variable de negación/inversa<sup>2</sup>. Una arista conecta dos vértices cuando las variables correspondientes están relacionadas por una implicación de la fórmula 2-CNF. La figura 9.1 muestra los grafos de implicación de las dos fórmulas 2-CNF vistas antes.

Como se puede ver en la figura 9.1, una fórmula 2-CNF con  $n$  variables (excluyendo las negaciones) y  $m$  cláusulas, tendrá  $V = \theta(2n) = \theta(n)$  vértices y  $E = O(2m) = O(m)$  aristas, en el grafo de implicación.

Ahora, una fórmula 2-CNF quedará satisfecha si, y solo si, “no hay ninguna variable que pertenezca al mismo componente fuertemente conexo (SCC) que su negación”.

<sup>2</sup>Truco de programación: le damos a una variable un índice  $i$  y a su negación  $i + 1$ . De esta forma, podemos llegar de una a otra mediante manipulación de bits  $i \oplus 1$ , donde  $\oplus$  es el operador ‘OR exclusivo’.

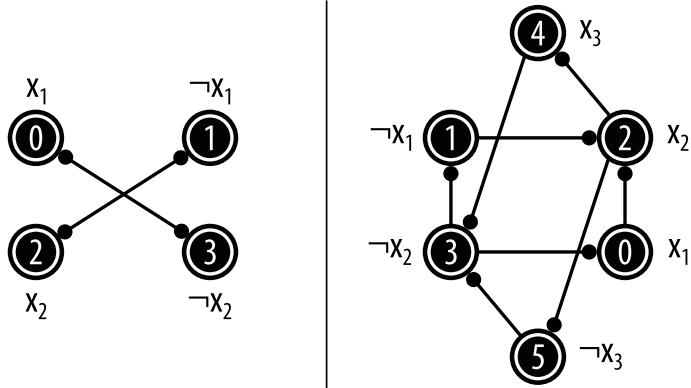


Figura 9.1: Los grafos de implicación de los ejemplos 1 (izquierda) y 2 (derecha)

En la parte izquierda de la figura 9.1, podemos ver que hay dos SCC:  $\{0,3\}$  y  $\{1,2\}$ . Como no hay ninguna variable que pertenezca al mismo SCC que su negación, podemos concluir que la fórmula 2-CNF del ejemplo 1 queda satisfecha.

En la parte derecha de la misma figura, observamos que los seis vértices pertenecen a un mismo SCC. Por lo tanto, tenemos los vértices 0 (que representa a  $x_1$ ) y 1 (que representa a<sup>3</sup>  $\neg x_1$ ), los vértices 2 ( $x_2$ ) y 3 ( $\neg x_2$ ) y los vértices 4 ( $x_3$ ) y 5 ( $\neg x_3$ ), en el mismo SCC. La conclusión es que no se puede satisfacer la fórmula 2-CNF del ejemplo 2.

Para encontrar los SCC de un grafo dirigido, podemos utilizar el algoritmo de Tarjan (sección 4.2.9) o el de Kosaraju (sección 9.17).

### Ejercicio 9.1.1\*

Para encontrar la asignación verdadera concreta, debemos realizar alguna tarea más que comprobar si ninguna variable pertenece al mismo SCC que su negación. ¿Cuáles son los pasos adicionales requeridos para encontrar, realmente, la asignación verdadera de una fórmula 2-CNF satisfecha?

### Ejercicios de programación

#### 1. UVa 10319 - Manhattan\*

La parte difícil de resolver problemas de 2-SAT, es identificar que se trata de uno de ellos y construir el grafo de implicación. En este problema, fijamos cada calle y avenida como una variable donde *verdadero* significa que solo se puede utilizar en una dirección concreta, y *falso* que solo se puede en la otra. Un camino sencillo tendrá esta forma: (calle  $a \wedge$  avenida  $b \vee$  (avenida  $c \wedge$  calle  $d$ ). Esto se puede transformar en la fórmula 2-CNF  $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$ . A partir de ahí, construir el grafo de implicación y comprobar si se satisface, mediante la comprobación de SCC. Existe un caso especial, donde la cláusula tiene un solo literal, es decir, el camino sencillo utiliza solo una calle o avenida.

<sup>3</sup>Utilizando el truco de programación anterior, podemos comprobar fácilmente si los vértices 1 y 0 son una variable y su negación, mediante  $1 = 0 \oplus 1$ .

## 9.2 Problema de la galería de arte

### Enunciado del problema

El problema de ‘la galería de arte’ conforma un conjunto de problemas de *visibilidad*, relacionados entre sí, dentro de la geometría computacional. A continuación, veremos diferentes variantes. El planteamiento común a todas ellas es el polígono simple (no necesariamente convexo)  $P$ , que describe la galería de arte; un conjunto de puntos  $S$  para identificar a los vigilantes, donde cada uno de ellos es un punto de  $P$ ; la regla de que un punto  $A \in S$  puede vigilar a otro punto  $B \in P$  si, y solo si,  $A \in S, B \in P$  y el segmento  $AB$  está contenido en  $P$ ; y una pregunta sobre si todos los puntos del polígono  $P$  están vigilados por  $S$ . Muchas de las variantes del problema de la galería de arte se clasifican como problemas NP-complejo. En este libro, nos centramos en las que tienen soluciones polinómicas.

1. Variante 1: determinar el límite superior del tamaño más pequeño del conjunto  $S$ .
2. Variante 2: determinar si  $\exists$  un punto crítico  $C$  en el polígono  $P$  y  $\exists$  otro punto  $D \in P$ , de forma que si el vigilante está en la posición  $C$ , no pueda proteger el punto  $D$ .
3. Variante 3: determinar si el polígono  $P$  se puede proteger con un solo vigilante.
4. Variante 4: determinar el tamaño más pequeño del conjunto  $S$ , si los vigilantes solo pueden estar en los vértices del polígono  $P$ , y solo hay que proteger los vértices.

Hay muchas más versiones y existe, al menos, un libro sobre ello<sup>4</sup> [46].

### Soluciones

1. La solución de la primera variante es un trabajo teórico del teorema de la galería de arte de Václav Chvátal. Afirma que  $\lfloor n/3 \rfloor$  vigilantes son siempre suficientes y, a veces, necesarios para proteger un polígono simple de  $n$  vértices (no incluimos la demostración).
2. La solución de la segunda variante implica comprobar si el polígono  $P$  es cóncavo (y, con ello, tiene un punto crítico). Podemos usar la negación de *isConvex* de la sección 7.3.4.
3. La solución de la tercera variante puede ser complicada, si no se ha visto con anterioridad. Podemos utilizar la función *cutPolygon*, tratada en la sección 7.3.6. Cortamos el polígono  $P$  con todas las líneas formadas por las aristas de  $P$ , en sentido contrario a las agujas del reloj, y guardamos, siempre, el lado izquierdo. Si, al final, seguimos teniendo un polígono que no esté vacío, se puede colocar ahí un vigilante protegiendo todo el polígono  $P$ .
4. La solución de la cuarta variante implica el cálculo de la cobertura de vértices mínima del ‘grafo de visibilidad’ del polígono  $P$ . En un grafo general, este problema es NP-complejo.

### Ejercicios de programación

- |                                                     |                                                                                                                                                                              |
|-----------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. <a href="#">UVa 00588 - Video Surveillance *</a> | (ver solución de la variante 3)                                                                                                                                              |
| 2. <a href="#">UVa 10078 - Art Gallery *</a>        | (ver solución de la variante 2)                                                                                                                                              |
| 3. <a href="#">UVa 10243 - Fire; Fire; Fire *</a>   | (variante 4: se puede reducir al problema de cobertura de vértices mínima <i>en un árbol</i> ; existe solución polinómica; en realidad la hemos tratado en la sección 4.7.1) |
| 4. LA 2512 - Art Gallery                            | (ver solución de la variante 3 y el área de un polígono)                                                                                                                     |
| 5. LA 3617 - How I Mathematician ...                | (variante 3)                                                                                                                                                                 |

<sup>4</sup>Versión en PDF en <http://cs.smith.edu/~orourke/books/ArtGalleryTheorems/art.html>.

## 9.3 Problema del viajante bitónico

### Enunciado del problema

El problema del viajante (TSP) bitónico se puede describir así: dada una lista de coordenadas de  $n$  vértices, en un espacio euclídeo bidimensional, que ya están ordenados por la coordenada  $x$  (y por la  $y$ , en caso de empate), encontrar la ruta de menor coste que comience en el vértice más a la izquierda, avance, estrictamente, de izquierda a derecha y, al llegar al vértice más a la derecha, vuelva, estrictamente de derecha a izquierda, al vértice inicial. El comportamiento de esa ruta se denomina ‘bitónico’.

La ruta resultante puede no ser la más corta posible, bajo la definición estándar del TSP (ver sección 3.5.2). La figura 9.2 muestra una comparativa entre las dos variantes. La ruta TSP 0-3-5-6-4-1-2-0 no es la ruta bitónica, porque, aunque inicialmente va de izquierda a derecha (0-3-5-6), y luego de derecha a izquierda (6-4-1), realiza otros dos pasos de izquierda a derecha (1-2) y de derecha a izquierda (2-0). La ruta 0-2-3-5-6-4-1-0 es bitónica válida, porque la podemos descomponer en dos caminos, 0-2-3-5-6 de izquierda a derecha y 6-4-1-0 en sentido contrario.

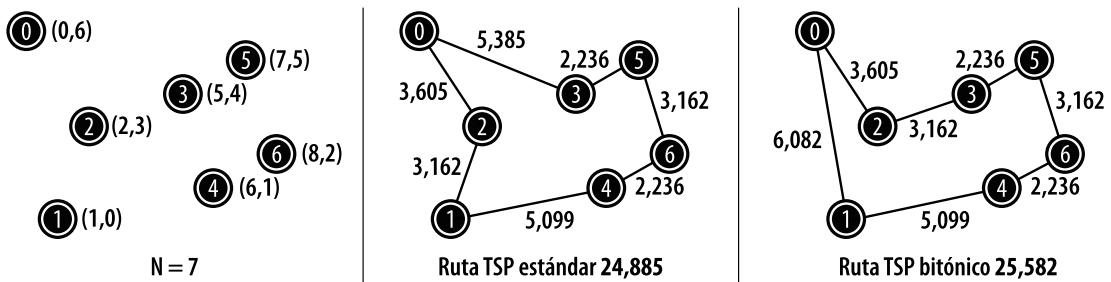


Figura 9.2: El problema del viajante estándar frente al bitónico

### Soluciones

Aunque una ruta TSP bitónica, de un conjunto de  $n$  vértices, suele ser más larga que la estándar, esta limitación bitónica nos permite calcular una ‘ruta suficientemente buena’ en  $O(n^2)$ , utilizando programación dinámica (como veremos a continuación), frente a la ruta en  $O(2^n \times n^2)$  del TSP estándar (sección 3.5.2).

La observación más importante para deducir la solución de DP, es el hecho de que podemos (y debemos) dividir la ruta en dos caminos: de izquierda a derecha (LR) y de derecha a izquierda (RL). Ambos incluyen al vértice 0 (el más a la izquierda) y al vértice  $n-1$  (el más a la derecha). El camino LR comienza en el vértice 0 y termina en el  $n-1$ . Por contra, el camino RL comienza en el vértice  $n-1$  y termina en el 0.

Recordemos que todos los vértices están ordenados por la coordenada  $x$  (y por la  $y$ , en caso de empate). Podemos considerar los vértices uno a uno. Ambos caminos, LR y RL, comienzan en el vértice 0. Digamos que  $v$  es el siguiente vértice a considerar. Para cada  $v \in [1 \dots n-2]$ , podemos decidir si añadir  $v$  como el siguiente punto del camino LR (para extender el camino LR hacia la derecha) o como el anterior para el camino RL de vuelta (el camino RL comenzará ahora en  $v$  y vuelve a 0). Para ello, debemos registrar dos parámetros más,  $p1$  y  $p2$ . Digamos que  $p1/p2$  son el vértice *final/inicial* del camino LR/RL, respectivamente.

El caso base se produce cuando  $v = n - 1$ , donde solo necesitamos conectar los dos caminos LR y RL con el vértice  $n - 1$ .

Con estas observaciones en mente, podemos escribir una solución de DP sencilla como esta:

```
1 double dp1(int v, int p1, int p2) { // se le llama con dp1(1, 0, 0)
2 if (v == n-1) return d[p1][v]+d[v][p2];
3 if (memo3d[v][p1][p2] > -0.5) return memo3d[v][p1][p2];
4 return memo3d[v][p1][p2] = min(
5 d[p1][v] + dp1(v+1, v, p2), // extender LR: p1->v, RL permanece: p2
6 d[v][p2] + dp1(v+1, p1, v)); // LR permanece: p1, extender RL: p2<-v
7 }
```

Sin embargo, la complejidad de  $dp1$  con tres parámetros ( $v$ ,  $p1$ ,  $p2$ ) es  $O(n^3)$ . No resulta eficiente, ya que el parámetro  $v$  se puede abandonar y recuperarse posteriormente de  $1 + \max(p1, p2)$  (ver esta técnica de optimización de la DP en la sección 8.3.6). La solución de DP mejorada que aparece a continuación se ejecuta en  $O(n^2)$ .

```
1 double dp2(int p1, int p2) { // se llama con dp2(0, 0)
2 int v = 1+max(p1, p2); // esta línea acelera la solución
3 if (v == n-1) return d[p1][v]+d[v][p2];
4 if (memo2d[p1][p2] > -0.5) return memo2d[p1][p2];
5 return memo2d[p1][p2] = min(
6 d[p1][v] + dp2(v, p2), // extender LR: p1->v, RL permanece: p2
7 d[v][p2] + dp2(p1, v)); // LR permanece: p1, extender RL: p2<-v
8 }
```

## Ejercicios de programación

1. [UVa 01096 - The Islands\\*](#) (LA 4791 - WorldFinals Harbin10; variante del TSP bitónico; mostrar el camino)
2. [UVa 01347 - Tour\\*](#) (LA 3305 - SoutheasternEurope05; versión pura del TSP bitónico; puedes empezar por aquí)

## 9.4 Emparejamiento de paréntesis

### Enunciado del problema

Los programadores están familiarizados con varios tipos de paréntesis: ‘()’, ‘{}’, ‘[]’, etc., ya que los utilizan constantemente en su código, cuando emplean sentencias o bucles. Los paréntesis se puede anidar, ‘((())’, ‘{{}}’’, ‘[[[]]]’, etc. Un código correcto, debe tener un conjunto de paréntesis coherente. El problema del emparejado de paréntesis implica determinar si un conjunto determinado de paréntesis está bien anidado. Por ejemplo, ‘((())’, ‘({})’, ‘(){}[]’ son correctos, mientras que ‘(((), ‘((), ‘)()’, ‘no lo son.

## Soluciones

Leemos los paréntesis, de uno en uno, de izquierda a derecha. Cada vez que encontramos un paréntesis de cierre, debemos emparejarlo con el último abierto del mismo tipo. Una vez hecho, esta pareja se deja de tener en consideración, y continúa el proceso. Necesitamos, para ello, una estructura ‘primero en entrar, último en salir’, la pila (sección 2.2).

Comenzamos con una pila vacía. Siempre que encontremos un paréntesis de apertura, lo insertamos en ella. Cuando encontremos un paréntesis de cierre, comprobaremos si es del mismo tipo que el que está en lo alto de la pila. Esto se debe a que el que esté en lo alto de la pila será el que tendremos que emparejar con el actual. Una vez que tenemos una pareja, eliminamos el elemento más alto de la pila y no lo volveremos a tener en consideración. Si conseguimos llegar al último paréntesis y, en ese momento, la pila está vacía, sabremos que el anidado es correcto.

Como debemos examinar cada uno de los  $n$  paréntesis una sola vez, y todas las operaciones de pila se ejecutan en  $O(1)$ , este algoritmo tiene, claramente, una complejidad  $O(n)$ .

## Variantes

El número de formas en que se pueden emparejar correctamente  $n$  pares de paréntesis, se encuentra con la fórmula de Catalan (sección 5.4.3). El método óptimo de multiplicación de matrices (el del problema de multiplicación de cadenas de matrices), también implica paréntesis. Esta variante se puede resolver con programación dinámica (sección 9.20).

### Ejercicios de programación

1. [UVa 00551 - Nesting a Bunch of ... \\*](#) (emparejamiento de paréntesis; stack; clásico)
2. [UVa 00673 - Parentheses Balance \\*](#) (similar a UVa 551; clásico)
3. [UVa 11111 - Generalized Matrioshkas \\*](#) (emparejamiento de paréntesis con algunos cambios)

## 9.5 Problema del cartero chino

### Enunciado del problema

El problema del cartero chino<sup>5</sup>/inspección de rutas, es el problema de hallar la longitud del camino/circuito más corto, que visita cada arista de un grafo dirigido ponderado y conexo. Si el grafo es euleriano (ver la sección 4.7.3), entonces la suma de los pesos de las aristas, a lo largo del camino euleriano que las cubre todas, es la solución óptima del problema. Este es el caso fácil. Pero cuando un grafo no es euleriano, como el de la parte izquierda de la figura 9.3, el problema se vuelve más difícil.

<sup>5</sup>El nombre se debe a que fue estudiado por primera vez por el matemático chino Mei-Ku Kuan, en 1962.

## Soluciones

La clave para resolver este problema es darse cuenta de que un grafo no euleriano  $G$ , debe tener un *número par* de vértices de grado impar (el lema del apretón de manos, enunciado por el propio Euler). Tomamos el subconjunto de vértices de  $G$ , que tienen grado impar y lo llamamos  $T$ . Ahora, creamos un grafo completo  $K_n$ , donde  $n$  es el tamaño de  $T$ .  $T$  forma los vértices de  $K_n$ . Una arista  $(i, j)$  de  $K_n$ , tiene un peso, que es el *peso del camino más corto* de un camino de  $i$  a  $j$ . Por ejemplo, en la parte central de la figura 9.3, la arista 2-5 de  $K_4$  tiene un peso  $2 + 1 = 3$ , por el camino  $2 \rightarrow 4 \rightarrow 5$ , y la arista 3-4 de  $K_4$  tiene un peso  $3 + 1 = 4$ , por el camino  $3 \rightarrow 5 \rightarrow 4$ .

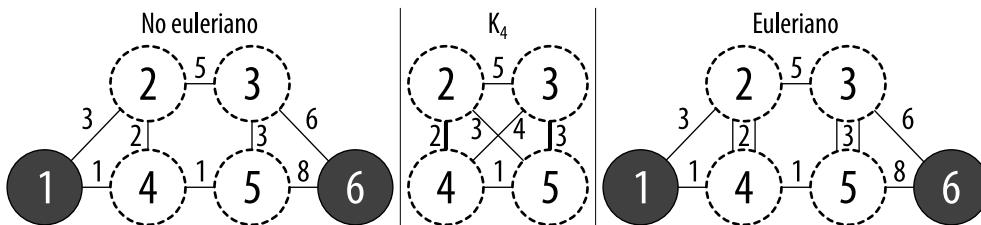


Figura 9.3: Ejemplo del problema del cartero chino

Si, ahora, *duplicamos* las aristas seleccionadas por el *emparejamiento perfecto de peso mínimo* de este grafo completo  $K_n$ , convertiremos el grafo no euleriano  $G$ , en el euleriano  $G'$ . Esto se debe a que, al duplicar las aristas, en realidad añadimos una arista entre un par de vértices de grado impar (haciendo que, con ello, pase a tener grado par). El emparejamiento perfecto de *peso mínimo* asegura que esta transformación se realiza de la *forma menos costosa*. La solución al emparejamiento perfecto de peso mínimo del  $K_4$  mostrado en la parte central de la figura 9.3, consiste en tomar las aristas 2-4 (de peso 2) y 3-5 (de peso 3).

Después de duplicar las aristas 2-4 y 3-5, volveremos a la versión fácil del problema del cartero chino. En la parte derecha de la figura 9.3, tenemos un grafo euleriano. La ruta en este grafo es sencilla. Una posible es:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 1$ , con un peso total de 34 (la suma de todos los pesos de las aristas en el grafo euleriano modificado  $G'$ , que es la suma de todos los pesos de las aristas de  $G$ , más el coste del emparejamiento perfecto de peso mínimo en  $K_n$ ).

Por lo tanto, la parte más difícil de resolver del problema del cartero chino, es encontrar el emparejamiento perfecto de peso mínimo en  $K_n$ , que *no es* un grafo bipartito (grafo completo). Si  $n$  es pequeño, podemos resolverlo con la técnica de programación dinámica con máscara de bits de la sección 8.3.1.

### Ejercicios de programación

1. **UVa 10296 - Jogging Trails \*** (ver el contenido de la sección)

## 9.6 Problema del par más cercano

### Enunciado del problema

En un conjunto  $S$  de  $n$  puntos en un plano 2D, hallar el par con menor distancia euclídea.

### Soluciones

#### Búsqueda completa

Una solución ingenua calcula las distancias entre todos los pares de puntos e informa del menor. Pero su complejidad es de  $O(n^2)$ .

#### Divide y vencerás

Podemos usar la siguiente estrategia de divide y vencerás, para lograr una complejidad  $O(n \log n)$ . Realizamos los tres pasos siguientes:

1. Dividir: ordenamos los puntos de  $S$  por su coordenada  $x$  ( $y$  en caso de empate). Después, dividimos  $S$  en dos conjuntos  $S_1$  y  $S_2$ , con una línea vertical  $x = d$ , de forma que  $|S_1| = |S_2|$  o  $|S_1| = |S_2| + 1$ , es decir, el número de puntos de cada conjunto está equilibrado.
2. Vencer: si solo tenemos un punto en  $S$ , devolvemos  $\infty$ . Si solo tenemos dos puntos, devolvemos la distancia euclídea.
3. Combinar: digamos que  $d_1$  y  $d_2$  son las distancias más pequeñas en  $S_1$  y  $S_2$ , respectivamente. Digamos también que  $d_3$  es la distancia más pequeña entre todos los pares de puntos  $(p_1, p_2)$ , donde  $p_1$  es un punto de  $S_1$  y  $p_2$  lo es de  $S_2$ . La distancia más pequeña será  $\min(d_1, d_2, d_3)$ , es decir, la respuesta puede estar en los conjuntos más pequeños de puntos  $S_1$  y  $S_2$ , o puede tener un punto en cada uno de ellos, cruzando la línea  $x = d$ .

El último paso, hecho de forma ingenua, se ejecuta en  $O(n^2)$ . Pero lo podemos optimizar. Pongamos que  $d' = \min(d_1, d_2)$ . Para cada punto a la izquierda de la línea divisoria  $x = d$ , el punto más cercano a la derecha de la línea solo puede estar dentro de un rectángulo de anchura  $d'$  y altura  $2 \times d'$ . Se puede demostrar que solo puede haber un máximo de 6 de esos puntos en este rectángulo. Esto significa que el paso de combinación solo necesitará  $O(6n)$  operaciones, y que la complejidad global de la solución es de  $T(n) = 2 \times T(n/2) + O(n)$ , o  $O(n \log n)$ .

### Ejercicio 9.6.1\*

Hay una solución más sencilla que la de divide y vencerás. Utiliza el algoritmo de la línea de barrido. ‘Barremos’ los puntos de  $S$  de izquierda a derecha. Supongamos que la mejor respuesta hasta el momento es  $d$ , y que estamos examinando el punto  $i$ . El nuevo punto más cercano potencial  $i$ , si lo es, debe tener su coordenada  $y$  a menos de  $d$  unidades del punto  $i$ . Comprobamos todos los candidatos y actualizamos  $d$  (que irá siendo más pequeño). Implementa esta solución y analiza su complejidad de tiempo.

### Ejercicios de programación

1. [UVa 10245 - The Closest Pair Problem \\*](#) (clásico; tratado en la sección)
2. [UVa 11378 - Bey Battle \\*](#) (también un problema de par más cercano)

## 9.7 Algoritmo de Dinic

En la sección 4.6, hemos visto el método, potencialmente impredecible, de Ford Fulkerson en  $O(|f^*|E)$  y el algoritmo de Edmonds Karp en  $O(VE^2)$  (búsqueda de aumentos de camino con BFS), que resulta preferible, para resolver el problema del flujo máximo. Prácticamente todos los problemas de flujo máximo de este libro se pueden resolver mediante Edmonds Karp.

Existen otros algoritmos de flujo máximo que, en teoría, tienen mejor rendimiento que el de Edmonds Karp. Uno de ellos es el algoritmo de Dinic, que se ejecuta en  $O(V^2E)$ . Como un grafo de flujo típico suele tener  $V < E$  y  $E \ll V^2$ , la complejidad del algoritmo de Dinic en el peor de los casos es, teóricamente, mejor que la de Edmonds Karp. Aunque los autores de este libro no han encontrado ningún caso en el que el algoritmo de Edmonds Karp haya recibido un veredicto de TLE, frente a un AC con Dinic, en el *mismo* grafo de flujo, *podría* ser beneficioso utilizar el algoritmo de Dinic en concursos de programación, para asegurar el resultado.

La idea de Dinic es similar a la de Edmonds Karp, ya que también encuentra los aumentos de camino iterativamente. Sin embargo, el algoritmo de Dinic utiliza el concepto de ‘flujos de bloqueo’ para encontrar esos caminos. Entender este concepto es clave para transformar el algoritmo de Edmonds Karp, fácil de entender, en el de Dinic.

Vamos a definir  $\text{dist}[v]$  como la longitud del camino más corto del vértice origen  $s$  a  $v$ , en el grafo residual. El grafo de nivel del grafo residual es  $L$ , donde la arista  $(u, v)$  de este grafo residual, está incluida en el grafo de nivel  $L$ , si  $\text{dist}[v] = \text{dist}[u] + 1$ . Por lo tanto, un ‘flujo de bloqueo’ es un flujo  $s - t$ , llamado  $f$ , de forma que, tras enviar el flujo  $f$  desde  $s$  a  $t$ , el grafo de nivel  $L$  ya no contenga un aumento de camino  $s - t$ .

Se ha demostrado (ver [11]) que el número de aristas de cada flujo de bloquillo aumenta en, al menos, una por cada iteración. Hay un máximo de  $V - 1$  flujos de bloquillo en el algoritmo, porque solo puede haber  $V - 1$  aristas en el camino sencillo ‘más largo’ de  $s$  a  $t$ . El grafo de nivel se puede construir con una BFS en  $O(E)$ , y se encuentra un flujo de bloquillo en cada grafo de nivel en  $O(VE)$ . Por lo tanto, la complejidad del peor caso de Dinic es de  $O(V \times (E + VE)) = O(V^2E)$ .

La implementación de Dinic es similar a la de Edmonds Karp, vista en la sección 4.6. En Edmonds Karp, ejecutamos una BFS, que ya nos genera el grafo de nivel  $L$ , pero solo la utilizamos para encontrar *un* aumento de camino, llamando a `augment(t, INF)`. En el caso de Dinic, necesitamos utilizar la información generada por la BFS de forma un poco distinta. El cambio clave es que, en vez de encontrar un flujo de bloquillo mediante una DFS en el grafo de nivel  $L$ , podemos simular el proceso ejecutando el procedimiento de `augment` desde *cada vértice*  $v$ , que esté conectado directamente al desagüe  $t$  del grafo de nivel, es decir, la arista  $(v, t)$  existe en el grafo de nivel  $L$  y llamamos a `augment(v, INF)`. Esto encontrará muchos, aunque no siempre todos, los *aumentos de camino* requeridos, que conforman el flujo de bloquillo del grafo de nivel  $L$ .

### Ejercicio 9.7.1\*

Implementa la *variante* del algoritmo de Dinic, a partir del código para Edmonds Karp de la sección 4.6, usando el cambio clave sugerido. Utiliza también la lista de adyacencia, como se pedía en el **ejercicio 4.6.3.3\***. Vuelve a resolver varios ejercicios de programación de la sección 4.6. ¿Notas alguna mejora de rendimiento?

### Ejercicio 9.7.2\*

Usando una estructura de datos llamada *árboles dinámicos*, el tiempo de ejecución de la búsqueda de un flujo de bloqueo se puede reducir de  $O(VE)$  a  $O(E \log V)$ , lo que hace que el peor caso de Dinic pase a ser  $O(VE \log V)$ . Estudia e implementa esta variante.

### Ejercicio 9.7.3\*

¿Qué ocurre si usamos el algoritmo de Dinic en un grafo de flujo que modele el problema MCBM de la sección 4.7.4? Consejo: ver la sección 9.12.

## 9.8 Fórmulas o teoremas

Hemos encontrado, en algunos concursos de programación, fórmulas raramente usadas. Conocerlas te dará una *ventaja absolutamente injusta* sobre otros concursantes si, en alguna ocasión, aparecen en un concurso el que participes.

1. Fórmula de Cayley: hay  $n^{n-2}$  árboles de expansión de un grafo completo con  $n$  vértices etiquetados. Ejemplo: UVa 10843 - Anne's game.
2. Desarreglo: una permutación de los elementos de un conjunto, de forma que ninguno de ellos aparezca en su posición original. El número de desarreglos  $der(n)$  se puede calcular de la siguiente forma:  $der(n) = (n-1) \times (der(n-1) + der(n-2))$ , donde  $der(0) = 1$  y  $der(1) = 0$ . Un problema sobre desarreglos es UVa 12024 - Hats (ver la sección 5.6).
3. El teorema de Erdős Gallai proporciona una condición necesaria, y suficiente, para que una secuencia de números naturales sea la *secuencia de grados* de un grafo sencillo. Una secuencia de enteros no negativos  $d_1 \geq d_2 \geq \dots \geq d_n$ , puede ser la secuencia de grados de un grafo sencillo de  $n$  vértices si  $\sum_{i=1}^n d_i$  es par y  $\sum_{i=1}^k d_i \leq k \times (k-1) + \sum_{i=k+1}^n \min(d_i, k)$  cumple con  $1 \leq k \leq n$ . Ejemplo: UVa 10720 - Graph Construction.
4. La fórmula de Euler para el grafo planar<sup>6</sup>:  $V - E + F = 2$ , donde  $F$  es el número de caras<sup>7</sup> del grafo planar. Ejemplo: UVa 10178 - Count the Faces.
5. Círculo de Moser: determinar el número de fragmentos en los que se divide un círculo, si  $n$  puntos de su circunferencia son conexos por cuerdas, sin que concurran tres. Solución:  $g(n) = {}^nC_4 + {}^nC_2 + 1$ . Ejemplo: UVa 10213 - How Many Pieces of Land? Los primeros cinco valores de  $g(n)$  son 1, 2, 4, 8, 16. Pueden parecer potencias de dos, pero no lo son.
6. Teorema de Pick<sup>8</sup>: digamos que  $i$  es el número de puntos enteros del polígono,  $A$  el área y  $b$  el número de puntos enteros del límite, entonces  $A = i + \frac{b}{2} - 1$ . Ejemplo: UVa 10088 - Trees on My Island.
7. El número de árboles de expansión de un grafo bipartito completo  $K_{n,m}$  es  $m^{n-1} \times n^{m-1}$ . Ejemplo: UVa 11719 - Gridlands Airport.

<sup>6</sup>Un grafo que se puede dibujar en el espacio euclídeo bidimensional, sin que se crucen dos aristas.

<sup>7</sup>Cuando se dibuja un grafo planar sin ningún cruce, cualquier ciclo que rodee una región sin aristas, que lleguen desde el ciclo a la región, forma una cara.

<sup>8</sup>Enunciado por Georg Alexander Pick.

## Ejercicios de programación

1. UVa 10088 - Trees on My Island  
(teorema de Pick)
2. UVa 10178 - Count the Faces  
(fórmula de Euler; un poco de búsqueda de unión)
3. **UVa 10213 - How Many Pieces ... \***  
(círculo de Moser; la fórmula es poco evidente;  $g(n) =_n C_4 +_n C_2 + 1$ )
4. **UVa 10720 - Graph Construction \***  
(teorema de Erdős-Gallai)
5. UVa 10843 - Anne's game  
(la fórmula de Cayley, para contar el número de árboles de expansión de un grafo con  $n$  vértices es  $n^{n-2}$ ; usar BigInteger de Java)
6. UVa 11414 - Dreams  
(similar a UVa 10720; teorema de Erdős-Gallai)
7. **UVa 11719 - Gridlands Airports \***  
(contar el número de árboles de expansión de un grafo bipartito completo; usar BigInteger de Java)

## 9.9 Algoritmo de eliminación gausiana

### Enunciado del problema

Una **ecuación lineal** se define como la ecuación en la que el orden de las incógnitas (variables) es **lineal** (una constante, o el producto de una constante, más la primera potencia de una incógnita). Por ejemplo, la ecuación  $X + Y = 2$  es lineal, pero  $X^2 = 4$ , no lo es.

Un **sistema de ecuaciones lineales** se define como la colección de  $n$  incógnitas (variables) en, normalmente,  $n$  ecuaciones lineales. Por ejemplo  $X + Y = 2$  y  $2X + 5Y = 6$ , donde la solución es  $X = 1\frac{1}{3}$ ,  $Y = \frac{2}{3}$ . Hay diferencias con la **ecuación diofántica lineal** (sección 5.5.9), ya que la solución de un **sistema de ecuaciones lineales** no tiene por qué estar formada por enteros.

En raras ocasiones, podemos encontrar este tipo de sistemas de ecuaciones lineales en un problema de un concurso de programación. Conocer la solución nos será muy útil.

### Soluciones

Para calcular la solución de un **sistema de ecuaciones lineales**, podemos utilizar técnicas como el algoritmo de **eliminación gausiana**. Este algoritmo aparece más habitualmente en libros de texto de ingeniería, bajo el tema de ‘métodos numéricos’. Algunos libros de ciencias de la computación también lo tratan, como [8]. A continuación, incluimos un algoritmo relativamente simple, en  $O(n^3)$ , utilizando la siguiente función de C++:

```
1 #define MAX_N 100 // ajustar este valor según sea necesario
2 struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
3 struct ColumnVector { double vec[MAX_N]; };
4
5 ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) { // O(N^3)
6 // entrada: N, aumento de matriz Aug, salida: vector de columnas X
7 int i, j, k, l; double t; ColumnVector X;
8
9 for (j = 0; j < N-1; j++) { // fase de eliminación hacia adelante
```

```

10 l = j;
11 for (i = j+1; i < N; i++) // qué fila tiene el mayor valor de columna
12 if (fabs(Aug.mat[i][j]) > fabs(Aug.mat[l][j]))
13 l = i; // recordar esta fila l
14 // intercambiar esta fila pivote, razón: minimizar error de precisión
15 for (k = j; k <= N; k++) // t es una variable double temporal
16 t = Aug.mat[j][k], Aug.mat[j][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
17 for (i = j+1; i < N; i++) // la verdadera eliminación hacia adelante
18 for (k = N; k >= j; k--)
19 Aug.mat[i][k] -= Aug.mat[j][k]*Aug.mat[i][j]/Aug.mat[j][j];
20 }
21
22 for (j = N-1; j >= 0; j--) { // fase de sustitución hacia atrás
23 for (t = 0.0, k = j+1; k < N; k++) t += Aug.mat[j][k]*X.vec[k];
24 X.vec[j] = (Aug.mat[j][N]-t) / Aug.mat[j][j]; // la respuesta
25 }
26 return X;
27 }

```



GaussianElimination.cpp



GaussianElimination.java

## Ejecución de ejemplo

En esta subsección, mostramos el funcionamiento, paso a paso, de la ‘eliminación gausiana’, mediante el siguiente ejemplo. Supongamos que tenemos este sistema de ecuaciones lineales:

$$X = 9 - Y - 2Z, \quad 2X + 4Y = 1 + 3Z, \quad 3X - 5Z = -6Y$$

En primer lugar, debemos transformar el sistema de ecuaciones lineales a su *forma básica*, es decir, colocar las incógnitas (variables) a la izquierda. Ahora tenemos:

$$1X + 1Y + 2Z = 9, \quad 2X + 4Y - 3Z = 1, \quad 3X + 6Y - 5Z = 0$$

Después, volvemos a escribir las ecuaciones lineales como una multiplicación de matrices:  $A \times x = b$ . Este truco también se utiliza en la sección 9.21. Queda así:

$$\begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ 0 \end{bmatrix}$$

Ahora, trabajaremos tanto con la matriz  $A$  (de tamaño  $N \times N$ ), como con el vector de columnas  $b$  (de tamaño  $N \times 1$ ). Así, los combinamos en la matriz de aumento  $N \times (N + 1)$  (la última columna, con tres flechas, es un comentario que ayuda a la explicación):

$$\left[ \begin{array}{ccc|c} 1 & 1 & 2 & 9 \\ 2 & 4 & -3 & 1 \\ 3 & 6 & -5 & 0 \end{array} \right] \rightarrow \begin{array}{l} 1X + 1Y + 2Z = 9 \\ 2X + 4Y - 3Z = 1 \\ 3X + 6Y - 5Z = 0 \end{array}$$

Pasamos el aumento de matriz a la función de eliminación gausiana que hemos visto. La primera fase es la de eliminación hacia adelante. Tomamos el valor absoluto más grande de la columna  $j = 0$ , desde la fila  $i = 0$  en adelante, después intercambiamos esa fila con  $i = 0$ . Este paso (adicional) ayuda a minimizar el error de precisión. Después de intercambiar las filas 0 y 2:

$$\left[ \begin{array}{ccc|c|l} 3 & 6 & -5 & 0 & \rightarrow 3X + 6Y - 5Z = 0 \\ 2 & 4 & -3 & 1 & \rightarrow 2X + 4Y - 3Z = 1 \\ 1 & 1 & 2 & 9 & \rightarrow 1X + 1Y + 2Z = 9 \end{array} \right]$$

La actividad principal realizada por el algoritmo de eliminación gausiana en esta fase de eliminación hacia adelante, es eliminar la variable  $X$  (primera variable), de la fila  $i + 1$  en adelante. En este ejemplo, eliminamos  $X$  de las filas 1 y 2. Hay que concentrarse en el comentario “la verdadera eliminación hacia adelante” del código anterior. Resultado:

$$\left[ \begin{array}{ccc|c|l} 3 & 6 & -5 & 0 & \rightarrow 3X + 6Y - 5Z = 0 \\ 0 & 0 & 0,33 & 1 & \rightarrow 0X + 0Y + 0,33Z = 1 \\ 0 & -1 & 3,67 & 9 & \rightarrow 0X - 1Y + 3,67Z = 9 \end{array} \right]$$

Seguimos con la eliminación de la siguiente variable (ahora es  $Y$ ). Tomamos el valor absoluto más grande de la columna  $j = 1$ , desde la fila  $i = 1$  en adelante. Entonces, intercambiamos esa fila con  $i = 1$ . En nuestro ejemplo, después del intercambio de la fila 1 con la 2, tendremos el siguiente aumento de matriz, en el que la variable  $Y$  ya ha sido eliminada de la fila 2:

$$\left[ \begin{array}{ccc|c|l} \text{fila 0} & 3 & 6 & -5 & 0 & \rightarrow 3X + 6Y - 5Z = 0 \\ \text{fila 1} & 0 & -1 & 3,67 & 9 & \rightarrow 0X - 1Y + 3,67Z = 9 \\ \text{fila 2} & 0 & 0 & 0,33 & 1 & \rightarrow 0X + 0Y + 0,33Z = 1 \end{array} \right]$$

Una vez que tenemos la matriz triangular inferior del aumento de matriz, con todos los valores en cero, podemos iniciar la segunda fase: sustitución hacia atrás. Nos fijamos en las últimas líneas del código de eliminación gausiana. Después de eliminar las variables  $X$  e  $Y$ , solo quedará la variable  $Z$  en la fila 2. Ahora estamos seguros de que  $Z = 1/0,33 = 3$ :

$$[\text{ fila 2 } | 0 \ 0 \ 0,33 | 1 | \rightarrow 0X + 0Y + 0,33Z = 1 \rightarrow Z = 1/0,33 = 3]$$

Una vez que tenemos  $Z = 3$ , podemos procesar la fila 1. Tendremos  $Y = (9 - 3,67 \times 3) / -1 = 2$ .

$$[\text{ fila 1 } | 0 \ -1 \ 3,67 | 9 | \rightarrow 0X - 1Y + 3,67Z = 9 \rightarrow Y = (9 - 3,67 \times 3) / -1 = 2]$$

Al tener  $Z = 3$  y  $Y = 2$ , procesamos la fila 0. Tendremos  $X = (0 - 6 \times 2 + 5 \times 3) / 3 = 1$ . Hecho:

$$[\text{ fila 0 } | 3 \ 6 \ -5 | 0 | \rightarrow 3X + 6Y - 5Z = 0 \rightarrow X = (0 - 6 \times 2 + 5 \times 3) / 3 = 1]$$

Por lo tanto, la solución al sistema de ecuaciones lineales dado es  $X = 1$ ,  $Y = 2$  y  $Z = 3$ .

### Ejercicios de programación

- UVa 11319 - Stupid Sequence?\*** (resolver el sistema de las 7 primeras ecuaciones lineales; entonces utilizar las 1500 ecuaciones para las comprobaciones de ‘secuencia inteligente’)

## 9.10 Emparejamiento de grafos

### Enunciado del problema

Emparejamiento de grafos: seleccionar un subconjunto de aristas  $M$  de un grafo  $G(V, E)$ , de forma que un mismo vértice no esté compartido por dos aristas. La mayor parte de las veces, lo que nos interesa es el emparejamiento de *cardinalidad máxima*, es decir, queremos saber el *número máximo de aristas* que podemos emparejar en un grafo  $G$ . Otra cuestión habitual es el emparejamiento *perfecto*, donde tenemos que cumplir tanto con la cardinalidad máxima como con el hecho de que ningún vértice quede sin emparejar. Si  $V$  es impar, es imposible lograr un emparejamiento perfecto. El emparejamiento perfecto se resuelve buscando la cardinalidad máxima y, después, comprobando si todos los vértices están emparejados.

Hay dos atributos importantes en los problemas de emparejamiento de grafos de los concursos de programación, que pueden alterar (de forma significativa) el nivel de dificultad: si el grafo de entrada es bipartito (es más difícil si no lo es) y si es no ponderado (es más difícil si lo es). Estas dos características generan cuatro variantes<sup>9</sup>, como mostramos a continuación (ver también la figura 9.4).

1. Emparejamiento bipartito de cardinalidad máxima no ponderado (MCBM no ponderado).  
Es la variante más sencilla y más común ( $V \leq 500$ ).
2. Emparejamiento bipartito de cardinalidad máxima ponderado (MCBM ponderado).  
Problema similar al anterior, pero ahora las aristas de  $G$  tienen peso. Normalmente, buscaremos el MCBM de menor peso total ( $V \leq 30$ ).
3. Emparejamiento de cardinalidad máxima no ponderado (MCM no ponderado).  
No se garantiza que el grafo sea bipartito ( $V \leq 100$ ).
4. Emparejamiento de cardinalidad máxima ponderado (MCM ponderado).  
Es la variante más difícil ( $V \leq 20$ ).

### Soluciones

#### Soluciones para MCBM no ponderado

Esta variante es la más sencilla, y ya hemos visto varias soluciones en las secciones 4.6 (flujo de red) y 4.7.4 (grafo bipartito). La siguiente lista enumera las tres posibles soluciones a los problemas de MCBM no ponderado:

1. Reducir el problema del MCBM no ponderado al de flujo máximo.

Ver más detalles en las secciones 4.6 y 4.7.4. La complejidad de tiempo depende del algoritmo de flujo máximo elegido, es decir, será de  $O(V^2E)$  con el algoritmo de Dinic.

---

<sup>9</sup>Hay otras variantes de emparejamiento de grafos, aparte de estas cuatro, como, por ejemplo, el problema del matrimonio estable. Sin embargo, en esta sección nos concentraremos en las cuatro mencionadas.

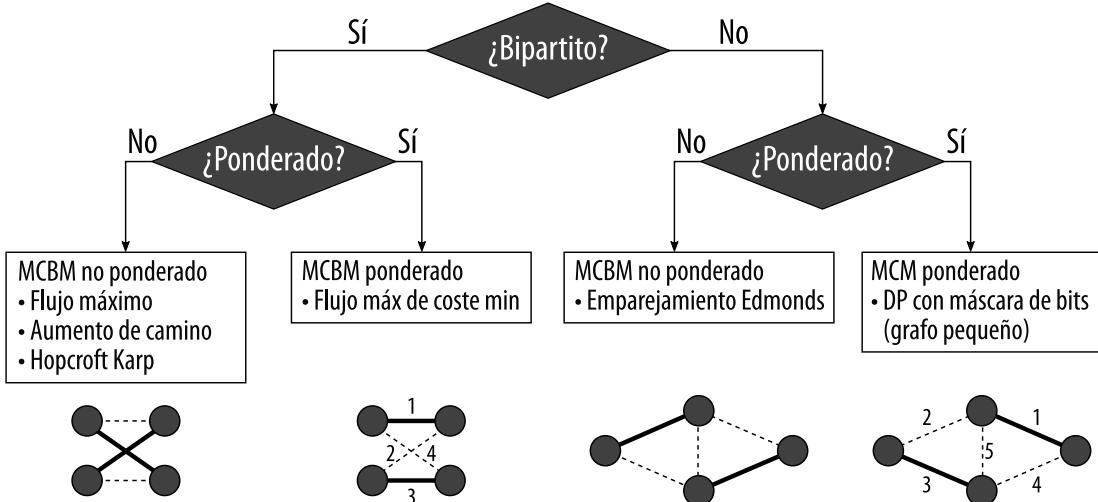


Figura 9.4: Las cuatro variantes de emparejamiento de grafos más comunes en concursos

## 2. Algoritmo de aumento de camino en $O(VE)$ para MCBM no ponderado.

Ver más detalles en la sección 4.7.4. Es lo suficientemente bueno para varios problemas de concursos que implican MCBM no ponderados.

## 3. Algoritmo de Hopcroft Karp en $O(\sqrt{V}E)$ para MCBM no ponderados.

Ver más detalles en la sección 9.12.

### Soluciones para MCBM ponderado

Cuando las aristas del grafo bipartito son ponderadas, no todos los MCBM posibles son óptimos. Necesitamos escoger un MCBM (no necesariamente único) que tenga el mínimo peso total. Una solución posible<sup>10</sup> es reducir el problema del MCBM ponderado al de flujo máximo de coste mínimo (MCMF) (ver sección 9.23).

Por ejemplo, en la figura 9.5 mostramos un caso de prueba del problema UVa 10746 - Crime Wave - The Sequel. Se trata de un problema de MCBM en un grafo bipartito completo  $K_{n,m}$ , pero donde cada arista tiene un coste asociado. Añadimos aristas desde el origen  $s$  hacia los vértices del lado izquierdo, con capacidad 1 y coste 0. También añadimos aristas desde los vértices del lado derecho, hasta el desagüe  $t$ , igualmente con capacidad 1 y coste 0. Las aristas dirigidas del lado izquierdo al derecho tienen capacidad 1 y coste de acuerdo al enunciado del problema. Una vez hemos obtenido este grafo de flujo ponderado, podemos ejecutar el algoritmo MCMF, como se explica en la sección 9.23, para lograr la respuesta buscada: flujo 1 = 0 → 2 → 4 → 8 con coste 5, flujo 2 = 0 → 1 → 4 → 2 (cancelar flujo 2-4) → 6 → 8 con coste 15 y flujo 3 = 0 → 3 → 5 → 8 con coste 20. El coste mínimo total es de  $5 + 15 + 20 = 40$ .

<sup>10</sup>Otra posible solución, si queremos obtener un emparejamiento bipartito *perfecto* de coste mínimo, es el algoritmo húngaro (Kuhn Munkres).

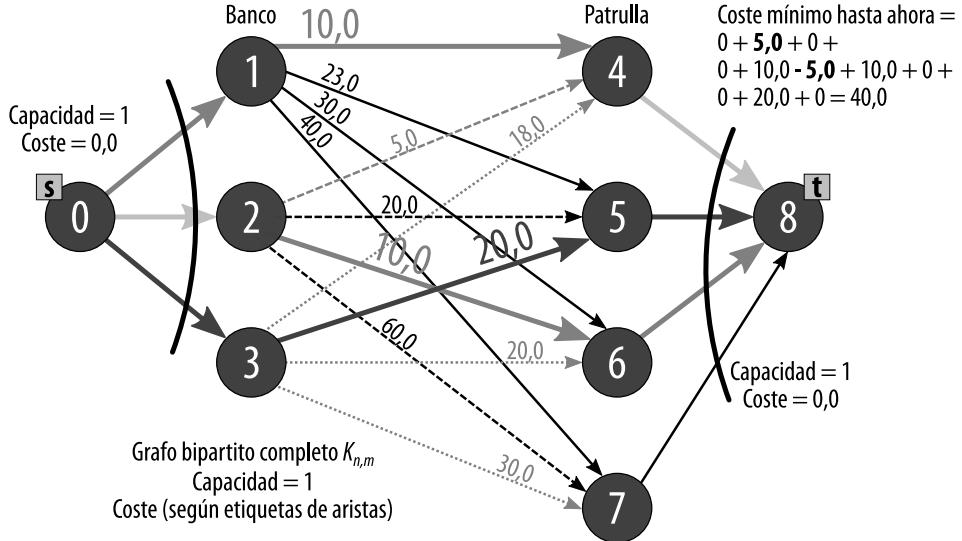


Figura 9.5: Caso de ejemplo de UVa 10746: 3 emparejamientos con coste mínimo = 40

### Soluciones para MCM no ponderado

Mientras que el problema de emparejamiento de grafos es sencillo con grafos bipartitos, es complejo en grafos generales. En el pasado, los científicos de la computación pensaron que estaban ante otro problema NP-completo, hasta que Jack Edmonds publicó, en su artículo de 1965 titulado “*Paths, trees, and flowers*” [13], un algoritmo polinómico para resolverlo.

El problema principal es que, en un grafo general, podemos encontrar aumentos de ciclos de longitud impar. Edmonds llamó a esos ciclos ‘flores’. La idea clave del algoritmo de emparejamiento de Edmonds es hundir repetidamente esas flores (potencialmente de forma recursiva), para que encontrar aumentos de camino vuelva al caso fácil del grafo bipartito. Entonces, el algoritmo de emparejamiento de Edmonds reajusta los emparejamientos, cuando esas flores vuelven a expandirse (se elevan).

La implementación del algoritmo de emparejamiento de Edmonds no es evidente. Por tanto, para hacer esta variante del emparejamiento de grafos más manejable, muchos autores de problemas limitan el tamaño de sus grafos generales no ponderados, para que sean lo suficientemente pequeños, del orden  $V \leq 18$ , y se pueda utilizar un algoritmo de DP con máscara de bits en  $O(V \times 2^V)$  para resolverlos (ver el [ejercicio 8.3.1.1](#)).

### Solución para MCM ponderado

Esta es, potencialmente, la variante más compleja. El grafo dado es un grafo general y las aristas tienen pesos asociados. En el entorno típico de un concurso de programación, la solución más probable será la DP con máscara de bits (sección 8.3.1), ya que los autores de los problemas procurarán que solo haya *un grafo general pequeño*.

## Visualización de emparejamiento de grafos

Para ayudar al lector a entender estas variantes de emparejamiento de grafos, y sus soluciones, hemos creado la siguiente herramienta de visualización:



Con ella, podrás dibujar tu propio grafo, y el sistema presentará los algoritmos de emparejamiento de grafos correctos, en base a dos características: si el grafo de entrada es bipartito y/o ponderado. La visualización del emparejamiento de Edmonds en nuestra herramienta es, probablemente, una de las primeras del mundo.

### Ejercicio 9.10.1\*

Implementa el algoritmo de Kuhn Munkres (consulta el artículo original [39, 45]).

### Ejercicio 9.10.2\*

Implementa el algoritmo de emparejamiento de Edmonds (consulta el artículo [13]).

## Ejercicios de programación

Hay algunos problemas de asignación (emparejamiento bipartito con capacidad) en la sección 4.6.

Hay algunos problemas de MCBM no ponderado y sus variantes en la sección 4.7.4.

Hay algunos problemas de MCBM ponderado en la sección 9.23.

### MCM no ponderado

1. [UVa 11439 - Maximizing the ICPC](#)\* (búsqueda binaria de la respuesta para obtener el peso mínimo; usar ese peso para reconstruir el grafo; usar el algoritmo de emparejamiento de Edmonds para comprobar si podemos lograr un emparejamiento perfecto en un grafo general)

Ver problemas sobre MCM (no) ponderado en un *grafo general pequeño* en la sección 8.3 (DP).

## 9.11 Distancia de círculo máximo

### Enunciado del problema

Una **esfera** es un objeto geométrico perfectamente redondo en un espacio tridimensional.

La **distancia de círculo máximo** entre dos puntos  $A$  y  $B$  en una esfera, es la distancia más corta sobre la **superficie de la esfera**. Este camino es un *arco* en el **círculo máximo** de esa esfera, que pasa por los dos puntos  $A$  y  $B$ . Podemos entender el círculo máximo como el círculo resultante de cortar la esfera con un plano, de forma que tengamos dos semiesferas *iguales* (ver la izquierda y el centro de la figura 9.6).

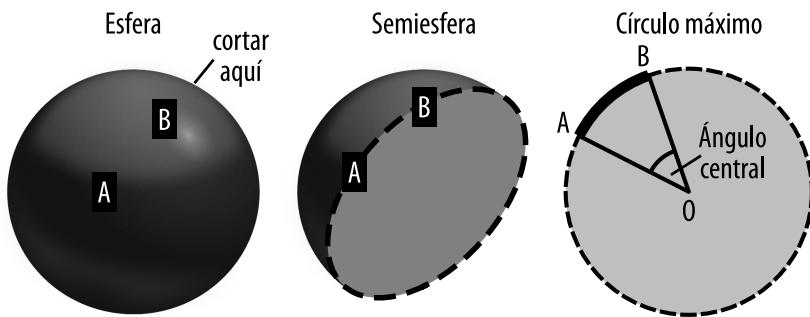


Figura 9.6: I: esfera, C: semiesfera y círculo máximo, D: gcDistance (arco  $A - B$ )

### Soluciones

Para encontrar la distancia de círculo máximo, tenemos que encontrar primero el ángulo central  $AOB$  (ver la parte derecha de la figura 9.6) del círculo máximo, donde  $O$  es el centro de ese círculo máximo (que es, también, el centro de la esfera). Dado el radio de la esfera/círculo máximo, podemos entonces determinar la longitud del arco  $A - B$ , que resulta ser la distancia de círculo máximo requerida.

Aunque ya es poco habitual, algunos problemas de concursos relativos a ‘la Tierra’, ‘líneas aéreas’, etc., utilizan la medida de esta distancia. Normalmente, los dos puntos de la superficie de la esfera se proporcionan como coordenadas terrestres, es decir, un par (latitud, longitud). El siguiente código de biblioteca nos ayudará a obtener la distancia de círculo máximo dados dos puntos y el radio de la esfera.

```
1 double gcDistance(double pLat, double pLong,
2 double qLat, double qLong, double radius) {
3 pLat *= PI/180; pLong *= PI/180; // convertir grados a radianes
4 qLat *= PI/180; qLong *= PI/180;
5 return radius * acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
6 cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
7 sin(pLat)*sin(qLat)); }
```



UVa11817.cpp



UVa11817.java

## Ejercicios de programación

1. UVa 00535 - Globetrotter (gcDistance)
2. UVa 10316 - Airline Hub (gcDistance)
3. UVa 10897 - Travelling Distance (gcDistance)
4. UVa 11817 - Tunnelling The Earth (gcDistance; distancia euclídea tridimensional)

## 9.12 Algoritmo de Hopcroft Karp

El algoritmo de Hopcroft Karp [28], es otro algoritmo utilizado para resolver el problema del emparejamiento bipartito de cardinalidad máxima no ponderado (MCBM), que complementa a la solución basada en el flujo máximo (más larga de programar) y el aumento de camino (que es el método de preferencia), tratados en la sección 4.7.4.

En nuestra opinión, la razón principal para utilizar el algoritmo de Hopcroft Karp, de código más largo, para resolver el MCBM no ponderado, frente al más corto y sencillo de aumento de camino, es el tiempo de ejecución. El algoritmo de Hopcroft Karp tiene una complejidad de  $O(\sqrt{VE})$ , que es (mucho) más rápida que la de  $O(VE)$  del aumento de camino, en grafos bipartitos (y densos) de tamaño medio ( $V \approx 500$ ).

Un ejemplo extremo es un grafo bipartito completo  $K_{n,m}$ , con  $V = n + m$  y  $E = n \times m$ . En un grafo así, el peor caso del algoritmo del aumento de camino tiene una complejidad de tiempo de  $O((n+m) \times n \times m)$ . Si  $m = n$ , tendremos una solución en  $O(n^3)$ , que solo sirve para  $n \leq 200$ .

El principal problema con el algoritmo del aumento de camino en  $O(VE)$ , es que podríamos explorar primero los aumentos de camino más largos (lo que es, en esencia, una ‘DFS modificada’). No resulta eficiente. Al explorar primero los aumentos de camino *más cortos*, Hopcroft y Karp demostraron que su algoritmo solo necesitaría  $O(\sqrt{V})$  iteraciones [28]. En cada una de ellas, el algoritmo de Hopcroft Kart ejecuta una BFS en  $O(E)$ , desde todos los vértices libres del conjunto izquierdo, y encuentra caminos de aumento de longitud creciente (comenzando desde la longitud 1: una arista libre, longitud 3: una arista libre, una arista emparejada y otra arista libre, longitud 5, 7, etc.). Después, realiza otra DFS en  $O(E)$ , para aumentar esos aumentos de camino (Hopcroft Karp puede aumentar *más de un emparejamiento* en cada iteración). Por lo tanto, la complejidad de tiempo global es de  $O(\sqrt{VE})$ .

En el ejemplo extremo del grafo bipartito completo  $K_{n,m}$ , visto antes, el algoritmo de Hopcroft Karp tiene una complejidad de tiempo, en el peor de los casos, de  $O(\sqrt{(n+m)} \times n \times m)$ .

Si  $m = n$ , tendremos una solución en  $O(n^{\frac{5}{2}})$ , que será válida para  $n \leq 600$ . Por lo tanto, si el autor del problema no ha sido lo suficientemente ‘amable’ y ha establecido  $n \approx 500$  y un grafo bipartito relativamente denso, en un problema de MCBM no ponderado, utilizar Hopcroft Karp resulta más seguro que el algoritmo estándar de aumentos de camino (sin embargo, en el **ejercicio 4.7.4.3\***, se presenta una técnica de programación competitiva inteligente, para hacer que el algoritmo del aumento de camino sea lo ‘suficientemente rápido’).

### Ejercicio 9.12.1\*

Implementa el algoritmo de Hopcroft Karp, a partir del algoritmo de aumento de camino de la sección 4.7.4, utilizando las ideas mencionadas.

### Ejercicio 9.12.2\*

Investiga los parecidos y las diferencias entre los algoritmos de Hopcroft Karp y Dinic, de la sección 9.7.

## 9.13 Caminos independientes y arista-disjuntos

### Enunciado del problema

Se dice que dos caminos que empiezan en un vértice  $s$  y van hasta un vértice de desagüe  $t$ , son *independientes* (vértice-disjuntos), si no comparten ningún vértice, aparte de  $s$  y  $t$ ,

También se dice que dos caminos que empiezan en un vértice  $s$  y van hasta un desagüe  $t$ , son arista-disjuntos si no comparten ninguna arista (pero pueden compartir más vértices que  $s$  y  $t$ ).

Dado un grafo  $G$ , encontrar el número máximo de caminos independientes y arista-disjuntos desde el origen  $s$  hasta el desagüe  $t$ .

### Soluciones

El problema de encontrar el número máximo de caminos independientes desde el origen  $s$  al desagüe  $t$ , se puede reducir al problema de flujo de red (máximo). Construimos una red de flujo  $N = (V, E)$ , a partir de  $G$ , con capacidades en los vértices, donde  $N$  es un calco de  $G$ , con la salvedad de que la capacidad de cada  $v \in V$  es 1 (es decir, cada vértice se puede utilizar una sola vez, ver cómo en la sección 4.6) y la capacidad de cada  $e \in E$  también es 1 (es decir, cada arista también se puede utilizar una sola vez). Despues, ejecutamos el algoritmo de Edmonds Karp con normalidad.

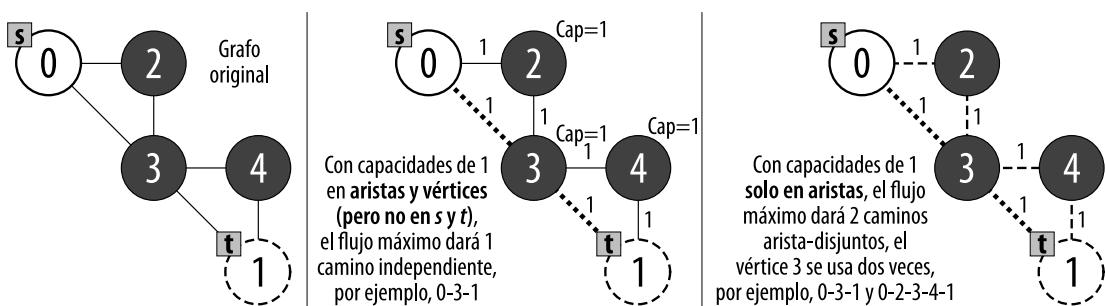


Figura 9.7: Comparativa de caminos independientes y caminos arista-disjuntos máximos

Encontrar el número máximo de caminos arista-disjuntos, desde  $s$  a  $t$ , es similar a encontrar los caminos independientes (máximos). La única diferencia es que, esta vez, no asignaremos capacidad a los vértices, lo que implica que dos caminos arista-disjuntos pueden compartir un mismo vértice. En la figura 9.7, hay una comparativa entre los caminos independientes y los caminos arista-disjuntos máximos desde  $s = 0$  a  $t = 6$ .

### Ejercicios de programación

1. [UVa 00563 - Crimewave \\*](#) (comprobar si el número máximo de caminos independientes del grafo de flujo, con capacidades de aristas y vértices de 1, es igual a  $n$  bancos)
2. [UVa 01242 - Necklace \\*](#) (LA 4271 - Hefei08; para obtener un collar, necesitamos lograr *dos* flujos arista-disjuntos  $s-t$ )

## 9.14 Índice de inversión

### Enunciado del problema

El problema del índice de inversión se define así: dada una lista de números, contar el número mínimo de intercambios de ‘ordenación de burbuja’ (intercambios entre pares de elementos consecutivos) necesarios para ordenar esa lista (normalmente de forma ascendente).

Por ejemplo, si el contenido de la lista es  $\{3, 2, 1, 4\}$ , necesitamos tres intercambios de ‘ordenación de burbuja’, para ordenarla de forma ascendente, es decir,  $(3, 2)$  para obtener  $\{2, \underline{3}, 1, 4\}$ ,  $(3, 1)$  para  $\{2, \underline{1}, \underline{3}, 4\}$  y, finalmente,  $(2, 1)$  para  $\{\underline{1}, \underline{2}, 3, 4\}$ .

### Soluciones

#### Solución en $O(n^2)$

La solución más obvia es contar el número de intercambios necesarios durante la ejecución del algoritmo de ordenación de burbuja en  $O(n^2)$ .

#### Solución en $O(n \log n)$

La mejor solución de divide y vencerás en  $O(n \log n)$ , para este problema de índice de inversión, es modificar la ordenación de mezcla. Durante el proceso de mezcla de esa ordenación, si el primer elemento de la sublistas ordenadas derecha se toma antes que el primer elemento de la sublistas ordenadas izquierda, decimos que se ha ‘producido una inversión’. Incrementamos el contador de índice de inversión en el tamaño de la sublistas izquierdas actuales. Al completarse la ordenación por mezcla, informamos del valor del contador. Como solo le añadimos  $O(1)$  pasos a la ordenación de mezcla, esta solución tiene su misma complejidad de tiempo de  $O(n \log n)$ .

En el ejemplo anterior, tenemos inicialmente  $\{3, 2, 1, 4\}$ . La ordenación por mezcla lo dividirá en las sublistas  $\{3, 2\}$  y  $\{1, 4\}$ . La sublistas izquierdas provocará una inversión, ya que tenemos que intercambiar 3 y 2 para obtener  $\{2, 3\}$ . La sublistas derechas  $\{1, 4\}$  no causará ninguna inversión, porque ya está ordenada. Ahora, mezclamos  $\{2, 3\}$  con  $\{1, 4\}$ . El primer número que

tomaremos será el 1, del principio de la sublistas derecha. Habrá dos inversiones más, porque la sublistas izquierda tiene dos miembros, {2, 3}, que se deben intercambiar con 1. Después de esto, no habrá más inversiones. Por lo tanto, en este ejemplo hay un total de 3 inversiones.

## Ejercicios de programación

- |                                     |                                                                       |
|-------------------------------------|-----------------------------------------------------------------------|
| 1. UVa 00299 - Train Swapping       | (se resuelve con ordenación de burbuja en $O(n^2)$ )                  |
| 2. <b>UVa 00612 - DNA Sorting *</b> | (necesita stable_sort en $O(n^2)$ )                                   |
| 3. <b>UVa 10327 - Flip Sort *</b>   | (se resuelve con ordenación de burbuja en $O(n^2)$ )                  |
| 4. UVa 10810 - Ultra Quicksort      | (necesita ordenación por mezcla en $O(n \log n)$ )                    |
| 5. UVa 11495 - Bubbles and Buckets  | (necesita ordenación por mezcla en $O(n \log n)$ )                    |
| 6. <b>UVa 11858 - Frosh Week *</b>  | (necesita ordenación por mezcla en $O(n \log n)$ ; entero de 64 bits) |

## 9.15 Problema de Josefo

### Enunciado del problema

El problema de Josefo es un problema clásico, en el que, inicialmente, tenemos a  $n$  personas formando un círculo, numeradas 1, 2, ...,  $n$ . Se va a ejecutar a cada persona  $k$ -ésima, y se la eliminará del círculo. Este proceso de conteo y ejecución se repite, hasta que solo quede una persona, que se salvará (la historia dice que su nombre era Josefo).

### Soluciones

#### Búsqueda completa en instancias pequeñas

Las instancias más pequeñas del problema de Josefo se pueden resolver con búsqueda completa (sección 3.2), sencillamente simulando el proceso, con ayuda de un *array* cíclico (o una lista enlazada circular). Pero las instancias más grandes necesitan soluciones mejores.

#### Caso especial cuando $k = 2$

Hay una elegante manera de determinar la posición del último superviviente para  $k = 2$ , utilizando la representación binaria del número  $n$ . Si  $n = 1b_1b_2b_3..b_n$ , entonces la respuesta es  $b_1b_2b_3..b_n1$ , es decir, movemos el bit más significativo de  $n$  al final, para hacer que sea el menos significativo. De esta forma, el problema de Josefo con  $k = 2$  se resuelve en  $O(1)$ .

#### Caso general

Digamos que  $F(n, k)$  determina la posición del superviviente en un círculo de tamaño  $n$ , con la regla de salto  $k$ , y numeramos a las personas 0, 1, ...,  $n - 1$  (sumaremos 1 a la respuesta final, para cumplir con el enunciado del problema original). Después de que la persona  $k$ -ésima muera, el círculo se reduce en un miembro, al tamaño  $n - 1$ , y la posición del superviviente

pasará a ser  $F(n-1, k)$ . Esta relación queda recogida en la ecuación  $F(n, k) = (F(n-1, k) + k) \% n$ . El caso base lo encontramos cuando  $n = 1$ , donde tenemos  $F(1, k) = 0$ . Esta recurrencia tiene una complejidad de tiempo de  $O(n)$ .

### Otras variantes

El problema de Josefo tiene tantas variantes que no podrían ser incluidas en este libro.

### Ejercicios de programación

- |                                            |                                                        |
|--------------------------------------------|--------------------------------------------------------|
| 1. UVa 00130 - Roman Roulette              | (problema de Josefo original)                          |
| 2. UVa 00133 - The Dole Queue              | (fuerza bruta; similar a UVa 130)                      |
| 3. UVa 00151 - Power Crisis                | (problema de Josefo original)                          |
| 4. UVa 00305 - Joseph                      | (se puede calcular previamente la respuesta)           |
| 5. UVa 00402 - M*A*S*H                     | (Josefo modificado; simulación)                        |
| 6. UVa 00440 - Eeny Meeny Moo              | (fuerza bruta; similar a UVa 151)                      |
| 7. UVa 10015 - Joseph's Cousin             | (Josefo modificado; $k$ dinámica; variante de UVa 305) |
| 8. <u>UVa 10771 - Barbarian tribes *</u>   | (fuerza bruta; tamaño de entrada pequeño)              |
| 9. <u>UVa 10774 - Repeated Josephus *</u>  | (caso repetido de Josefo cuando $k = 2$ )              |
| 10. <u>UVa 11351 - Last Man Standing *</u> | (usar la recurrencia del caso general)                 |

## 9.16 Movimientos del caballo

### Enunciado del problema

En el ajedrez, el caballo se mueve haciendo un recorrido ‘en forma de L’. Formalmente, un caballo puede saltar de una casilla  $(r_1, c_1)$  a otra  $(r_2, c_2)$ , en un tablero de  $n \times n$ , si, y solo si,  $(r_1 - r_2)^2 + (c_1 - c_2)^2 = 5$ . Una consulta común es el número mínimo de movimientos necesarios para llevar al caballo de una casilla inicial a otra de destino. Puede haber muchas consultas en un mismo tablero de ajedrez.

### Soluciones

#### Una BFS por consulta

Si el tablero de ajedrez es pequeño, podemos permitirnos ejecutar una BFS por consulta. Lo hacemos ejecutando la BFS desde la casilla de inicio. Cada casilla tiene un máximo de 8 aristas, conectadas a otras casillas (las que están en los límites del tablero tienen menos). Detenemos la BFS tan pronto como lleguemos a la casilla de destino. Podemos utilizar la BFS en este problema de camino más corto, ya que el grafo no es ponderado (sección 4.4.2). Como hay hasta  $O(n^2)$  casillas en un tablero de ajedrez, la complejidad de tiempo total es de  $O(n^2 + 8n^2) = O(n^2)$  por consulta, o  $O(Qn^2)$  si hay  $Q$  consultas.

### Una BFS calculada previamente y tratamiento de casos especiales

La solución anterior no es la más eficiente para resolver el problema. Si el tablero de ajedrez dado es grande y hay varias consultas como, por ejemplo,  $n = 1000$  y  $Q = 16$ , en el problema UVa 11643 - Knight Tour, la técnica anterior tendrá un veredicto de TLE.

Una solución mejor es aprovechar el hecho de que, si el tablero es lo suficientemente grande y elegimos dos casillas aleatorias  $(r_a, c_a)$  y  $(r_b, c_b)$  en su centro, con el mínimo de movimientos de  $d$  pasos entre ellos, desplazar las posiciones de las casillas por un factor constante no alterará la respuesta, es decir, el mínimo de movimientos del caballo desde  $(r_a + k, c_a + k)$  y desde  $(r_b + k, c_b + k)$ , sigue siendo de  $d$  pasos, para un factor constante  $k$ .

Por lo tanto, podemos ejecutar *una* BFS desde una casilla de origen arbitraria y hacer algunos ajustes en la respuesta. Sin embargo, hay algunos casos límite de los que ocuparse. Encontrarlos puede ser un auténtico problema, y debemos estar preparados para recibir muchos veredictos WA si no los conocemos. Para hacer esta sección interesante, hemos derivado este último paso crucial a un ejercicio. Intenta el problema UVa 11643 después de obtener estas respuestas.

### Ejercicio 9.16.1\*

Busca los casos especiales y ocípate de ellos. Pistas:

1. Casos separados cuando  $3 \leq n \leq 4$  y  $n \geq 5$ .
2. Concéntrate en las casillas de las esquinas y los bordes del tablero.
3. ¿Qué ocurre si las casillas de inicio y destino están demasiado cerca?

### Ejercicios de programación

1. [UVa 00439 - Knight Moves \\*](#) (basta con una BFS por consulta)
2. [UVa 11643 - Knight Tour \\*](#) (la distancia entre dos posiciones interesantes cualesquiera se puede obtener utilizando una tabla BFS calculada previamente (y ocupándose de los casos límite); después, se convierte en un problema TSP clásico; ver la sección 3.5.2)

## 9.17 Algoritmo de Kosaraju

Encontrar componentes fuertemente conexos (SCC) en un grafo dirigido es un problema de grafos clásico, que hemos tratado en la sección 4.2.9. Hemos visto una DFS modificada, llamada algoritmo de Tarjan, que puede resolver este problema eficientemente, en un tiempo  $O(V + E)$ .

En esta sección, presentamos otro algoritmo basado en la DFS, que se puede utilizar para encontrar los SCC de un grafo dirigido, llamado algoritmo de Kosaraju. La idea básica de este algoritmo es ejecutar la DFS *dos veces*. La *primera* DFS opera sobre el *grafo dirigido original*, y

registra el recorrido ‘postorden’ de los vértices, como en la búsqueda del orden topológico<sup>11</sup> de la sección 4.2.5. La *segunda* DFS se realiza en la *transposición del grafo dirigido original*, utilizando el ‘postorden’ encontrado por la primera DFS. Estas dos pasadas de DFS son suficientes para encontrar los SCC del grafo dirigido. A continuación, incluimos la implementación en C++ de este algoritmo. Animamos al lector a que lea más detalles de cómo funciona el algoritmo en [7].

```

1 void Kosaraju(int u, int pass) { // pass = 1 (original), 2 (transposición)
2 dfs_num[u] = 1;
3 vvi neighbor; // usar una lista de adyacencia diferente en las 2 pasadas
4 if (pass == 1) neighbor = AdjList[u]; else neighbor = AdjListT[u];
5 for(int j = 0; j < (int)neighbor.size(); j++) {
6 ii v = neighbor[j];
7 if (dfs_num[v.first] == DFS_WHITE)
8 Kosaraju(v.first, pass);
9 }
10 S.push_back(u); // como en búsqueda del orden topológico en sección 4.2.5
11 }
12
13 // en int main()
14 S.clear(); // primera pasada registra el 'postorden' del grafo original
15 dfs_num.assign(N, DFS_WHITE);
16 for (i = 0; i < N; i++)
17 if (dfs_num[i] == DFS_WHITE)
18 Kosaraju(i, 1);
19 numSCC = 0; // segunda pasada: explorar SCC según resultado de la primera
20 dfs_num.assign(N, DFS_WHITE);
21 for (i = N - 1; i >= 0; i--)
22 if (dfs_num[S[i]] == DFS_WHITE) {
23 numSCC++;
24 Kosaraju(S[i], 2); // AdjListT -> transposición del grafo original
25 }
26
27 printf("There are %d SCCs\n", numSCC);

```



UVa11383.cpp



UVa11838.java

El algoritmo de Kosaraju necesita la rutina de transposición del grafo (o construir dos estructuras de datos de grafos por adelantado), como se mencionaba brevemente en la sección 2.4.1, y necesita dos pasadas por la estructura de datos del grafo. El algoritmo de Tarjan, de la sección 4.2.9, no necesita la transposición del grafo y solo realiza una pasada. Sin embargo, los dos algoritmos de búsqueda de SCC son igualmente buenos y se pueden utilizar para resolver prácticamente todos los problemas de SCC de este libro.

---

<sup>11</sup>Pero puede no ser un orden topológico válido, ya que el grafo dirigido de entrada podría ser cíclico

## 9.18 Ancestro común mínimo

### Enunciado del problema

Dado un árbol  $T$  con raíz y  $n$  vértices, el ancestro común mínimo (LCA) entre dos vértices  $u$  y  $v$ , o  $LCA(u, v)$ , se define como el menor vértice de  $T$  que tenga a  $u$  y  $v$  como descendientes. Permitimos que un vértice sea descendiente de sí mismo, es decir, cabe la posibilidad de que  $LCA(u, v) = u$  o  $LCA(u, v) = v$ .

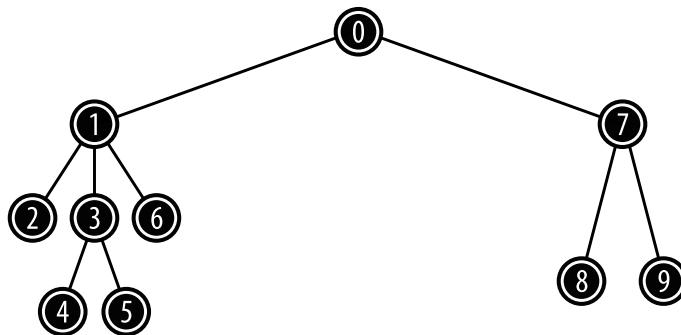


Figura 9.8: Ejemplo de un árbol  $T$  con raíz con  $n = 10$  vértices

Por ejemplo, en la figura 9.8, verificamos que  $LCA(4, 5) = 3$ ,  $LCA(4, 6) = 1$ ,  $LCA(4, 1) = 1$ ,  $LCA(8, 9) = 7$ ,  $LCA(4, 8) = 0$  y  $LCA(0, 0) = 0$ .

### Soluciones

#### Solución de búsqueda completa

Una solución ingenua es hacerlo en dos pasos. Viajamos desde el primer vértice  $u$  hasta la raíz de  $T$ , y registramos todos los vértices recorridos (que podría significar  $O(n)$ , si el árbol está muy desequilibrado). Desde el segundo vértice  $v$ , también podemos ir hasta la raíz de  $T$  pero, esta vez, nos detendremos la primera vez que encontramos un vértice común (también puede ser  $O(n)$  si el  $LCA(u, v)$  es la raíz y el árbol está muy desequilibrado). Ese vértice común es el LCA. Necesitamos  $O(n)$  por cada consulta  $(u, v)$ , lo que resulta muy lento con muchas consultas.

Por ejemplo, si queremos calcular el  $LCA(4, 6)$  del árbol de la figura 9.8, utilizando esta solución de búsqueda completa, primero tenemos que recorrer el camino  $4 \rightarrow 3 \rightarrow 1 \rightarrow 0$ , y registramos esos 4 vértices. Después, recorremos el camino  $6 \rightarrow 1$  y nos detenemos. Informamos de que el LCA es el vértice 1.

#### Reducción a consulta del rango mínimo

Podemos reducir el problema del LCA al de consulta del rango mínimo (RMQ) (sección 2.4.3). Si la estructura del árbol  $T$  no cambia durante las  $Q$  consultas, podemos utilizar una estructura de datos de tabla dispersa, con tiempo de construcción  $O(n \log n)$  y tiempo de RMQ  $O(1)$ . En la

sección 9.33 veremos en detalle la estructura de tabla dispersa. De momento, vamos a analizar el proceso de reducción de LCA a RMQ.

Podemos reducir el LCA a RMQ en tiempo lineal. La idea clave está en observar que  $LCA(u, v)$  es el vértice más superficial del árbol que se visita durante un recorrido DFS a  $u$  y  $v$ . Lo que tenemos que hacer, es ejecutar una DFS sobre el árbol y registrar información sobre la profundidad y el tiempo de visita de cada nodo. Alcanzaremos un total de  $2 \times n - 1$  vértices durante la DFS, porque los vértices internos se visitarán varias veces. Necesitamos construir tres *arrays* durante esta DFS:  $E[0..2*n-2]$  (que registra la secuencia de nodos visitados),  $L[0..2*n-2]$  (que registra la profundidad de cada nodo visitado) y  $H[0..N-1]$  (donde  $H[i]$  registra el índice de la primera aparición del nodo  $i$  en  $E$ ).

A continuación, incluimos la sección clave de la implementación:

```

1 int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;
2
3 void dfs(int cur, int depth) {
4 H[cur] = idx;
5 E[idx] = cur;
6 L[idx++] = depth;
7 for (int i = 0; i < children[cur].size(); i++) {
8 dfs(children[cur][i], depth+1);
9 E[idx] = cur; // vuelta al nodo actual
10 L[idx++] = depth;
11 }
12 }
13
14 void buildRMQ() {
15 idx = 0;
16 memset(H, -1, sizeof H);
17 dfs(0, 0); // asumimos que la raiz tiene indice 0
18 }
```



Por ejemplo, si llamamos a `dfs(0, 0)` sobre el árbol de la figura 9.8, tendremos<sup>12</sup>:

| Índice | 0 | 1 | 2 | 3 | 4 | 5 | 6  | 7  | 8  | 9   | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|--------|---|---|---|---|---|---|----|----|----|-----|----|----|----|----|----|----|----|----|----|
| H      | 0 | 1 | 2 | 4 | 5 | 7 | 10 | 13 | 14 | 16  | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |    |
| E      | 0 | 1 | 2 | 1 | 3 | 4 | 3  | 5  | 3  | (1) | 6  | 1  | 0  | 7  | 8  | 7  | 9  | 7  | 0  |
| L      | 0 | 1 | 2 | 1 | 2 | 3 | 2  | 3  | 2  | 1   | 2  | 1  | 0  | 1  | 2  | 1  | 2  | 1  | 0  |

Tabla 9.1: Reducción de LCA a RMQ

Una vez que tenemos los tres *arrays* con los que trabajar, podemos resolver el LCA utilizando RMQ. Asumimos que  $H[u] < H[v]$  o, en caso contrario, intercambiamos  $u$  y  $v$ . Aquí vemos que

<sup>12</sup>En la sección 4.2.1,  $H$  se denominaba `dfs_num`.

el problema se reduce a encontrar el vértice de menor profundidad en  $E[H[u]..H[v]]$ . Así, la solución viene dada por  $LCA(u, v) = E[\text{RMQ}(H[u], H[v])]$ , donde  $\text{RMQ}(i, j)$  se ejecuta sobre el *array* L. Si utilizamos la estructura de tabla dispersa de la sección 9.33, el *array* que habrá que procesar en la fase de construcción es L.

Por ejemplo, si queremos obtener el  $LCA(4, 6)$  del árbol de la figura 9.8, calcularemos  $H[4] = 5$  y  $H[6] = 10$ , y buscaremos el vértice de profundidad mínima en  $E[5..10]$ . Llamar a  $\text{RMQ}(5, 10)$  sobre el *array* L (ver las casillas subrayadas de la fila L en la tabla 9.1) devuelve 9. El valor de  $E[9] = 1$  (ver la casilla en cursiva de la fila E en la misma tabla), por lo que informaremos de que 1 es la respuesta de  $LCA(4, 6)$ .

### Ejercicios de programación

1. UVa 10938 - Flea circus (problema básico de ancestro común mínimo, según acabamos de ver)
2. UVa 12238 - Ants Colony (similar a UVa 10938)

## 9.19 Construcción de un cuadrado mágico (de tamaño impar)

### Enunciado del problema

Un cuadrado mágico es un *array* bidimensional de tamaño  $n \times n$ , que contiene enteros en el rango  $[1..n^2]$ , con la propiedad ‘mágica’: la suma de los enteros de cada fila, columna y diagonal es la misma. Por ejemplo, para  $n = 5$ , podemos construir el siguiente cuadrado mágico, donde la suma de las filas, columnas y diagonales es igual a 65.

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Nuestra tarea consiste en construir un cuadrado mágico dado su tamaño  $n$ , si este es impar.

### Soluciones

Si no conocemos la solución, podemos utilizar *backtracking* recursivo estándar para intentar colocar cada entero  $\in [1..n^2]$ , de uno en uno. Esta solución de búsqueda completa es demasiado lenta para un  $n$  grande.

Por suerte, existe una ‘estrategia de construcción’ interesante para cuadrados mágicos de tamaño impar, llamada ‘método siamés (De la Loubère)’. Comenzamos con un *array* cuadrado bidimensional. Inicialmente, colocamos el entero 1 en el centro de la primera fila. Después, nos desplazamos al noreste, saltando al extremo opuesto cuando sea necesario. Si la nueva celda está vacía, le añadimos el siguiente entero. Si está ocupada, nos movemos una fila hacia abajo y seguimos hacia el noreste. La figura 9.9 muestra este método siamés. Deducir esta estrategia,

si no se conoce previamente el problema no resulta evidente (aunque tampoco es imposible mediante la atenta observación de varios cuadrados mágicos de tamaño impar).

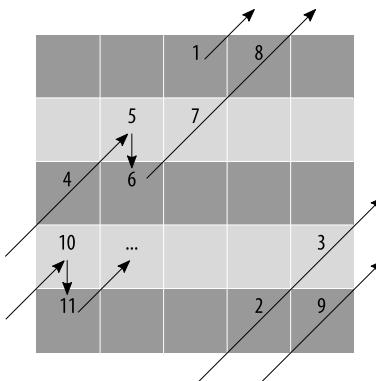


Figura 9.9: Estrategia de construcción de un cuadrado mágico para un  $n$  impar

Hay otros casos especiales para cuadrados mágicos de diferentes tamaños. No es necesario aprenderlos todos, ya que, probablemente, nunca aparecerán en un concurso de programación. Sin embargo, es fácil imaginar que los concursantes que los conozcan, tendrían una enorme ventaja en caso de enfrentarse a un problema que los incluya.

### Ejercicios de programación

1. [UVa 01266 - Magic Square](#)\* (sigue la estrategia que hemos visto)

## 9.20 Multiplicación de cadenas de matrices

### Enunciado del problema

Dadas  $n$  matrices,  $A_1, A_2, \dots, A_n$ , donde cada  $A_i$  tiene un tamaño  $P_{i-1} \times P_i$ , mostrar el producto correctamente expresado de  $A_1 \times A_2 \times \dots \times A_n$ , que minimice el número de multiplicaciones escalares. Se dice que un producto de matrices está correctamente expresado si cumple una de las siguientes condiciones:

1. Es una sola matriz.
2. Es el producto de 2 productos correctamente expresados, indicado entre paréntesis.

Por ejemplo, dado el *array* de 3 matrices  $P = \{10, 100, 5, 50\}$  (que implica que la matriz  $A_1$  tiene tamaño  $10 \times 100$ ,  $A_2$  tiene tamaño  $100 \times 5$  y  $A_3$  tiene  $5 \times 50$ ). Podemos expresar correctamente estas tres matrices de dos maneras:

1.  $(A_1 \times (A_2 \times A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$  multiplicaciones escalares.

2.  $((A_1 \times A_2) \times A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$  multiplicaciones escalares.

En el ejemplo anterior, podemos ver que el coste de multiplicar estas 3 matrices, en términos del número de multiplicaciones escalares, depende de la forma en que las expresemos. Sin embargo, una comprobación exhaustiva de todas las posibles expresiones es demasiado lenta, ya que el número de posibilidades es enorme (existen  $\text{Cat}(n-1)$  expresiones correctas de  $n$  matrices, ver la sección 5.4.3).

## Multiplicación de matrices

Podemos multiplicar dos matrices  $a$ , de tamaño  $p \times q$ , y  $b$ , de tamaño  $q \times r$ , si el número de columnas de  $a$  y  $b$  son iguales (la dimensión interna es coincidente). El resultado de esta multiplicación es la matriz  $c$ , de tamaño  $p \times r$ . El coste de la operación es de  $O(p \times q \times r)$  multiplicaciones, y se puede implementar con el siguiente código de C++:

```

1 #define MAX_N 10 // aumentar/reducir este valor según sea necesario
2 struct Matrix { int mat[MAX_N][MAX_N]; };
3
4 Matrix matMul(Matrix a, Matrix b, int p, int q, int r); // O(pqr)
5 Matrix c; int i, j, k;
6 for (i = 0; i < p; i++)
7 for (j = 0; j < r; j++)
8 for (c.mat[i][j] = k = 0; k < q; k++)
9 c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
10 return c;

```

Por ejemplo, si tenemos las siguientes matrices  $a$  de  $2 \times 3$ , y  $b$  de  $3 \times 1$ , necesitaremos  $2 \times 3 \times 1 = 6$  multiplicaciones escalares.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{bmatrix} = \begin{bmatrix} c_{1,1} = a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + a_{1,3} \times b_{3,1} \\ c_{2,1} = a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} + a_{2,3} \times b_{3,1} \end{bmatrix}$$

Cuando las dos matrices son cuadradas, de tamaño  $n \times n$ , esta multiplicación se ejecuta en  $O(n^3)$  (ver la sección 9.21, que es muy similar).

## Soluciones

El problema de la multiplicación de cadenas de matrices es uno de los ejemplos clásicos que se utilizan para ilustrar la técnica de la programación dinámica (DP). Como ya hemos tratado la DP en detalle en la sección 3.5, solo comentaremos las ideas clave. Para este problema, en realidad, no multiplicamos las matrices como hemos visto en la subsección anterior. Nos basta con encontrar la expresión correcta óptima de las  $n$  matrices.

Digamos que  $\text{coste}(i, j)$ , donde  $i < j$ , indica el número de multiplicaciones escalares necesarias para multiplicar las matrices  $A_i \times A_{i+1} \times \dots \times A_j$ . En este caso, tenemos las siguientes recurrencias de búsqueda completa:

1.  $\text{coste}(i, j) = 0$  si  $i = j$ .
2.  $\text{coste}(i, j) = \min(\text{coste}(i, k) + \text{coste}(k + 1, j) + P_{i-1} \times P_k \times P_j), \forall k \in [i \dots j - 1]$ .

El coste óptimo se almacena en  $\text{coste}(1, n)$ . Hay  $O(n^2)$  pares diferentes del subproblema  $(i, j)$ . Por lo tanto, necesitamos una tabla de DP de tamaño  $O(n^2)$ . Calcular cada subproblema necesita hasta  $O(n)$ . Así, la complejidad de tiempo de esta solución de DP, para la multiplicación de cadenas de matrices, es de  $O(n^3)$ .

### Ejercicios de programación

1. [UVa 00348 - Optimal Array Mult ... \\*](#) (DP;  $s(i, j)$ ; mostrar también la solución óptima; la secuencia de multiplicación de matrices óptima no es única; por ejemplo, pensemos en que todas las matrices sean cuadradas)

## 9.21 Potencia de matrices

### Algunas definiciones y ejemplos de uso

En esta sección, trataremos el caso especial de una matriz<sup>13</sup>: la *matriz cuadrada*<sup>14</sup>. Para ser más precisos, trataremos una operación especial: las *potencias de una matriz cuadrada*. En términos matemáticos,  $M^0 = I$  y  $M^p = \prod_{i=1}^p M$ .  $I$  es la matriz de *identidad*<sup>15</sup> y  $p$  es la potencia de la matriz cuadrada  $M$ . Si podemos realizar esta operación en  $O(n^3 \log p)$ , que es el objetivo de esta sección, podremos resolver problemas interesantes de los concursos de programación, como:

- Calcular un *único*<sup>16</sup> número de Fibonacci  $\text{fib}(p)$ , en tiempo  $O(\log p)$ , en vez de  $O(p)$ . Imaginemos que tenemos  $p = 2^{30}$ , la solución  $O(p)$  tendrá un veredicto de TLE, pero la solución en  $\log_2(p)$  solo necesita 30 pasos. Se consigue mediante la siguiente igualdad:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} \text{fib}(p+1) & \underline{\text{fib}(p)} \\ \underline{\text{fib}(p)} & \text{fib}(p-1) \end{bmatrix}$$

Por ejemplo, para calcular  $\text{fib}(11)$ , solo tenemos que multiplicar la matriz de Fibonacci 11 veces, es decir, elevarla a la undécima potencia. La respuesta está en la segunda diagonal de la matriz.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} = \begin{bmatrix} 144 & \underline{\textbf{89}} \\ \underline{\textbf{89}} & 55 \end{bmatrix} = \begin{bmatrix} \text{fib}(12) & \underline{\text{fib}(11)} \\ \underline{\text{fib}(11)} & \text{fib}(10) \end{bmatrix}$$

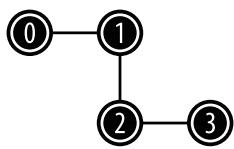
<sup>13</sup>Una matriz es un *array* rectangular bidimensional. La matriz de tamaño  $m \times n$ , tiene  $m$  filas y  $n$  columnas. Los elementos de la matriz vienen, normalmente, determinados por el nombre de la matriz con dos subíndices.

<sup>14</sup>Una matriz cuadrada es aquella que tiene el mismo número de filas y columnas, es decir, su tamaño es  $n \times n$ .

<sup>15</sup>La matriz de identidad es aquella compuesta únicamente de ceros, salvo sus diagonales, compuestas de unos.

<sup>16</sup>Si necesitamos  $\text{fib}(n)$  para todos los  $n \in [0 \dots n]$ , es mejor utilizar la solución de DP en  $O(n)$ .

- Calcular el número de caminos de longitud  $L$  de un grafo almacenado en una matriz de adyacencia, que es una matriz cuadrada, en  $O(n^3 \log L)$ . Por ejemplo, veamos el pequeño grafo de tamaño  $n = 4$ , almacenado en la siguiente matriz de adyacencia  $M$ . La entrada  $M[0][1]$  muestra los diferentes caminos, de distintas longitudes, entre los vértices 0 y 1, después elevar  $M$  a la  $L$ -ésima potencia.



- $0 \rightarrow 1$  con longitud 1:  $0 \rightarrow 1$  (solo 1 camino)
- $0 \rightarrow 1$  con longitud 2: imposible
- $0 \rightarrow 1$  con longitud 3:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 1$  (y  $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ )
- $0 \rightarrow 1$  con longitud 4: imposible
- $0 \rightarrow 1$  con longitud 5:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 1$  (y 4 más)

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} M^2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} M^3 = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{bmatrix} M^5 = \begin{bmatrix} 0 & 5 & 0 & 3 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 5 \\ 3 & 0 & 5 & 0 \end{bmatrix}$$

- Acelerar *algunos* problemas de DP, como veremos más adelante en esta sección

### La idea de la exponenciación (potencia) eficiente

A efectos de esta sección, asumimos que las funciones de la biblioteca del lenguaje, como `pow(base, b)`, y otras relacionadas que también elevan un número *base* a una cierta potencia entera *p*, no existen. A partir de aquí, si realizamos una exponenciación ‘según está definida’, como se muestra a continuación, tendremos una solución  $O(p)$ . Resulta totalmente ineficiente, sobre todo si *p* es grande<sup>17</sup>.

```

1 int normalExp(int base, int p) { // para simplificar, usamos el tipo int
2 int ans = 1; // también asumimos que ans no será mayor de 2^31 - 1
3 for (int i = 0; i < p; i++) ans *= base; // esto es O(p)
4 return ans; }
```

Hay una solución mejor, que utiliza el principio de divide y vencerás. Podemos expresar  $A^p$ :

- $A^0 = 1$  (caso base).
- $A^1 = A$  (otro caso base, pero mira el **ejercicio 9.21.1\***).
- $A^p = A^{p-1} \times A$ , si *p* es impar.
- $A^p = (A^{p/2})^2$ , si *p* es par.

Como esta técnica divide constantemente el valor de *p* entre 2, se ejecuta en  $O(\log p)$ .

<sup>17</sup>Si encuentras un tamaño de entrada ‘gigantesco’ en un concurso de programación, como  $1000M$ , *normalmente* el autor del problema estará buscando una solución logarítmica. Piensa en que  $\log_2(1000M) \approx \log_2(2^{30})$ , solo es 30!.

Por ejemplo, por definición:  $2^9 = 2 \times 2 \approx O(p)$  multiplicaciones. Pero con divide y vencerás:  $2^9 = 2^8 \times 2 = (2^4)^2 \times 2 = ((2^2)^2)^2 \times 2 \approx O(\log p)$  multiplicaciones.

A continuación, incluimos una implementación recursiva típica de esta exponentiación de divide y vencerás, omitiendo aquellos casos en los que la respuesta supere la capacidad de un entero de 32 bits:

```
1 int fastExp(int base, int p) { // O(log p)
2 if (p == 0) return 1;
3 else if (p == 1) return base; // ver el ejercicio a continuación
4 else {
5 int res = fastExp(base, p / 2); res *= res;
6 if (p % 2 == 1) res *= base;
7 return res; } }
```

### Ejercicio 9.21.1\*

¿Realmente necesitamos el segundo caso base, `if (p == 1) return base;`?

### Ejercicio 9.21.2\*

Elevar un número a una cierta potencia (entera), puede provocar un desbordamiento con facilidad. Una variante interesante consiste en calcular  $base^p \pmod{m}$ . Convierte la función `fastExp(base, p)` en `modPow(base, p, m)` (ver también las secciones 5.3.2 y 5.5.8).

### Ejercicio 9.21.3\*

Transforma la implementación recursiva de divide y vencerás en una versión iterativa. Consejo: sigue leyendo esta sección.

## Exponenciación de matrices cuadradas (potencia de matrices)

Podemos utilizar la misma técnica eficiente de exponentiación en  $O(\log p)$  que hemos visto, para realizar exponentiaciones de matrices cuadradas (potencia de matrices) en  $O(n^3 \log p)$ , porque cada multiplicación de matrices<sup>18</sup> consume  $O(n^3)$ . La implementación *iterativa* (en contraste con la recursiva que hemos visto) es la siguiente:

<sup>18</sup>Existe un algoritmo más rápido, aunque más complejo, de multiplicación de matrices, el algoritmo de Strassen en  $O(2.8074)$ . Normalmente no lo utilizaremos en concursos de programación. La multiplicación de dos matrices de Fibonacci, que hemos visto en la sección 9.21, solo necesita  $2^3 = 8$  multiplicaciones, ya que  $n = 2$ . Lo podemos tratar como  $O(1)$ , por lo que podemos calcular  $fib(p)$  en  $O(\log p)$ .

```

1 #define MAX_N 2 //matriz Fibonacci, aumentar/reducir el valor según el caso
2 struct Matrix { int mat[MAX_N][MAX_N]; }; // devuelve un array bidimensional
3
4 Matrix matMul(Matrix a, Matrix b) { // O(n^3)
5 Matrix ans; int i, j, k;
6 for (i = 0; i < MAX_N; i++)
7 for (j = 0; j < MAX_N; j++)
8 for (ans.mat[i][j] = k = 0; k < MAX_N; k++) // si es necesario, usar
9 ans.mat[i][j] += a.mat[i][k] * b.mat[k][j]; // aritmética modular
10 return ans; }
11
12 Matrix matPow(Matrix base, int p) { // O(n^3 log p)
13 Matrix ans; int i, j;
14 for (i = 0; i < MAX_N; i++) for (j = 0; j < MAX_N; j++)
15 ans.mat[i][j] = (i == j); // preparar la matriz de identidad
16 while(p) { // versión iterativa de exponentiación con divide y vencerás
17 if (p & 1) ans = matMul(ans, base); // si p es impar (último bit activo)
18 base = matMul(base, base); // elevar la base al cuadrado
19 p >= 1; // dividir p por 2
20 }
21 return ans; }

```



UVa10229.cpp



UVa10229.java

## Aceleración de DP con potencia de matrices

Ahora, trataremos cómo derivar las matrices cuadradas necesarias para dos problemas de DP, y mostraremos que, elevar estas dos matrices a las potencias requeridas, puede acelerar el cálculo de los problemas de DP originales.

Comenzamos con la matriz de Fibonacci de  $2 \times 2$ . Sabemos que  $fib(0) = 0$ ,  $fib(1) = 1$  y, para  $n \geq 2$ , tenemos  $fib(n) = fib(n - 1) + fib(n - 2)$ . Podemos calcular  $fib(n)$  en  $O(n)$ , utilizando programación dinámica, al hallar progresivamente  $fib(n)$ , de uno en uno, desde  $[2..n]$ . Sin embargo, estas transiciones de DP se pueden acelerar reescribiendo la recurrencia de Fibonacci en forma de matriz, como vemos a continuación.

En primer lugar, escribimos dos versiones de la recurrencia de Fibonacci, ya que hay dos términos en la misma:

$$\begin{aligned} fib(n+1) + fib(n) &= fib(n+2) \\ fib(n) + fib(n-1) &= fib(n+1) \end{aligned}$$

Después, representamos la recurrencia como una matriz:

$$\left[ \begin{array}{cc} a & b \\ c & d \end{array} \right] \times \left[ \begin{array}{c} fib(n+1) \\ fib(n) \end{array} \right] = \left[ \begin{array}{c} fib(n+2) \\ fib(n+1) \end{array} \right]$$

Ahora, tenemos  $a \times fib(n+1) + b \times fib(n) = fib(n+2)$  y  $c \times fib(n+1) + d \times fib(n) = fib(n+1)$ . Al expresar la recurrencia de DP de la forma anterior, tenemos una *matriz cuadrada* de  $2 \times 2$ . Los valores apropiados para  $a, b, c$  y  $d$  serán  $1, 1, 1, 0$ , y esta es la matriz de Fibonacci de  $2 \times 2$  que hemos visto antes. Cada multiplicación de la matriz hace avanzar un paso el cálculo por DP del número de Fibonacci. Si multiplicamos esta matriz de Fibonacci de  $2 \times 2$   $p$  veces, haremos avanzar el cálculo de DP del número de Fibonacci  $p$  pasos. Ahora tendremos:

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \dots \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_p \times \begin{bmatrix} fib(n+1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} fib(n+1+p) \\ fib(n+p) \end{bmatrix}$$

Si, por ejemplo, establecemos  $n = 0$  y  $p = 11$ , y utilizamos la potencia de matrices en  $O(\log p)$ , en vez de multiplicar la matriz  $p$  veces, obtendremos los siguientes cálculos:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} \times \begin{bmatrix} fib(1) \\ fib(0) \end{bmatrix} = \begin{bmatrix} 144 & 89 \\ 89 & 55 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 144 \\ \underline{89} \end{bmatrix} = \begin{bmatrix} fib(12) \\ \underline{fib(11)} \end{bmatrix}$$

Esta matriz de Fibonacci también se puede expresar en la forma que hemos visto antes:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} fib(p+1) & fib(p) \\ fib(p) & fib(p-1) \end{bmatrix}$$

Veamos otro ejemplo de cómo derivar la matriz cuadrada, necesaria para otro problema de DP, UVa 10655 - Contemplation! Algebra. El enunciado es muy sencillo: dados los valores de  $p = a + b$ ,  $q = a \times b$  y  $n$ , encontrar el valor de  $a^n + b^n$ .

En primer lugar, trabajamos sobre la fórmula, para poder utilizar  $p = a + b$  y  $q = a \times b$ :

$$a^n + b^n = (a + b) \times (a^{n-1} + b^{n-1}) - (a \times b) \times (a^{n-2} + b^{n-2})$$

A continuación, establecemos  $X_n = a^n + b^n$ , para tener  $X_n = p \times X_{n-1} - q \times X_{n-2}$ . Después, escribimos esta recurrencia dos veces, de la siguiente forma:

$$\begin{aligned} p \times X_{n+1} - q \times X_n &= X_{n+2} \\ p \times X_n - q \times X_{n-1} &= X_{n+1} \end{aligned}$$

Y expresamos la recurrencia en forma de matriz:

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix} = \begin{bmatrix} X_{n+2} \\ X_{n+1} \end{bmatrix}$$

Si elevamos la matriz cuadrada de  $2 \times 2$  a la  $n$ -ésima potencia, en tiempo  $O(\log n)$ , y multiplicamos la matriz cuadrada resultante por  $X_1 = a^1 + b^1 = a + b = p$  y  $X_0 = a^0 + b^0 = 1 + 1 = 2$ , tendremos  $X_{n+1}$  y  $X_n$ . La respuesta que buscábamos es  $X_n$ . Esto es más rápido que el cálculo de DP estándar en  $O(n)$ , para la misma recurrencia.

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} X_1 \\ X_0 \end{bmatrix} = \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix}$$

## Ejercicios de programación

- |                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 10229 - Modular Fibonacci<br>2. <u>UVa 10518 - How Many Calls?</u> <sup>*</sup><br><br>3. <u>UVa 10655 - Contemplation! Algebra</u> <sup>*</sup><br>4. UVa 10870 - Recurrences<br>5. <u>UVa 11486 - Finding Paths in Grid</u> <sup>*</sup><br><br>6. UVa 12470 - Tribonacci | (tratado en esta sección y aritmética modular)<br>(deducir el patrón de las respuestas de un $n$ pequeño: la respuesta es $2 \times fib(n) - 1$ ; entonces usar la solución de UVa 10229)<br>(tratado en esta sección)<br>(empezar formando la matriz requerida; potencia de matrices)<br>(modelar como matriz de adyacencia; elevar la matriz de adyacencia a la $N$ -ésima potencia en $O(\log N)$ , para obtener el número de caminos)<br>(muy similar a UVa 10229; la matriz de $3 \times 3$ es $[0\ 1\ 0;\ 0\ 0\ 1;\ 1\ 1\ 1]$ ; la respuesta está en la matriz[1][1], después de elevarla a la $n$ -ésima potencia, con módulo 1000000009) |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 9.22 Conjunto independiente ponderado máximo

### Enunciado del problema

Dado un grafo  $G$ , con *vértices ponderados*, encontrar su conjunto independiente *ponderado* máximo (MWIS). Un conjunto independiente (IS)<sup>19</sup> es un conjunto de vértices de un grafo, de forma que no haya dos que sean adyacentes. Nuestra tarea consiste en elegir un IS de  $G$ , con el peso máximo total (de vértices). Es un problema difícil en un grafo general. Sin embargo, si el grafo  $G$  es un árbol o un grafo bipartito, tenemos soluciones eficientes.

### Soluciones

#### En un árbol

Si el grafo  $G$  es un árbol<sup>20</sup>, podemos encontrar su MWIS utilizando DP<sup>21</sup>. Digamos que  $C(v, tomado)$  es el MWIS de un subárbol con raíz en  $v$ , si ha sido *tomado* como parte del MWIS. Las recurrencias de búsqueda completa son:

1. Si  $v$  es un vértice hoja:

a)  $C(v, verdadero) = w(v)$

Si se ha tomado la hoja  $v$ , el peso de este subárbol es el peso de  $v$ .

b)  $C(v, falso) = 0$

Si no se ha tomado la hoja  $v$ , entonces el peso de este subárbol es 0.

<sup>19</sup>El complemento del conjunto independiente es la cobertura de vértices.

<sup>20</sup>En la mayoría de los problemas relacionados con árboles, nuestra primera tarea debe ser ‘enraizar el árbol’, en caso de que no tenga raíz. Si el árbol no tiene un vértice establecido como raíz, elige uno cualquiera para que cumpla esa función. Al hacerlo, aparecerán los subproblemas en relación a los subárboles, como en este problema de MWIS en un árbol.

<sup>21</sup>Es posible resolver algunos problemas de optimización en *árboles* mediante programación dinámica. La solución implica, normalmente, pasar información desde/hacia el padre y obtener información desde/hacia el hijo de un árbol con raíz.

2. Si  $v$  es un vértice interno:

$$a) C(v, \text{verdadero}) = w(v) + \sum_{\text{hijo} \in \text{hijos}(v)} C(\text{hijo}, \text{falso})$$

Si se ha tomado la raíz  $v$ , sumamos su peso, pero *no se podrán* tomar hijos de  $v$ .

$$b) C(v, \text{falso}) = \sum_{\text{hijo} \in \text{hijos}(v)} \max(C(\text{hijo}, \text{verdadero}), C(\text{hijo}, \text{falso}))$$

Si no se ha tomado la raíz  $v$ , sus hijos se pueden tomar, o no. Devolvemos el mayor.

La respuesta es  $\max(C(\text{raíz}, 1), C(\text{raíz}, 0))$ , es decir, tomar, o no, la raíz. Esta solución de DP solo necesita  $O(V)$  espacio y  $O(V)$  tiempo.

### En un grafo bipartito

Si el grafo  $G$  es bipartito, tenemos que reducir el problema del MWIS<sup>22</sup> al de flujo máximo. Asignamos el coste de los vértices original (el peso por tomar ese vértice) como la capacidad desde el origen a ese vértice, en el conjunto izquierdo del grafo bipartito, y la capacidad desde ese vértice al desagüe, en el conjunto derecho. Después, le damos capacidad ‘infinita’ a cualquier arista entre los conjuntos izquierdo y derecho. El MWIS de este grafo bipartito es el peso del coste de todos los vértices, menos el valor del flujo máximo del grafo de flujo.

## 9.23 Flujo (máximo) de coste mínimo

### Enunciado del problema

El problema del flujo de coste mínimo, consiste en encontrar el camino *más barato* posible para enviar una cierta cantidad de flujo (normalmente, máxima) en una red de flujo. En este problema, cada arista tiene dos atributos: la capacidad de flujo y el *coste unitario*, para enviar una unidad de flujo por ella. Algunos autores de problemas dedican simplificarlo, estableciendo que la capacidad de las aristas sea una constante entera, y solo varían su coste.

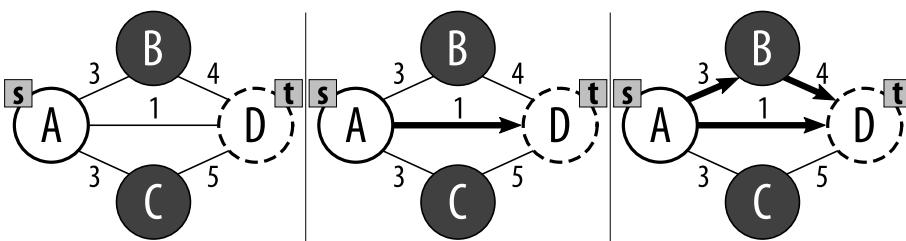


Figura 9.10: Ejemplo del problema de flujo máximo de coste mínimo (MCMF) (UVa 10594 [49])

La parte izquierda de la figura 9.10, muestra una instancia (modificada) del problema UVa 10594. Aquí, cada arista tiene una capacidad uniforme de 10 unidades, y el coste de una unidad aparece en la etiqueta de la arista. Queremos enviar 20 unidades de flujo desde  $A$  hasta  $D$  (el flujo máximo de este grafo de flujo es de 30 unidades), lo que se puede satisfacer enviando 10

<sup>22</sup>El problema del conjunto independiente máximo (MIS) no ponderado en un grafo bipartito, se puede reducir al de emparejamiento bipartito de cardinalidad máxima (MCBM), ver la sección 4.7.4.

unidades  $A \rightarrow D$ , con un coste  $1 \times 10 = 10$  (parte central de la figura 9.10), y otras 10 unidades  $A \rightarrow B \rightarrow D$ , con un coste  $(3 + 4) \times 10 = 70$  (derecha de la figura 9.10). El coste total es de  $10 + 70 = 80$ , que es el mínimo. Si, en su lugar, decidiésemos enviar 20 unidades a través de  $A \rightarrow D$  (10 unidades) y  $A \rightarrow C \rightarrow D$ , tendríramos un coste de  $1 \times 10 + (3 + 5) \times 10 = 10 + 80 = 90$ , lo que resulta mayor que el óptimo de 80.

## Soluciones

El flujo (máximo) de coste mínimo, o MCMF, se puede resolver insertando la BFS en  $O(E)$  (para encontrar el aumento de camino más corto, en términos de número de saltos) del algoritmo de Edmonds Karp, en el algoritmo de Bellman Ford, en  $O(VE)$  (para encontrar el aumento de camino más corto/barato, en términos de *coste de camino*). Necesitamos un algoritmo de búsqueda del camino más corto, que pueda procesar aristas de peso negativo, ya que *podrían aparecer* durante la cancelación de ciertos flujos en una arista inversa (pues tendremos que *restar* el coste tomado por este aumento de camino, al implicar la cancelación de flujo el hecho de que ya no queremos usar esa arista). En la figura 9.5 hay un ejemplo.

La necesidad de un algoritmo de camino más corto, como Bellman Ford, ralentiza la implementación del MCMF al entorno de  $O(V^2E^2)$ , pero esto debería quedar compensado por el autor del problema que, en la mayoría de los casos, utilizará un grafo de entrada con límites pequeños.

## Ejercicios de programación

- |                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                           |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. UVa 10594 - Data Flow<br>2. <b>UVa 10746 - Crime Wave - The Sequel *</b><br>3. UVa 10806 - Dijkstra, Dijkstra<br>4. <b>UVa 10888 - Warehouse *</b><br>5. <b>UVa 11301 - Great Wall of China *</b> | (problema básico de flujo máximo de coste mínimo)<br>(emparejamiento bipartito <i>ponderado</i> mínimo)<br>(enviar 2 flujos arista-disjuntos con coste mínimo)<br>(BFS/SSSP; emparejamiento bipartito <i>ponderado</i> mínimo)<br>(modelado; capacidad de vértices; MCMF) |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 9.24 Cobertura de caminos mínima en un DAG

### Enunciado del problema

El problema de la cobertura de caminos mínima (MPC) en un DAG, se describe como el problema de encontrar el número mínimo de caminos que cubra *cada vértice* del DAG  $G = (V, E)$ . Se dice que un camino  $v_0, v_1, \dots, v_k$  cubre todos los vértices de su recorrido.

Problema de referencia (UVa 1201 - Taxi Cab Scheme): imagina que los vértices de la figura 9.11.A son pasajeros, y dibujamos una arista entre dos vértices  $u$  y  $v$ , si un taxi puede dar servicio al pasajero  $u$  y, después, al  $v$ , *con puntualidad*. La pregunta es: ¿cuál es el número mínimo de taxis que hay que desplegar para atender a *todos* los pasajeros?

La respuesta es dos taxis. En la figura 9.11.D, vemos una posible solución óptima. Un taxi (línea punteada), atiende a los pasajeros 1, 2 y, finalmente, 4. Otro taxi (línea rayada), atiende a los pasajeros 3 y 5. Se ha dado servicio a todos los pasajeros con solo dos taxis. Hay otra solución óptima,  $1 \rightarrow 3 \rightarrow 5$  y  $2 \rightarrow 4$ .

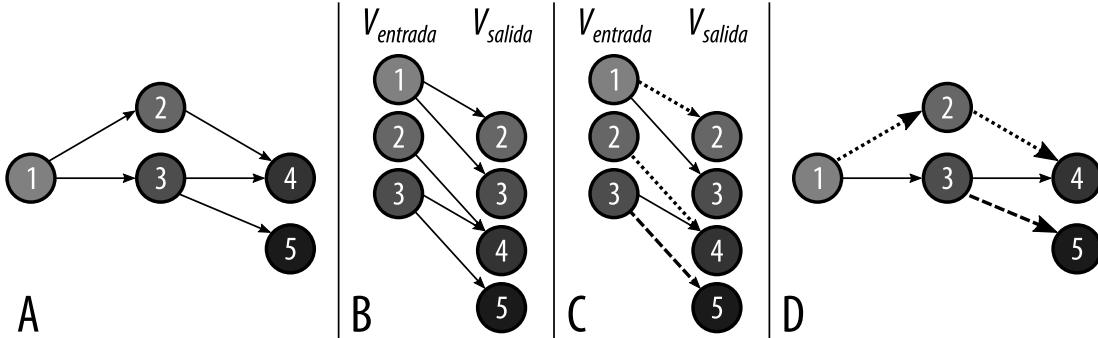


Figura 9.11: Cobertura de caminos mínima en un DAG (de UVa 1201 [49])

## Soluciones

Tenemos una solución polinómica: construir un *grafo bipartito*  $G' = (V_{salida} \cup V_{entrada}, E')$  a partir de  $G$ , donde  $V_{salida} = \{v \in V : v \text{ tiene grado de entrada positivo}\}$  y  $E' = \{(u, v) \in (V_{salida}, V_{entrada}) : (u, v) \in E\}$ . Este grafo  $G'$  es bipartito. Un emparejamiento en el grafo bipartito  $G'$  nos obliga a elegir, al menos, una arista saliente desde cada  $u \in V_{salida}$  (e, igualmente, al menos una arista entrante para cada  $v \in V_{entrada}$ ). El DAG  $G$  tiene, inicialmente,  $n$  vértices, que se pueden cubrir con  $n$  caminos de longitud 0 (los propios vértices). Un emparejamiento entre los vértices  $a$  y  $b$ , utilizando la arista  $(a, b)$ , indica que podemos utilizar un camino menos, ya que la arista  $(a, b) \in E'$ , cubre a ambos vértices en  $a \in V_{salida}$  y  $b \in V_{entrada}$ . Así, si el MCBM de  $G'$  tiene tamaño  $m$ , solo necesitamos  $n - m$  caminos para cubrir cada vértice de  $G$ .

El MCBM de  $G'$ , que es necesario para revolver el MPC de  $G$ , se puede calcular mediante varias soluciones polinómicas. Por ejemplo, flujo máximo, algoritmo de aumento de caminos o algoritmo de Hopcroft Karp (ver la sección 9.10). Como la solución de emparejamientos bipartitos se ejecuta en tiempo polinómico, la solución para el MPC en un DAG también lo hace. El MPC en un grafo general es NP-complejo.

## Ejercicios de programación

- 1. [UVa 01184 - Air Raid \\*](#) (LA 2696 - Dhaka02; MPC en un DAG  $\approx$  MCBM)
- 2. [UVa 01201 - Taxi Cab Scheme \\*](#) (LA 3126 - NorthwesternEurope04; MPC en un DAG)

## 9.25 Ordenación de tortitas

### Enunciado del problema

La ordenación de tortitas es un problema clásico<sup>23</sup> de las ciencias de la computación, pero se utiliza escasamente. El problema se puede describir así: tenemos una pila de  $N$  tortitas. La

<sup>23</sup>Bill Gates, fundador de Microsoft, solo ha escrito un artículo de investigación en su vida, y trata sobre la ordenación de tortitas [22].

tortita de abajo y la de arriba tienen índices 0 y  $N - 1$ , respectivamente. El tamaño de una tortita viene dado por su diámetro (*un entero*  $\in [1 \dots MAX\_D]$ ). Todas las tortitas de la pila tienen diámetros *diferentes*. Por ejemplo, una pila A de  $N = 5$  tortitas,  $\{3, 8, 7, 6, 10\}$ , se puede visualizar como:

|              |    |  |
|--------------|----|--|
| 4 (arriba)   | 10 |  |
| 3            | 6  |  |
| 2            | 7  |  |
| 1            | 8  |  |
| 0 (abajo)    | 3  |  |
| <hr/> Índice | A  |  |

La tarea consiste en ordenar la pila *de forma descendente*, es decir, la tortita más grande quedará abajo y la más pequeña, arriba. Sin embargo, para hacer que el problema sea más realista, la ordenación de la pila de tortitas solo se puede llevar a cabo mediante una secuencia de ‘volteos’ de tortitas, determinada por la función `volteo(i)`. Un movimiento de `volteo(i)` consiste en la inserción de una espátula entre dos tortitas de la pila (entre los índices  $i$  y  $N - 1$ ) y el volteo (inversión) de las tortitas que quedan sobre la misma (inversión de la subpila  $[i..N - 1]$ ).

Por ejemplo, la pila A se puede transformar en la pila B, mediante `volteo(0)`, es decir, insertando la espátula entre los índices 0 y 4, y volteando las tortitas que quedan en medio. La pila B se puede transformar en la C mediante `volteo(3)`. La C en la D mediante `volteo(1)`. Y así sucesivamente. El objetivo es que la pila quede ordenada de forma *descendente*, es decir, que el estado final sea el de la pila E.

|              |    |   |    |   |    |   |     |     |   |
|--------------|----|---|----|---|----|---|-----|-----|---|
| 4 (arriba)   | 10 | ← | 3  | ← | 8  | ← | 6   | 3   |   |
| 3            | 6  |   | 8  | ← | 3  | 7 | ... | 6   |   |
| 2            | 7  |   | 7  |   | 7  | 3 |     | 7   |   |
| 1            | 8  |   | 6  |   | 6  | ← | 8   | 8   |   |
| 0 (abajo)    | 3  | ← | 10 |   | 10 |   | 10  | 10  |   |
| <hr/> Índice | A  |   | B  |   | C  |   | D   | ... | E |

Para hacer que la tarea resulte más emocionante, tienes que calcular el *número mínimo de operaciones volteo(i)* necesarias para que la pila de  $N$  tortitas quede ordenada.

La primera línea de la entrada es un entero  $T$  y, después, hay  $T$  casos de prueba, uno por línea. Cada caso comienza con un entero  $N$ , seguido de  $N$  enteros que describen el contenido inicial de la pila. La salida debe ser un entero, que refleja el número mínimo de operaciones `volteo(i)` necesarias para ordenar la pila.

Límites:  $1 \leq T \leq 100$ ,  $1 \leq N \leq 10$  y  $N \leq MAX\_D \leq 1000000$ .

## Casos de prueba de ejemplo

### Ejemplo de entrada

```
7
4 4 3 2 1
8 8 7 6 5 4 1 2 3
5 5 1 2 4 3
5 555555 111111 222222 444444 333333
8 1000000 999999 999998 999997 999996 999995 999994 999993
5 3 8 7 6 10
10 9 2 10 3 1 6 8 4 7 5
```

### Ejemplo de salida

```
0
1
2
2
0
4
11
```

### Explicación

- La primera pila ya está ordenada de forma descendente.
- La segunda pila se puede ordenar con una llamada a `volteo(5)`.
- La tercera (y, también, la cuarta) pila se puede ordenar llamando a `volteo(3)` y `volteo(1)`: 2 volteos.
- La quinta pila, aunque contiene enteros grandes, ya está ordenada, por lo que no es necesario ningún volteo.
- La sexta pila es, en realidad, la que aparecía como ejemplo en el enunciado del problema. Se puede ordenar utilizando un mínimo de 4 volteos:  
Solución 1: `volteo(0)`, `volteo(1)`, `volteo(2)`, `volteo(1)`: 4 volteos.  
Solución 2: `volteo(1)`, `volteo(2)`, `volteo(1)`, `volteo(0)`: también 4 volteos.
- La séptima pila, con  $N = 10$ , sirve para verificar la velocidad de la solución.

### Soluciones

En primer lugar, debemos realizar una observación sobre el hecho de que el diámetro de las tortitas, en realidad, es irrelevante. Solo tenemos que escribir un código sencillo que ordene los

diámetros (potencialmente enormes) de  $[1..1M]$  y los etiquete como  $[0..N - 1]$ . De esta forma, podemos describir cualquier pila de tortitas como una permutación de  $N$  enteros.

Si solo necesitamos ordenar las tortitas, podemos utilizar un algoritmo voraz, no óptimo, en  $O(2 \times N - 3)$ : volteamos la tortita más grande para ponerla en lo alto de la pila, y después la volteamos abajo. Hacemos lo mismo con la segunda más grande, y así sucesivamente. Si seguimos haciendo esto, lograremos una pila ordenada en  $O(2 \times N - 3)$  pasos, independientemente del estado inicial.

Sin embargo, para lograr el mínimo de operaciones de voldeo, necesitamos ser capaces de modelar este problema como uno de caminos más cortos en un grafo estado-espacio ponderado (sección 8.2.3). El vértice de este grafo estado-espacio es una permutación de  $N$  tortitas. Un vértice está conectado con aristas no ponderadas a otros  $O(N - 1)$  vértices, a través de varias operaciones de voldeo (menos una, ya que voltear la tortita superior no cambia nada). Entonces, podremos usar BFS desde la permutación inicial, para encontrar el camino más corto hasta la permutación de destino (donde está ordenada de forma descendente). Hay un máximo de  $V = O(N!)$  vértices y  $V = O(N! \times (N - 1))$  aristas en este grafo estado-espacio. Por lo tanto, una BFS en  $O(V + E)$  se ejecuta en  $O(N \times N!)$ , por cada caso de prueba, o  $O(T \times N \times N!)$ , para el total de casos. Programar esta BFS ya es una tarea compleja de por sí (secciones 4.4.2 y 8.2.3). Pero esta solución sigue siendo demasiado lenta para el caso de prueba más grande.

Una optimización sencilla, consiste en ejecutar la BFS desde la permutación de destino (ya ordenada) hacia el resto de permutaciones *una sola vez*, para todos los  $N$  posibles en  $[1..10]$ . Esta solución tiene una complejidad de tiempo aproximada de  $O(10 \times N \times N! + T)$ , mucho más rápida que la anterior, pero todavía demasiado lenta para un concurso de programación típico.

Una solución mejor es la técnica de búsqueda, más sofisticada, de ‘encuentro en el medio’ (BFS bidireccional), para reducir el espacio de búsqueda a un nivel manejable (sección 8.2.4). En primer lugar, hacemos un análisis preliminar (o consultamos ‘Pancake Number’, en la página web <http://oeis.org/A058986>) para identificar que, para el caso de prueba más grande, cuando  $N = 10$ , necesitamos *como mucho* 11 volteos para ordenar la pila. Por lo tanto, calculamos previamente la BFS desde la permutación de destino a todas las otras, para todos los  $N \in [1..10]$ , pero nos detenemos al llegar a la profundidad  $\lfloor \frac{11}{2} \rfloor = 5$ . Después, ejecutamos una BFS, para cada caso de prueba, nuevamente desde la permutación inicial, con una profundidad máxima de 5. Si encontramos un vértice común con la BFS desde el destino, calculada previamente, sabremos que la respuesta es la distancia desde la permutación inicial hasta este vértice, más la distancia desde este vértice a la permutación de destino. Si no encontramos ningún vértice común, sabremos que la respuesta será el máximo de 11 volteos. En el caso de prueba más grande, con  $N = 10$  para todos los casos, esta solución tiene una complejidad de tiempo de  $O((10 + T) \times 10^5)$ , lo que la hace viable.

## Ejercicios de programación

### 1. UVa 00120 - Stacks Of Flapjacks \*

(ordenación de tortitas; versión voraz)

2. El problema de ordenación de tortitas descrito en esta sección.

## 9.26 Algoritmo de factorización de enteros rho de Pollard

En la sección 5.5.4, hemos visto que el algoritmo de división por tentativa optimizado se puede utilizar para encontrar los factores primos de enteros hasta  $\approx 9 \times 10^{13}$  (ver el [ejercicio 5.5.4.1](#)), en el *ámbito de un concurso* (es decir, en ‘unos pocos segundos’, en vez de en minutos, horas o días). Pero, ¿qué ocurre si nos dan un entero sin signo de 64 bits (es decir, hasta  $\approx 1 \times 10^{19}$ ) para factorizar?

Se puede lograr una factorización de enteros *más rápida* mediante el algoritmo rho de Pollard [52, 3]. La idea clave de este algoritmo es que dos enteros  $x$  e  $y$  son congruentes módulo  $p$  ( $p$  es uno de los factores de  $n$ , el número que queremos factorizar) con una probabilidad de 0,5, después de que se hayan elegido aleatoriamente ‘unos pocos ( $1,177\sqrt{p}$ ) enteros’.

Los detalles teóricos de este algoritmo son, seguramente, poco relevantes para la programación competitiva. En esta sección, proporcionamos directamente una implementación en C++, que se puede utilizar para manejar enteros compuestos que quepan en un entero sin signo de 64 bits, en el ámbito de un concurso. Sin embargo, el algoritmo rho de Pollard no puede factorizar un entero  $n$ , si este es un primo grande, debido a la forma en que funciona. Para gestionar un caso así, tendremos que implementar una prueba de primalidad rápida (probabilística), como el algoritmo de Miller Rabin (ver el [ejercicio 5.3.2.4\\*](#)).

```
1 #define abs_val(a) (((a)>0)?(a):- (a))
2 typedef long long ll;
3
4 ll mulmod(ll a, ll b, ll c) {//devuelve (a * b) % c, y evita desbordamiento
5 ll x = 0, y = a % c;
6 while (b > 0) {
7 if (b % 2 == 1) x = (x + y) % c;
8 y = (y * 2) % c;
9 b /= 2;
10 }
11 return x % c;
12 }
13
14 ll gcd(ll a,ll b) { return !b ? a : gcd(b, a % b); } // GCD estándar
15
16 ll pollard_rho(ll n) {
17 int i = 0, k = 2;
18 ll x = 3, y = 3; // semilla aleatoria = 3, otros valores posibles
19 while (1) {
20 i++;
21 x = (mulmod(x, x, n) + n - 1) % n; // función generadora
22 ll d = gcd(abs_val(y - x), n); // la función clave
23 if (d != 1 && d != n) return d; // encontrado un factor no trivial
24 if (i == k) y = x, k *= 2;
25 }
26
27 int main() {
```

```

28 ll n = 2063512844981574047LL; // asumimos que n no es un primo grande
29 ll ans = pollard_rho(n); // dividir n en dos factores no triviales
30 if (ans > n / ans) ans = n / ans; // hacer que ans sea el factor pequeño
31 printf("%lld %lld\n", ans, n / ans); // debería ser: 1112041493 1855607779
32 } // return 0;

```

También podemos implementar el algoritmo rho de Pollard en Java, y utilizar la función `isProbablePrime` de la clase `BigInteger`. De esta forma, podemos aceptar un  $n$  mayor de  $2^{64}-1$ , como 17798655664295576020099, que es  $\approx 2^{74}$ , y factorizarlo en  $143054969437 \times 124418296927$ . Sin embargo, el tiempo de ejecución del algoritmo rho aumenta con  $n$  grandes. El hecho de que la factorización de enteros sea una tarea muy compleja, sigue siendo la base fundamental de la criptografía moderna.

Es una buena idea probar la implementación completa del algoritmo rho de Pollard (esto es, incluir el algoritmo de prueba de primalidad probabilístico y otros pequeños detalles), para resolver los dos siguientes ejercicios de programación.



Pollardsrho.cpp



Pollardsrho.java

## Ejercicios de programación

1. [UVa 11476 - Factoring Large\(t\) ... \\*](#) (ver el contenido de la sección)
2. POJ 1811 - Prime Test (ver <http://poj.org/problem?id=1811>)

## 9.27 Calculadora posfija y conversión

### Expresiones algebraicas

Hay tres tipos de expresiones algebraicas: infijas (la forma en que los humanos las escribimos de forma natural), prefijas<sup>24</sup> (notación polaca) y posfijas (notación polaca inversa). En las expresiones infijas/prefijas/posfijas, el operador se coloca en el medio, antes o después de dos operandos, respectivamente. La tabla 9.2 muestra tres expresiones infijas, sus correspondientes prefijas/posfijas y sus valores.

| Infija                      | Prefija                 | Posfija                 | Valor |
|-----------------------------|-------------------------|-------------------------|-------|
| $2 + 6 * 3$                 | $+ 2 * 6 3$             | $2 6 3 * +$             | 20    |
| $(2 + 6) * 3$               | $* + 2 6 3$             | $2 6 + 3 *$             | 24    |
| $4 * (1 + 2 * (9 / 3) - 5)$ | $* 4 - + 1 * 2 / 9 3 5$ | $4 1 2 9 3 / * + 5 - *$ | 8     |

Tabla 9.2: Ejemplos de expresiones infijas, prefijas y posfijas

<sup>24</sup>Un lenguaje de programación que utiliza este tipo es Scheme.

## Calculadora posfija

Las expresiones posfijas son más eficientes, en términos de cálculo, que las infijas. En primer lugar, hacen innecesario el uso de paréntesis (a veces complejos), pues las reglas de precedencia ya están integradas en la propia expresión. En segundo lugar, podemos calcular resultados parciales en el momento en el que se especifica un operador. Estas dos características no se encuentran en las expresiones infijas.

Una expresión posfija se puede calcular en  $O(n)$ , utilizando el algoritmo de la calculadora posfija. Inicialmente, empezamos con una pila vacía. Leemos la expresión de izquierda a derecha, con un elemento cada vez. Si encontramos un operando, lo añadimos a la pila. Si encontramos un operador, extraemos los dos elementos superiores de la pila, realizamos la operación solicitada, y volvemos a añadir el resultado a la pila. Por último, cuando se hayan leído todos los elementos, devolvemos el elemento superior (el único, en realidad) de la pila, como respuesta final.

Como cada uno de los  $n$  elementos solo se procesa una vez, y todas las operaciones en una pila tienen un coste  $O(1)$ , el algoritmo de la calculadora posfija se ejecuta en  $O(n)$ .

La tabla 9.3 muestra un ejemplo de un cálculo posfijo.

| Posfija     | Pila (abajo a arriba) | Comentarios                                   |
|-------------|-----------------------|-----------------------------------------------|
| 41293/*+5-* | 41293                 | Los primeros cinco elementos son operandos    |
| 41293/*+5-* | 4123                  | Tomamos 3 y 9, calculamos 9 / 3, insertamos 3 |
| 41293/*+5-* | 416                   | Tomamos 3 y 2, calculamos 2 * 3, insertamos 6 |
| 41293/*+5-* | 47                    | Tomamos 6 y 1, calculamos 1 + 6, insertamos 7 |
| 41293/*+5-* | 475                   | Un operando                                   |
| 41293/*+5-* | 475                   | Tomamos 5 y 7, calculamos 7 - 5, insertamos 2 |
| 41293/*+5-* | 42                    | Tomamos 2 y 4, calculamos 4 * 2, insertamos 8 |
| 41293/*+5-* | 8                     | Devolvemos 8 como respuesta                   |

Tabla 9.3: Ejemplo de cálculo posfijo

### Ejercicio 9.27.1\*

¿Qué ocurre si recibimos expresiones prefijas? ¿Cómo las evaluamos en  $O(n)$ ?

## Conversión infija a posfija

Sabiendo que las expresiones posfijas son más eficientes, en términos de cálculo, que las infijas, muchos compiladores convierten las expresiones infijas del código fuente (son las que utilizan la mayoría de los lenguajes de programación) en posfijas. Para utilizar la calculadora posfija que hemos visto antes, necesitamos ser capaces de convertir expresiones infijas en posfijas con eficiencia. Un algoritmo posible es el llamado ‘patio de maniobras’, inventado por Edsger Dijkstra (el inventor del algoritmo de Dijkstra, sección 4.4.3).

El algoritmo del patio de maniobras tiene un cierto parecido al de emparejamiento de paréntesis (sección 9.4) y a la calculadora posfija. También utiliza una pila, inicialmente vacía. Leemos la expresión de izquierda a derecha, con un elemento cada vez. Si encontramos un operando, lo mostramos inmediatamente. Si encontramos un paréntesis de apertura, lo insertamos en

la pila. Si encontramos un paréntesis de cierre, extraeremos los elementos superiores de la pila, hasta encontrar un paréntesis de apertura (pero no lo mostramos). Si encontramos un operador, seguimos mostrando la salida, extrayendo el elemento superior de la pila, si tiene igual o mayor precedencia que el operador, o hasta que encontremos un paréntesis de apertura, después insertamos el operador en la pila. Al final, seguiremos mostrando la salida y extrayendo el elemento superior de la pila, hasta que esta quede vacía.

Como cada uno de los  $n$  elementos se procesa una sola vez, y todas las operaciones en una pila tienen un coste  $O(1)$ , el algoritmo del patio de maniobras se ejecuta en  $O(n)$ .

La tabla 9.4 muestra un ejemplo de ejecución del algoritmo del patio de maniobras.

| Infija                      | Pila      | Posfija          | Comentarios              |
|-----------------------------|-----------|------------------|--------------------------|
| $4 * (1 + 2 * (9 / 3) - 5)$ |           | 4                | Salida inmediata         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | *         | 4                | Insertar en la pila      |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (       | 4                | Insertar en la pila      |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (       | 41               | Salida inmediata         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+      | 41               | Insertar en la pila      |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+      | 412              | Salida inmediata         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+ *    | 412              | Insertar en la pila      |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+ * (  | 412              | Insertar en la pila      |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+ * (  | 4129             | Salida inmediata         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+ */   | 4129             | Insertar en la pila      |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+ */ ( | 41293            | Salida inmediata         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (+ *    | 41293 /          | Mostrar solo '/'         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (-      | 41293 /* +       | Mostrar '*', después '+' |
| $4 * (1 + 2 * (9 / 3) - 5)$ | * (-      | 41293 /* + 5     | Salida inmediata         |
| $4 * (1 + 2 * (9 / 3) - 5)$ | *         | 41293 /* + 5 -   | Mostrar solo '-'         |
| $4 * (1 + 2 * (9 / 3) - 5)$ |           | 41293 /* + 5 - * | Vaciar la pila           |

Tabla 9.4: Ejemplo de ejecución del algoritmo del patio de maniobras

### Ejercicios de programación

1. UVa 00727 - Equation \* (problema clásico de conversión de infija a posfija)

## 9.28 Números romanos

### Enunciado del problema

Los números romanos forman un sistema numérico utilizado en la antigua Roma. En realidad, es un sistema numérico decimal, pero utiliza ciertas letras del alfabeto, en vez de los dígitos [0..9] (se describen a continuación), no es posicional y no tiene un símbolo para el 0.

En el sistema de números romanos hay 7 letras básicas, que acompañamos de sus valores decimales: I=1, V=5, X=10, L=50, C=100, D=500 y M=1000. También utilizan los siguientes pares de letras: IV=4, IX=9, XL=40, XC=90, CD=400, CM=900.

Los problemas de programación relativos a números romanos suelen tratar de la conversión hacia y desde números arábigos (el sistema decimal que utilizamos habitualmente). Son problemas que aparecen muy raramente en concursos de programación, y el tipo de conversión se detecta al instante al leer el enunciado del problema.

## Soluciones

En esta sección, proporcionamos una biblioteca de conversión que hemos utilizado para resolver varios problemas de programación que implican números romanos. Aunque se podría deducir este código con facilidad, al menos no será necesario depurarlo<sup>25</sup> si ya lo tienes entre tus notas.

```
1 void AtoR(int A) {
2 map<int, string> cvt;
3 cvt[1000] = "M"; cvt[900] = "CM"; cvt[500] = "D"; cvt[400] = "CD";
4 cvt[100] = "C"; cvt[90] = "XC"; cvt[50] = "L"; cvt[40] = "XL";
5 cvt[10] = "X"; cvt[9] = "IX"; cvt[5] = "V"; cvt[4] = "IV";
6 cvt[1] = "I";
7 // procesar los valores de mayor a menor
8 for (map<int, string>::reverse_iterator i = cvt.rbegin();
9 i != cvt.rend(); i++)
10 while (A >= i->first) {
11 printf("%s", ((string)i->second).c_str());
12 A -= i->first; }
13 printf("\n");
14 }
15
16 void RtoA(char R[]) {
17 map<char, int> RtoA;
18 RtoA['I'] = 1; RtoA['V'] = 5; RtoA['X'] = 10; RtoA['L'] = 50;
19 RtoA['C'] = 100; RtoA['D'] = 500; RtoA['M'] = 1000;
20
21 int value = 0;
22 for (int i = 0; R[i]; i++)
23 if (R[i+1] && RtoA[R[i]] < RtoA[R[i+1]]) { // ver siguiente carácter
24 value += RtoA[R[i + 1]] - RtoA[R[i]]; // por definición
25 i++; } // saltar este carácter
26 else value += RtoA[R[i]];
27 printf("%d\n", value);
28 }
```



UVa11616.cpp



UVa11616.java

<sup>25</sup>Si el problema utiliza un estándar de números romanos diferente, será necesario modificar el código.

- |                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. <a href="#">UVa 00344 - Roman Digititis *</a></li> <li>2. <a href="#">UVa 00759 - The Return of the ...</a></li> <li>3. <a href="#">UVa 11616 - Roman Numerals *</a></li> <li>4. <a href="#">UVa 12397 - Roman Numerals *</a></li> </ol> | (contar cuántos caracteres romanos se utilizan para expresar todos los números del 1 al N)<br>(números romanos y comprobación de validez)<br>(problema de conversión de números romanos)<br>(conversión; cada dígito romano tiene valor) |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 9.29 Problema de selección

### Enunciado del problema

El problema de selección es el problema de encontrar el elemento  $k$ -ésimo más pequeño<sup>26</sup> de un *array* de  $n$  elementos. Otro nombre para el problema de selección es el de estadísticos de orden. De ahí que el elemento mínimo (más pequeño) sea el estadístico de orden primero, el máximo (más grande) el estadístico de orden  $n$ -ésimo, y la mediana el estadístico de orden  $\frac{n}{2}$  (hay 2 medianas si  $n$  es par).

El problema de selección se utiliza como ejemplo de referencia en la introducción del capítulo 3. Es esta sección, trataremos con más detalle el problema, sus variantes y sus distintas soluciones.

### Soluciones

**Casos especiales:**  $k = 1$  y  $k = n$

Buscar los elementos mínimo ( $k = 1$ ) o máximo ( $k = n$ ) de un *array* arbitrario se puede hacer con  $\Omega(n - 1)$  comparaciones: establecemos el primer elemento como respuesta temporal  $y$ , después, comparamos esta respuesta con el resto de  $n - 1$  elementos, de uno en uno, y mantenemos el más pequeño (o el más grande, según cuál estemos buscando). Por último, informamos de la respuesta.  $\Omega(n - 1)$  es el límite inferior, es decir, no podemos hacerlo mejor. Aunque este problema es sencillo para  $k = 1$  o  $k = n$ , encontrar el resto de estadísticos de orden, forma general del problema de selección, es más complicado.

### Algoritmo en $O(n^2)$ , datos estáticos

Un algoritmo ingenuo para encontrar el elemento más pequeño  $k$ -ésimo es el siguiente: encontrar el elemento más pequeño, ‘descartarlo’ (por ejemplo, estableciéndolo a un ‘valor más grande de relleno’), y repetir el proceso  $k$  veces. Cuando  $k$  se acerca a 1 (o a  $n$ ), este algoritmo en  $O(kn)$  se puede seguir tratando como  $O(n)$ , es decir, consideraremos  $k$  una ‘constante pequeña’. Sin embargo, el peor caso es cuando tenemos que encontrar la mediana ( $k = \frac{n}{2}$ ), donde este algoritmo se ejecuta en  $O(\frac{n}{2} \times n) = O(n^2)$ .

---

<sup>26</sup>Encontrar el elemento  $k$ -ésimo más grande es equivalente a encontrar el  $(n-k+1)$ -ésimo más pequeño.

### Algoritmo en $O(n \log n)$ , datos estáticos

Un algoritmo mejor consiste en ordenar primero (dicho de otra forma, procesar previamente) el *array* en  $O(n \log n)$ . Una vez hecho esto, podemos encontrar el elemento más pequeño  $k$ -ésimo en  $O(1)$ , devolviendo el contenido del índice  $k - 1$  (indexación desde 0) del *array* ordenado. La parte principal de este algoritmo es la fase de ordenación. Asumiendo que tengamos un buen algoritmo de ordenación, en  $O(n \log n)$ , el tiempo de ejecución total será de  $O(n \log n)$ .

### Algoritmo con expectativa de $O(n)$ , datos estáticos

Un algoritmo, incluso mejor, para el problema de selección, lo encontramos aplicando el paradigma de divide y vencerás. La idea clave surge de utilizar el algoritmo de partición en  $O(n)$  (la versión aleatorizada) de la ordenación *quick*, como subrutina.

Un algoritmo de partición aleatorizada `RandomizedPartition(A, l, r)`, partitiona un rango dado  $[l..r]$  del *array*  $A$ , en torno a un pivote (aleatorio). El pivote  $A[p]$  es un elemento de  $A$ , donde  $p \in [l..r]$ . Despues de la partición, se coloca a todos los elementos  $\leq A[p]$  antes del pivote, y a todos los elementos  $> A[p]$ , despues. Se devuelve el índice final  $q$  del pivote. Este algoritmo de partición aleatorizada se puede ejecutar en  $O(n)$ .

Despues de realizar  $q = \text{RandomizedPartition}(A, 0, n-1)$ , se colocará a todos los elementos  $\leq A[q]$  antes del pivote y, por ello,  $A[q]$  estará en su estadístico de orden correcto, que es  $q + 1$ . Despues, solo hay 3 posibilidades:

1.  $q + 1 = k$ ,  $A[q]$  es la respuesta deseada. Devolvemos este valor y detenemos la ejecución.
2.  $q+1 > k$ , la respuesta deseada está dentro de la partición izquierda, es decir, en  $A[0..q-1]$ .
3.  $q+1 < k$ , la respuesta deseada está dentro de la partición derecha, es decir, en  $A[q+1..n-1]$ .

Este proceso se puede repetir recursivamente en un rango más pequeño del espacio de búsqueda, hasta que encontremos la respuesta que necesitamos. A continuación, mostramos un fragmento de código C++ que implementa este algoritmo:

```
1 int RandomizedSelect(int A[], int l, int r, int k) {
2 if (l == r) return A[l];
3 int q = RandomizedPartition(A, l, r);
4 if (q+1 == k) return A[q];
5 else if (q+1 > k) return RandomizedSelect(A, l, q-1, k);
6 else return RandomizedSelect(A, q+1, r, k);
7 }
```

Este algoritmo `RandomizedSelect` se ejecuta con una expectativa de tiempo de  $O(n)$ , y es muy improbable que llegue a su peor caso de  $O(n^2)$ , ya que utiliza un pivote aleatorio en cada paso. El análisis completo incluye probabilidad y expectativa de valores. El lector interesado puede encontrarlo en otras fuentes, como [7].

Un análisis simplificado, pero no riguroso, consiste en asumir que `RandomizedSelect` divide el *array* en dos a cada paso, y que  $n$  es una potencia de dos. Por ello, ejecuta `RandomizedPartition` en  $O(n)$  en la primera ocasión, en  $O(\frac{n}{2})$  en la segunda, en  $O(\frac{n}{4})$  en la tercera y, finalmente, en

$O(1)$  en la iteración  $1 + \log_2 n$ . El coste de RandomizedSelect viene determinado, principalmente, por el coste de RandomizedPartition, ya que el resto de pasos de RandomizedSelect son  $O(1)$ . Por ello, el coste total es de  $O(n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{n}) = O(n \times (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{n})) \leq O(2n) = O(n)$ .

### Solución de biblioteca para el algoritmo con expectativa de $O(n)$ , datos estáticos

La STL de C++ incluye la función `nth_element` en `<algorithm>`. Esta función, implementa el algoritmo con expectativa de  $O(n)$  que hemos visto. A día de hoy, no somos conscientes de que esta función tenga una equivalente en Java.

### Procesamiento previo en $O(n \log n)$ , algoritmo en $O(\log n)$ , datos dinámicos

Todas las soluciones presentadas hasta el momento asumían que el *array* dado era estático, que no cambiaba entre cada consulta del elemento más pequeño  $k$ -ésimo. Sin embargo, si el contenido del *array* puede variar con frecuencia, es decir, se añade un nuevo elemento, se elimina uno existente o se modifica algún valor, las soluciones mencionadas son poco eficientes.

Cuando los datos subyacentes son dinámicos, necesitamos utilizar un árbol de búsqueda binaria equilibrado (ver la sección 2.3). En primer lugar, insertamos los  $n$  elementos en un BST equilibrado, en tiempo  $O(n \log n)$ . También añadimos información sobre el tamaño de cada subárbol con raíz en cada vértice. De esta forma, podemos encontrar el elemento más pequeño  $k$ -ésimo en tiempo  $O(\log n)$ , mediante la comparación de  $k$  con  $q$ , el tamaño del subárbol izquierdo de la raíz:

1. Si  $q + 1 = k$ , entonces la raíz es la respuesta deseada. Devolvemos este valor y detenemos la ejecución.
2. Si  $q + 1 > k$ , la respuesta deseada está dentro del subárbol izquierdo de la raíz.
3. Si  $q + 1 < k$ , la respuesta deseada está dentro del subárbol derecho de la raíz, y buscaremos el elemento más pequeño  $(k - q - 1)$ -ésimo en él. Este ajuste de  $k$  es necesario para garantizar que la respuesta es correcta.

Este proceso, que es similar al del algoritmo con expectativa de  $O(n)$  para el problema de selección estático, se puede repetir recursivamente hasta que encontremos la respuesta buscada. Como la comprobación del tamaño de un subárbol se puede hacer en  $O(1)$ , si hemos añadido la información correcta al BST, el tiempo de ejecución, en el peor de los casos, será de  $O(\log n)$ , desde la raíz hasta la hoja más profunda del BST equilibrado.

Sin embargo, como tenemos que añadir información al BST equilibrado, este algoritmo no puede utilizar los tipos `map`/`set` incorporados a la STL de C++ (o `TreeMap`/`TreeSet` en Java), ya que su funcionalidad no puede ser modificada. Por lo tanto, tendremos que escribir nuestra propia rutina de BST equilibrado (por ejemplo, un árbol AVL, ver la figura 9.12, un árbol rojo negro, etc., lo que incrementa el tiempo de programación) y, por ello, un problema de selección con *datos dinámicos* puede ser extremadamente difícil de resolver.

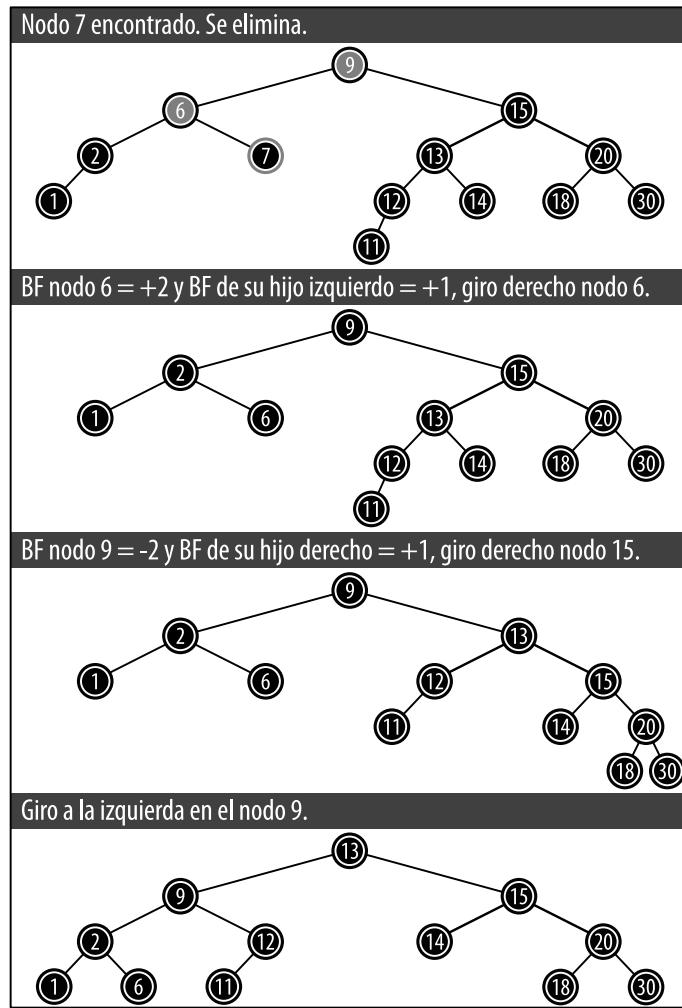


Figura 9.12: Ejemplo de eliminación en un árbol AVL (se elimina el 7)

### 9.30 Algoritmo más rápido para el camino más corto

El algoritmo más rápido para el camino más corto (SPFA), utiliza una cola para eliminar operaciones redundantes en el algoritmo de Bellman Ford. Fue publicado en chino por Duan Fanding, en 1994. Hoy día, es un algoritmo muy popular entre los programadores chinos, pero no es tan conocido en otras partes del mundo.

El SPFA necesita las siguientes estructuras de datos:

1. Un grafo almacenado en una lista de adyacencia: `AdjList` (sección 2.4.1).
2. `vi dist`, para registrar la distancia del origen a cada vértice.
3. Una `queue<int>` que almacena los vértices a procesar.
4. `vi in_queue`, para indicar si un vértice está en la cola o no.

Las tres primeras estructuras de datos son idénticas a las de los algoritmos de Dijkstra o Bellman Ford, de la sección 4.4. La cuarta es específica del SPFA. Podemos implementar el algoritmo de la siguiente manera:

```
1 // dentro de int main()
2 // inicialmente, solo S tiene dist = 0 y está en la cola
3 vi dist(n, INF); dist[S] = 0;
4 queue<int> q; q.push(S);
5 vi in_queue(n, 0); in_queue[S] = 1;
6
7 while (!q.empty()) {
8 int u = q.front(); q.pop(); in_queue[u] = 0;
9 for (j = 0; j < (int)AdjList[u].size(); j++) { // todos los vecinos de u
10 int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
11 if (dist[u] + weight_u_v < dist[v]) { // si podemos relajar
12 dist[v] = dist[u] + weight_u_v; // relajamos
13 if (!in_queue[v]) { // añadir a la cola
14 q.push(v); // solo si no está ya
15 in_queue[v] = 1;
16 } } } }
```



UVa10986.cpp



UVa10986.java

Este algoritmo se ejecuta en  $O(kE)$ , donde  $k$  es un número dependiente del grafo.  $k$  puede ser, como máximo, igual a  $V$  (que es igual a la complejidad de tiempo del algoritmo de Bellman Ford). Sin embargo, hemos comprobado que, para la mayoría de los problemas de SSSP del juez en línea de la UVa, que mencionamos en este libro, SPFA (que utiliza una cola) es igual de rápido que Dijkstra (que utiliza una cola de prioridad).

SPFA puede trabajar con aristas de peso negativo. Si el grafo no tiene ciclos negativos, SPFA funciona sin problema. Si el grafo tiene ciclos negativos, SPFA también puede detectarlo, pues debe haber algún vértice (que esté en el ciclo negativo) que entre en la cola más de  $V - 1$  veces. Podemos modificar el código anterior, para registrar cada vez que un vértice entra en la cola. Si descubrimos que cualquier vértice lo ha hecho más de  $V - 1$  veces, podemos concluir que el grafo tiene ciclos negativos.

## 9.31 Ventana corredera

### Enunciado del problema

Existen diversas variantes del problema de la ventana corredera, pero todas surgen de la misma idea básica: ‘deslizar’ un *subarray* (al que llamamos ‘ventana’, y que puede tener longitud estática o dinámica) de forma lineal, de izquierda a derecha, sobre el *array* original de  $n$  elementos, para calcular algo. Algunas de esas variantes son:

1. Encontrar el *subarray* de menor tamaño (longitud de ventana mínima), de forma que la suma del *subarray* sea mayor o igual que una determinada constante  $S$  en  $O(n)$ . Ejemplos:
  - Para el array  $A_1 = \{5, 1, 3, [5, 10], 7, 4, 9, 2, 8\}$  y  $S = 15$ , la respuesta es 2.
  - Para el array  $A_2 = \{1, 2, [3, 4, 5]\}$  y  $S = 11$ , la respuesta es 3.
2. Encontrar el *subarray* de menor tamaño (longitud de ventana mínima), de forma que sus elementos contengan todos los enteros del rango  $[1..K]$ . Ejemplos:
  - Para el array  $A = \{1, [2, 3, 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4], 5, 3, 1, 10, 3, 3\}$  y  $K = 4$ , la respuesta es 13.
  - Para el mismo array  $A = \{[1, 2, 3], 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4, 5, 3, 1, 10, 3, 3\}$  y  $K = 3$ , la respuesta es 3.
3. Encontrar la suma máxima de determinado *subarray* de tamaño (estático)  $K$ . Ejemplos:
  - Para el array  $A_1 = \{10, [50, 30, 20], 5, 1\}$  y  $K = 3$ , la respuesta es 100, al sumar el *subarray* resaltado.
  - Para el array  $A_2 = \{49, 70, 48, [61, 60], 60\}$  y  $K = 2$ , la respuesta es 121, al sumar el *subarray* resaltado.
4. Encontrar el mínimo de *cada* uno de los posibles *subarrays* de tamaño (estático)  $K$ :
  - Para el array  $A = \{0, 5, 5, 3, 10, 0, 4\}$ ,  $n = 7$  y  $K = 3$ , hay  $n - K + 1 = 7 - 3 + 1 = 5$  *subarrays* posibles, de tamaño  $K = 3$ , que son,  $\{0, 5, 5\}$ ,  $\{5, 5, 3\}$ ,  $\{5, 3, 10\}$ ,  $\{3, 10, 0\}$  y  $\{10, 0, 4\}$ . El mínimo de cada *subarray* es 0, 3, 3, 0, 0, respectivamente.

## Soluciones

Ignoraremos las soluciones ingenuas de estas variantes del problema de la ventana corredera y pasaremos, directamente, a las soluciones  $O(n)$ . Las siguientes cuatro soluciones se ejecutan en el mencionado  $O(n)$ , ya que lo que hacemos es ‘deslizar’ una ventana sobre el array original de  $n$  elementos, utilizando algunos trucos interesantes.

En la primera variante, mantenemos una ventana creciente (añadimos el elemento actual al final, lado derecho, de la ventana) y sumamos el valor del elemento actual a una suma continua, o decreciente (eliminamos el primer elemento, lado izquierdo, de la ventana) mientras la suma continua sea  $\geq S$ . Mantenemos la menor longitud de la ventana durante el proceso, e informamos de la respuesta.

En la segunda variante, mantenemos una ventana creciente si el rango  $[1..K]$  no ha quedado ya cubierto por los elementos de la ventana actual, o decreciente en caso contrario. Mantenemos la menor longitud de la ventana durante el proceso, e informamos de la respuesta. La comprobación de si el rango  $[1..K]$  está cubierto, o no, se puede simplificar mediante un método de conteo de frecuencia. Cuando todos los enteros  $\in [1..K]$  tienen una frecuencia distinta de cero, decimos que el rango  $[1..K]$  está cubierto. El aumento de la ventana incrementa la frecuencia de un entero determinado, lo que puede provocar que el rango  $[1..K]$  quede completamente cubierto (no tiene ‘huecos’), mientras que reducir la ventana, reduce también la frecuencia del entero eliminado y, si la frecuencia de ese entero llega a 0, el rango  $[1..K]$ , antes cubierto, dejará de estarlo (tiene un ‘hueco’).

Para la tercera variante, insertamos los primeros  $K$  enteros en la ventana, calculamos su suma y declaramos esta como el máximo actual. Después, deslizamos la ventana hacia la derecha, añadiendo un elemento al lado derecho de la ventana y eliminando otro del lado izquierdo, manteniendo así la longitud de la ventana en  $K$ . Incrementamos la suma con el valor del elemento añadido, menos el valor del eliminado, y los comparamos con el máximo que teníamos registrado, para verificar si estamos ante un nuevo máximo. Repetimos este proceso de deslizamiento  $n - K$  veces, e informamos de la suma máxima encontrada.

La cuarta variante es un poco más complicada, especialmente si  $n$  es grande. Para lograr una solución en  $O(n)$ , necesitamos utilizar una estructura de datos deque (cola de dos extremos), para modelar la ventana. Esto es debido a que la deque cuenta con inserción y eliminación eficiente, en  $O(1)$ , tanto en el principio como en el final de la cola (ver los comentarios sobre la deque en la sección 2.2). En esta ocasión, mantendremos la ventana (es decir, la deque) ordenada de forma ascendente, en otras palabras, que el índice más alto de la deque almacena el menor valor. Sin embargo, esto cambia el orden de los elementos del array. Para mantener un registro de si un elemento se encuentra, en un momento dado, dentro de la ventana, también necesitamos recordar su índice. El siguiente código de C++ explica las acciones en más detalle. Esta ventana ordenada se puede reducir por sus dos extremos, y puede crecer desde el final. De ahí la necesidad de utilizar una estructura de datos de deque<sup>27</sup>.

```

1 void SlidingWindow(int A[], int n, int K) {
2 // ii, o pair<int, int>, representa el par (A[i], i)
3 deque<ii> window; // mantenemos la 'ventana' con orden ascendente
4 for (int i = 0; i < n; i++) { // esto es O(n)
5 while (!window.empty() && window.back().first >= A[i])
6 window.pop_back(); // para mantener la 'ventana' siempre ordenada
7
8 window.push_back(ii(A[i], i));
9
10 // usar el segundo campo para ver si forma parte de la ventana actual
11 while (window.front().second <= i-K) // eliminación perezosa
12 window.pop_front();
13
14 if (i+1 >= K) // desde la primera ventana de longitud K en adelante
15 printf("%d\n", window.front().first); // respuesta para esta ventana
16 }
}

```

## Ejercicios de programación

- |                                                                                                                                                       |                                                                                                                                                                 |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. <a href="#">UVa 01121 - Subsequence *</a><br>2. <a href="#">UVa 11536 - Smallest Sub-Array *</a><br>3. IOI 2011 - Hottest<br>4. IOI 2011 - Ricehub | (variante 1 de ventana corredera)<br>(variante 2 de ventana corredera)<br>(ejercicio de práctica, variante 3 de ventana corredera)<br>(ventana corredera y más) |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|

---

<sup>27</sup>Estructura de datos que no resulta necesaria en las variantes 1 a 3.

## 5. IOI 2012 - Tourist Plan

(ejercicio de práctica; otra variante de ventana corredora; la mejor respuesta que empieza en la ciudad 0 y acaba en la ciudad  $i \in [0..N-1]$ , es la suma de la felicidad de las  $K-i$  superiores  $\in [0..i]$ ; usar priority\_queue; mostrar la suma máxima)

## 9.32 Ordenación en tiempo lineal

### Enunciado del problema

Dado un *array* (no ordenado) de  $n$  elementos, ¿podemos ordenarlo en tiempo  $O(n)$ ?

### Límite teórico

En el caso general, el límite inferior de un algoritmo de ordenación genérico (basado en la comparación) es  $\Omega(n \log n)$  (ver la demostración utilizando un modelo de árbol de decisión en otras fuentes, como [7]). Sin embargo, si los  $n$  elementos tienen una propiedad especial, podemos utilizar un algoritmo más rápido en  $O(n)$ , lineal, al no realizar comparaciones entre elementos. A continuación, veremos dos ejemplos.

### Soluciones

#### Ordenación por cuentas

Si el *array*  $A$  contiene  $n$  enteros en un rango *pequeño*  $[L..R]$  (por ejemplo, la edad de los humanos de  $[1..99]$  años, en el problema UVa 11462 - Age Sort), podemos utilizar el algoritmo de ordenación por cuentas. A efectos de la siguiente explicación, asumimos que el array  $A$  es  $\{2, 5, 2, 2, 3, 3\}$ . La idea de la ordenación por cuentas es la siguiente:

1. Preparar un ‘*array* de frecuencias’  $f$ , de tamaño  $k = R-L+1$ , e inicializarlo con ceros.  
En el *array* de ejemplo anterior, tenemos  $L = 2$ ,  $R = 5$  y  $k = 4$ .
2. Hacemos una pasada en el *array*  $A$ , y actualizamos la frecuencia de cada entero que encontramos, es decir, para cada  $i \in [0..n-1]$ , ejecutamos  $f[A[i]-L]++$ .  
En el *array* de ejemplo anterior, tenemos  $f[0] = 3$ ,  $f[1] = 2$ ,  $f[2] = 0$ ,  $f[3] = 1$ .
3. Una vez que conoczcamos la frecuencia de cada entero del pequeño rango, calculamos las sumas de los prefijos de cada  $i$ , es decir,  $f[i] = f[i-1] + f[i] \forall i \in [1..k-1]$ . Ahora,  $f[i]$  contiene el número de elementos menores o iguales a  $i$ .  
En el *array* de ejemplo anterior, tenemos  $f[0] = 3$ ,  $f[1] = 5$ ,  $f[2] = 5$ ,  $f[3] = 6$ .
4. Despues, vamos hacia atrás desde  $i = n-1$  hasta  $i = 0$ . Colocamos  $A[i]$  en el índice  $f[A[i]-L]-1$ , ya que es su ubicación correcta. Reducimos  $f[A[i]-L]$  en uno, para que la siguiente copia de  $A[i]$ , si la hay, se coloque justo antes del  $A[i]$  actual.

En el *array* de ejemplo anterior, empezamos colocando  $A[5] = 3$  en el índice  $f[A[5]-2]-1 = f[1]-1 = 5-1 = 4$ , y reducimos  $f[1]$  a 4. Despúes, colocamos  $A[4] = 3$  (mismo valor que  $A[5] = 3$ ) en el índice  $f[A[4]-2]-1 = f[1]-1 = 4-1 = 3$ , y reducimos  $f[1]$  a 3. Entonces, colocamos  $A[3] = 2$  en el índice  $f[A[3]-2]-1 = 2$ , y reducimos  $f[0]$  a 2. Repetimos los tres pasos hasta que obtengamos un *array* ordenado {2, 2, 2, 3, 3, 5}.

La complejidad de tiempo de la ordenación por cuentas es de  $O(n + k)$ . Cuando  $k = O(n)$ , el algoritmo, en teoría, se ejecuta en tiempo lineal, al *no* compararar los enteros. Sin embargo, en el entorno de un concurso de programación,  $k$  normalmente no será muy grande, para evitar un veredicto de límite de memoria superado. Por ejemplo, la ordenación por cuentas tendrá problemas para ordenar un *array* A, con  $n = 3$ , que contenga {1, 1000000000, 2}, ya que incluye un  $k$  muy grande.

### Ordenación radix

Si el *array* A contiene  $n$  enteros no negativos, con un rango  $[L..R]$  relativamente amplio, pero un número pequeño de dígitos, podemos utilizar el algoritmo de ordenación *radix*.

La idea de la ordenación *radix* es sencilla. En primer lugar, hacemos que todos los enteros tengan  $d$  dígitos, donde  $d$  es el número de dígitos del entero más grande en A, añadiendo ceros si fuese necesario. Después, la ordenación *radix* ordenará los números dígito a dígito, empezando desde el *menos* significativo avanzando hacia el *más* significativo. Utiliza otro algoritmo de *ordenación estable*, como subrutina para ordenar los dígitos, como la ordenación por cuentas en  $O(n + k)$  que ya hemos visto. Por ejemplo:

| Entrada<br>$d = 4$ | Añadir<br>ceros | Ordenar por el<br>cuarto dígito | Ordenar por el<br>tercer dígito | Ordenar por el<br>segundo dígito | Ordenar por el<br>primer dígito |
|--------------------|-----------------|---------------------------------|---------------------------------|----------------------------------|---------------------------------|
| 323                | 0323            | 032(2)                          | 00(1)3                          | 0(0)13                           | (0)013                          |
| 1257               | 1257            | 032(3)                          | 03(2)2                          | 1(2)57                           | (0)322                          |
| 13                 | 0013            | 001(3)                          | 03(2)3                          | 0(3)22                           | (0)323                          |
| 322                | 0322            | 125(7)                          | 12(5)7                          | 0(3)23                           | (1)257                          |

Para un *array* de  $n$  enteros de  $d$  dígitos, realizaremos  $O(d)$  pasadas de ordenación de cuentas, con una complejidad de tiempo de  $O(n + k)$  cada una. Por lo tanto, la complejidad de la ordenación *radix* es de  $O(d \times (n + k))$ . Si utilizamos esta ordenación para  $n$  enteros con signo de 32 bits ( $\approx d = 10$  dígitos) y  $k = 10$ , tendremos una complejidad total de  $O(10 \times (n + 10))$ . Podemos seguir asumiéndolo como ejecución en tiempo lineal, pero con un factor constante alto.

Considerando la dificultad de escribir la compleja rutina de ordenación *radix*, frente a la sencillez de llamar a la función estándar *sort* de la STL de C++ (o *Collections.sort* de Java), con tiempo  $O(n \log n)$ , resultará extraño que se utilice en concursos de programación. En este libro, solo hemos utilizado esta combinación de ordenaciones *radix* y de cuentas en nuestra implementación del *array* de sufijos (sección 6.6.4).

### Ejercicio 9.32.1\*

¿Qué deberíamos hacer si queremos utilizar la ordenación *radix*, pero el *array* A contiene, al menos, un número negativo?

### 1. UVa 11462 - Age Sort\* (problema de ordenación por cuentas estándar)

## 9.33 Estructura de datos de tabla dispersa

En la sección 2.4.3, hemos visto que se puede utilizar la estructura de datos del árbol de segmentos, para resolver el problema de la consulta de rango mínimo, o problema de búsqueda del índice que contenga el menor elemento de un rango  $[i..j]$  del *array* subyacente A. El tiempo de procesamiento previo de construcción de un árbol de segmentos es de  $O(n)$  y, una vez generado, cada RMQ tiene un consumo de solo  $O(\log n)$ . Con un árbol de segmentos, podemos tratar la *versión dinámica* de este problema de la RMQ, es decir, cuando se actualiza el *array* subyacente, solo necesitaremos  $O(\log n)$  para actualizar el árbol correspondiente.

Sin embargo, en algunos problemas que implican RMQ, el *array* subyacente A permanece inmutado tras la primera consulta. En este caso, hablamos del problema *estático* de la RMQ. Aunque, evidentemente, se puede utilizar un árbol de segmentos para resolver el problema estático de la RMQ, existe una alternativa de DP con un tiempo de procesamiento de  $O(n \log n)$  y un coste por consulta de  $O(1)$ . Un ejemplo lo encontramos en el problema del ancestro común mínimo (LCA), de la sección 9.18.

La idea clave de la solución de DP radica en dividir A en *subarrays* de longitud  $2^j$ , para cada entero no negativo  $j$ , de forma que  $2^j \leq n$ . Mantendremos un *array* SpT, de tamaño  $n \times \log n$ , donde  $\text{SpT}[i][j]$  almacena el índice del valor mínimo del *subarray* que comience en el índice i, y con longitud  $2^j$ . Este *array* SpT será disperso, ya que no todas sus celdas almacenarán un valor (de ahí el nombre de ‘tabla dispersa’). Usamos la abreviatura SpT, para diferenciar esta estructura de datos del árbol de segmentos (ST).

Para construir el *array* SpT, utilizaremos una técnica similar a la de muchos algoritmos de divide y vencerás, como es la ordenación por mezcla. Sabemos que, en un *array* de longitud 1, su único elemento es también el menor. Este será nuestro caso base. Para encontrar el índice del elemento más pequeño de un *array* de tamaño  $2^j$ , podemos comparar los valores de los índices de los elementos más pequeños de dos *subarrays* distintos, de tamaño  $2^{j-1}$ , y tomar el menor. Construir un *array* SpT como este, tiene un coste de  $O(n \log n)$ . Te recomendamos que analices detenidamente el constructor de la clase RMQ del código fuente que incluimos a continuación, y que implementa la construcción del *array* SpT.

Es fácil entender cómo se procesaría una consulta, si la longitud del rango fuese una potencia de 2. Como esta es, exactamente, la información que almacena SpT, devolveríamos, sin más, la entrada correspondiente. Sin embargo, para calcular el resultado de una consulta con índices de inicio y fin arbitrarios, tenemos que recuperar la entrada de dos *subarrays* más pequeños y tomar el mínimo de ellos. Es posible que estos dos *subarrays* se superpongan, pero la cuestión es que queremos cubrir el rango completo con dos *subarrays*, y no tomar nada fuera del mismo. Esto siempre es posible, incluso aunque la longitud de los *subarrays* tenga que ser una potencia de 2. En primer lugar, hallamos la longitud del rango de consulta, que es  $j - i + 1$ . Después, le aplicamos  $\log_2$  y redondeamos el resultado a la baja, es decir,  $k = \lfloor \log_2(j - i + 1) \rfloor$ . De esta forma,  $2^k \leq (j - i + 1)$ . La figura 9.13 muestra, con sencillez, el aspecto que podrían tener los dos *subarrays*. Como, potencialmente, puede haber subproblemas superpuestos, clasificamos esta parte de la solución como programación dinámica.

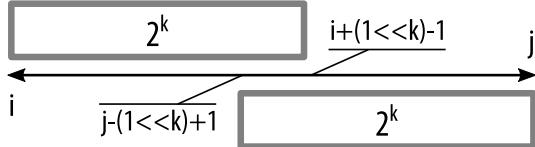


Figura 9.13: Explicación de RMQ( $i, j$ )

A continuación, incluimos una implementación de una tabla dispersa, para resolver el problema estático de la RMQ. Puedes compararla con la del árbol de segmentos de la sección 2.4.3.

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define MAX_N 1000 // ajustar el valor según sea necesario
5 #define LOG_TWO_N 10 // $2^{10} > 1000$, ajustar el valor según sea necesario
6
7 class RMQ { // consulta de rango mínimo
8 private:
9 int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
10 public:
11 RMQ(int n, int A[]) { // constructor y rutina de procesamiento previo
12 for (int i = 0; i < n; i++) {
13 _A[i] = A[i];
14 SpT[i][0] = i; // RMQ de subarray con índice de inicio $i + \text{long. } 2^0=1$
15 }
16 // los dos bucles anidados siguientes tienen complejidad = $O(n \log n)$
17 for (int j = 1; (1<<j) <= n; j++) // para cada $j < 2^j \leq n$, $O(\log n)$
18 for (int i = 0; i + (1<<j) - 1 < n; i++) // para cada i válida, $O(n)$
19 if (_A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]]) // RMQ
20 SpT[i][j] = SpT[i][j-1]; // inicio en índice i de longitud $2^{(j-1)}$
21 else // inicio en índice $i+2^{(j-1)}$ de longitud $2^{(j-1)}$
22 SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
23 }
24
25 int query(int i, int j) { // esta consulta es $O(1)$
26 int k = (int)floor(log((double)j-i+1) / log(2.0)); // $2^k \leq (j-i+1)$
27 if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];
28 else return SpT[j-(1<<k)+1][k];
29 } };

```



SparseTable.cpp



SparseTable.java

Para el mismo caso de prueba con  $n = 7$  y  $A = \{18, 17, 13, 19, 15, 11, 20\}$ , que el de la sección 2.4.3, el contenido de la tabla dispersa SpT es:

| Índice | 0 | 1     | 2     |
|--------|---|-------|-------|
| 0      | 0 | 1     | 2     |
| 1      | 1 | 2     | 2     |
| 2      | 2 | 2     | 5     |
| 3      | 3 | 4     | 5     |
| 4      | 4 | 5     | vacío |
| 5      | 5 | 5     | vacío |
| 6      | 6 | vacío | vacío |

En la primera columna, tenemos  $j = 0$ , que indica la RMQ del *subarray* que empieza en el índice  $i$ , con longitud  $2^0 = 1$ , eso significa  $\text{SpT}[i][j] = i$ .

En la segunda columna, tenemos  $j = 1$ , que indica la RMQ del *subarray* que empieza en el índice  $i$ , con longitud  $2^1 = 2$ . Es de importancia que la última fila está vacía.

En la tercera columna, tenemos  $j = 2$ , que indica la RMQ del *subarray* que empieza en el índice  $i$ , con longitud  $2^2 = 4$ . Como vemos, las tres últimas filas están vacías.

## 9.34 Torres de Hanoi

### Enunciado del problema

La descripción clásica del problema es la siguiente: tenemos tres varillas,  $A$ ,  $B$  y  $C$ , y  $n$  discos, todos de diferentes tamaños. Comenzando con todos los discos apilados en orden ascendente en una de las varillas (la  $A$ ), la tarea consiste en desplazar los  $n$  discos a otra de las varillas (la  $C$ ). No se permite colocar un disco encima de otro de tamaño menor, y solo se puede mover un disco cada vez, desde la parte superior de una varilla a otra.

### Soluciones

Existe una solución sencilla de *backtracking* recursivo para el problema clásico. La cuestión de mover  $n$  discos de la varilla  $A$  a la  $C$ , utilizando la varilla  $B$  como intermedia, se puede dividir en los siguientes subproblemas:

1. Mover  $n - 1$  discos de la varilla  $A$  a la  $B$ , utilizando la  $C$  como intermedia. Después de realizar este paso recursivo, nos queda el disco  $n$  solo en la varilla  $A$ .
2. Mover el disco  $n$  de la varilla  $A$  a la  $C$ .
3. Mover  $n - 1$  discos de la varilla  $B$  a la  $C$ , utilizando la  $A$  como intermedia. Estos  $n - 1$  discos quedarán encima del disco  $n$ , que ahora está abajo de la pila de la varilla  $C$ .

Los pasos 1 y 3 son recursivos. El caso base lo encontramos cuando  $n = 1$ , donde nos basta con mover el único disco desde su varilla de origen a la de destino, saltando la intermedia. Esta es una implementación de ejemplo en C++:

```

1 #include <cstdio>
2 using namespace std;
3
4 void solve(int count, char source, char destination, char intermediate) {
5 if (count == 1)
6 printf("Move top disc from pole %c to pole %c\n", source, destination);
7 else {
8 solve(count-1, source, intermediate, destination);
9 solve(1, source, destination, intermediate);
10 solve(count-1, intermediate, destination, source);
11 }
12 }
13
14 int main() {
15 solve(3, 'A', 'C', 'B');// probar un valor mayor para el primer parámetro
16 } // return 0;

```

El número mínimo de movimientos necesarios para resolver el problema clásico de las Torres de Hanoi, con  $n$  discos, mediante esta solución de *backtracking* recursivo, es de  $2^n - 1$ .

## Ejercicios de programación

- 1. UVa 10017 - The Never Ending ... \*** (versión clásica del problema)

## 9.35 Notas del capítulo

Este capítulo contiene 34 temas poco habituales, 10 de ellos son algoritmos poco usados (destacados en **negrita**), los otros 24 son problemas.

|                                                     |                                                             |
|-----------------------------------------------------|-------------------------------------------------------------|
| Problema 2-SAT                                      | Problema de la galería de arte                              |
| Problema del viajante bitónico                      | Emparejamiento de paréntesis                                |
| Problema del cartero chino                          | Problema del par más cercano                                |
| <b>Algoritmo de Dinic</b>                           | <b>Fórmulas o teoremas</b>                                  |
| <b>Algoritmo de eliminación gausiana</b>            | Emparejamiento de grafos                                    |
| <b>Distancia de círculo máximo</b>                  | <b>Algoritmo de Hopcroft Karp</b>                           |
| Caminos independientes y arista-disjuntos           | Índice de inversión                                         |
| <b>Problema de Josefo</b>                           | Movimientos del caballo                                     |
| <b>Algoritmo de Kosaraju</b>                        | Ancestro común mínimo                                       |
| Construcción de cuadrados mágicos (de tamaño impar) | Multiplicación de cadenas de matrices                       |
| <b>Potencia de matrices</b>                         | Conjunto independiente ponderado máximo                     |
| Flujo (máximo) de coste mínimo                      | Cobertura de caminos mínima en un DAG                       |
| Ordenación de tortitas                              | <b>Algoritmo de factorización de enteros rho de Pollard</b> |
| Calculadora posfixa y conversión                    | Números romanos                                             |
| Problema de selección                               | Algoritmo más rápido de camino más corto                    |
| <b>Ventana corredera</b>                            | Ordenación en tiempo lineal                                 |
| Estructura de datos de tabla dispersa               | Torres de Hanoi                                             |

Sin embargo, después de añadir tanta materia nueva a la presente edición del libro, hemos advertido que hay muchos otros temas sobre ciencias de la computación que no hemos cubierto.

Finalizamos este capítulo y, con ello, este libro, con una lista de un buen número de palabras clave que no han sido incluidas, en parte debido a la ‘fecha límite de entrega’ que, nosotros mismos, nos hemos impuesto.

Quedan muchas estructuras de datos *exóticas*, que raramente aparecerán en concursos de programación: montículo de Fibonacci, varias técnicas de *hashing* (tablas de *hashes*), descomposición pesada-ligera de un árbol con raíz, árbol de intervalos, árbol *k-d*, lista enlazada (esta la hemos evitado intencionadamente), árbol *radix*, árbol de rangos, lista de saltos, *treap*, etc.

El tema del flujo de red es mucho más amplio que la materia incluida en la sección 4.6 y otras de este capítulo. Otros asuntos, como el problema de eliminación de béisbol, el problema de circulación, el árbol Gomory-Hu, el algoritmo *push-re etiquetado*, el algoritmo de corte mínimo de Stoer-Wagner y el poco conocido algoritmo de Suurballe, se podrían añadir en el futuro.

También podríamos añadir un tratamiento más detallado de algunos algoritmos de la sección 9.10, en concreto, el algoritmo de emparejamiento de Edmonds [13], el algoritmo de Gale Shapley para el problema del matrimonio estable, y el algoritmo Kuhn Munkres (húngaro) [39, 45].

Existen, por supuesto, muchos otros problemas y algoritmos matemáticos que se deben considerar, como el teorema del resto chino, inversa multiplicativa modular, función de Möbius, varios problemas *exóticos* de teoría de números, varios métodos numéricos, etc.

En las secciones 6.4 y 6.6, hemos visto las soluciones KMP y de árbol/*array* de sufijos para el problema de la coincidencia de cadenas. La coincidencia de cadenas es un tema muy bien estudiado y existen otros algoritmos, como los de Aho Corasick, Boyer Moore y Rabin Karp.

En la sección 8.2, vimos técnicas de búsqueda más avanzadas. Algunos problemas de concursos son NP-complejos (o NP-completos), pero con un tamaño de entrada pequeño. La solución se suele encontrar en una búsqueda completa creativa. Hemos mencionado algunos problemas NP-complejos/NP-completos en este libro, pero podríamos añadir más, como el problema del coloreado de grafos, el problema *max clique*, el problema del viajante comprador, etc.

Por último, enumeramos muchas otras materias que, potencialmente, podrán encontrar su lugar en futuras ediciones de este libro, como la transformación de Burrows-Wheeler, el algoritmo de Chu-Liu Edmonds, la codificación Huffman, el algoritmo del ciclo de peso medio mínimo de Karp, técnicas de programación lineal, círculos de Malfatti, el problema de la cobertura de círculos mínima, el árbol de expansión de diámetro mínimo, el árbol de expansión mínimo de un vértice con limitación de grado, otras bibliotecas de geometría computacional no mencionadas en el capítulo 7, el árbol de búsqueda binario óptimo para ilustrar la aceleración de DP de Knuth-Yao [2], el algoritmo del calibre giratorio, el problema de la supercadena común más corta, el problema del árbol de Steiner, búsqueda ternaria, el puzzle de Triomino, etc.



## Apéndice A

### uHunt

**uHunt** (<http://uhunt.felix-halim.net>) es una herramienta orientada al autoaprendizaje para el Online Judge (UVa OJ [49]), creada por Felix Halim. Aporta dinamismo a la resolución de problemas del Online Judge, y lo logra mediante las siguientes características:

1. Información y estadísticas en tiempo real sobre los envíos, para que los usuarios puedan mejorar rápidamente sus soluciones (ver la figura A.1). Un usuario podrá conocer, inmediatamente, la clasificación de la eficiencia de su solución en relación al resto. Si encontramos una diferencia importante entre el rendimiento de nuestra solución y la de la mejor para ese problema, podremos deducir que no conocemos determinados algoritmos, estructuras de datos o técnicas, que resultarían necesarios. uHunt también cuenta con un ‘comparador de estadísticas’. Si tienes un *rival* (u otro usuario al que admirás), puedes comparar tu lista de problemas resueltos con la suya, y tratar de ponerte a su nivel.

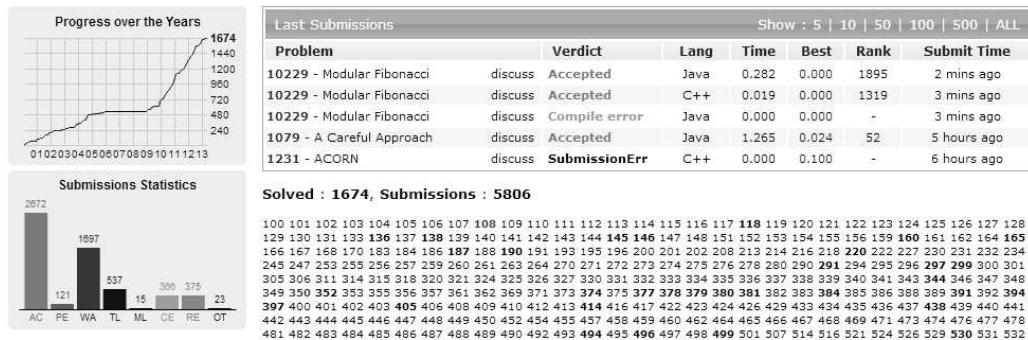


Figura A.1: Estadísticas de Steven el 24 de mayo de 2013

2. Una API para que otros desarrolladores puedan construir sus propias herramientas. De hecho, esta característica ya ha sido utilizada para crear un sistema de gestión de concursos completo, una herramienta de línea de comandos para enviar soluciones y obtener el resultado en la misma consola y una aplicación móvil para consultar las estadísticas.
3. Un medio para que los usuarios se ayuden entre ellos. El sistema de charla, situado en la esquina superior derecha de la página, se ha utilizado constantemente para intercambiar ideas y ayudar a otros a resolver problemas. Esto proporciona un entorno de aprendizaje en el que cualquier usuario tiene la posibilidad de pedir consejo.

4. Una selección del siguiente problema a resolver, ordenada por dificultad creciente (aproximadamente por el número de usuarios diferentes que han logrado un veredicto de aceptado). Resulta útil para aquellos usuarios que desean resolver problemas cuya dificultad esté acorde a sus conocimientos. La razón es que, si un usuario está empezando a resolver problemas y necesita ganar confianza, deberá resolver problemas cuya dificultad aumente de forma gradual. Resulta mucho mejor que intentar, directamente, los problemas más difíciles y, con ello, no recibir más que veredictos de respuesta incorrecta, sin tener idea de lo que está pasando. Los  $\approx 149008$  usuarios del Online Judge aportan información estadística a cada problema, información que utilizamos para este propósito. Los problemas más sencillos tienen un número mayor de envíos y una mejor tasa de soluciones aceptadas. Sin embargo, como un usuario puede seguir enviando soluciones para un problema que ya tiene aceptado, esta cifra no resulta precisa por sí misma para determinar la dificultad de un problema. Podríamos encontrar un ejemplo radical en un usuario que, habiendo resuelto un problema, realiza 50 envíos aceptados adicionales, en un intento de mejorar su tiempo de ejecución. Tal problema no tendría por qué ser más sencillo que otro donde fuesen 49 los usuarios que hubiesen obtenido el veredicto de aceptado. Para gestionar esta información, el criterio de ordenación predeterminado de uHunt se denomina ‘dacu’, que significa ‘usuarios aceptados distintos’. El problema, difícil, de nuestro ejemplo radical, solo tendrá un dacu = 1, mientras que el más fácil puntuará dacu = 49 (figura A.2).

| Volume : ALL |        |                           | View : [ unsolved   solved   both ] |      | Show : [ 25   50   100 ] |      |      |       | Volumes |  |
|--------------|--------|---------------------------|-------------------------------------|------|--------------------------|------|------|-------|---------|--|
| No           | Number | Problem Title             | nos                                 | anos | %anos                    | dacu | best | v1    | v2      |  |
| 1            | 705    | Slash Maze                | discuss                             | 4662 | 1898                     | 40%  | 1078 | 0.000 | 72%     |  |
| 2            | 10254  | The Priest Mathematician  | discuss                             | 3810 | 1670                     | 43%  | 843  | 0.004 | 42%     |  |
| 3            | 10202  | Pairsunomious Numbers     | discuss                             | 3012 | 1187                     | 39%  | 836  | 0.000 | 61%     |  |
| 4            | 134    | Loglan-A Logical Langu... | discuss                             | 3304 | 892                      | 26%  | 736  | 0.000 | 75%     |  |
| 5            | 132    | Bumpy Objects             | discuss                             | 3083 | 1241                     | 40%  | 728  | 0.000 | 58%     |  |
| 6            | 254    | Towers of Hanoi           | discuss                             | 5884 | 1081                     | 18%  | 723  | 0.008 | 58%     |  |
| 7            | 704    | Colour Hash               | discuss                             | 3593 | 1538                     | 42%  | 699  | 0.008 | 44%     |  |
| 8            | 302    | John's trip               | discuss                             | 7031 | 1418                     | 20%  | 648  | 0.006 | 40%     |  |
| 9            | 10776  | Determine The Combin...   | discuss                             | 1878 | 838                      | 44%  | 618  | 0.000 | 15%     |  |
|              |        |                           |                                     |      |                          |      |      | v11   | v12     |  |

Figura A.2: A la caza del siguiente problema más fácil mediante ‘dacu’

5. Un sistema de creación de concursos virtual. Los usuarios pueden decidir crear un concurso cerrado para ellos, eligiendo entre todos los problemas disponibles, con un tiempo de competición determinado. Resulta muy útil al entrenar, tanto individualmente como por equipos. Algunos concursos pueden tener *sombra* (concurseantes del pasado), para que el usuario pueda comparar su capacidad con la de otros participantes de un tiempo anterior.

## World Finals Warmup I

## Quick Submit

| Last Submissions [ hide last submissions ] |                           |                            |              |      |       |       |      |             |  | Show : 5   10   20   50   100 |  |  |  |
|--------------------------------------------|---------------------------|----------------------------|--------------|------|-------|-------|------|-------------|--|-------------------------------|--|--|--|
| #                                          | Problem Title             | Author Name                | Verdict      | Lang | Time  | Best  | Rank | Submit Time |  |                               |  |  |  |
| 545                                        | 12439 February 29         | Ivan Reyes (spberman)      | Time limit   | Java | 1.000 | 0.000 | -1   | 5 secs ago  |  |                               |  |  |  |
| 544                                        | 12439 February 29         | Bulat S. (soul_rebel)      | Wrong answer | C++  | 0.016 | 0.000 | -1   | 10 secs ago |  |                               |  |  |  |
| 543                                        | 12435 Consistent Verdicts | Andrej Gajduk (Gajduk)     | Wrong answer | Java | 0.804 | 0.292 | -1   | 12 secs ago |  |                               |  |  |  |
| 542                                        | 12439 February 29         | Patrick Klitzke (philolo1) | Accepted     | C++  | 0.012 | 0.000 | -1   | 16 secs ago |  |                               |  |  |  |
| 541                                        | 12439 February 29         | Mike Shvets (mike.shvets)  | Wrong answer | C++  | 0.012 | 0.000 | -1   | 16 secs ago |  |                               |  |  |  |

| Contest Ranklist |               |       |       |          |          |          |       |          |           |          |       |          | Time remaining: 22 hours 8 minutes |
|------------------|---------------|-------|-------|----------|----------|----------|-------|----------|-----------|----------|-------|----------|------------------------------------|
| #                | Author Name   | 12433 | 12434 | 12435    | 12436    | 12437    | 12438 | 12439    | 12440     | 12441    | 12442 | 12443    | AC / Time                          |
| 1                | panyuchao     |       |       | 0:40     | (2) 1:16 |          |       | (1) 1:11 | 1:44      |          | 0:52  | 5 / 6:46 |                                    |
| 2                | surwdkgo      |       |       | 0:58     |          |          |       | 0:11     |           |          | 1:13  | 0:46     | 4 / 3:10                           |
| 3                | PMP Forever   |       |       | 0:19     |          | (1) 1:35 |       | 0:38     |           |          | 1:05  |          | 4 / 3:58                           |
| 4                | Anton Raichuk |       |       | (1) 0:41 |          |          |       | (1) 0:14 | (1) --:-- | (4) 1:15 | 0:35  | 4 / 4:47 |                                    |
| 5                | Submitor      |       |       | 0:52     |          | (3) 1:49 |       | (1) 0:36 |           |          | 1:18  |          | 4 / 5:57                           |

Figura A.3: Podemos reproducir concursos pasados, mediante los ‘concursos virtuales’

6. La integración de los  $\approx 1675$  ejercicios de programación de este libro, en varias categorías (ver la figura A.4). Los usuarios pueden comprobar qué ejercicios de programación del libro ya han resuelto y mantener un registro del progreso de su trabajo. Estos ejercicios de programación se pueden utilizar incluso sin hacer uso del libro. Un usuario puede personalizar su técnica de entrenamiento, resolviendo *problemas de contenido similar*. Sin esa categorización (manual), el modo de entrenamiento es difícil de ejecutar. También resaltamos (\*) aquellos problemas que consideramos **obligatorios \*** (hasta 3 por categoría).

**3rd Edition's Exercises (switch to: 1st, 2nd, 3rd )**

| Book Chapters                    | Starred ★ | ALL |
|----------------------------------|-----------|-----|
| 1. Introduction                  | 92%       | 80% |
| 2. Data Structures and Libraries | 75%       | 77% |
| 3. Problem Solving Paradigms     | 75%       | 76% |
| 4. Graph                         | 76%       | 74% |
| 5. Mathematics                   | 87%       | 79% |
| 6. String Processing             | 78%       | 78% |
| 7. (Computational) Geometry      | 61%       | 72% |
| 8. More Advanced Topics          | 47%       | 58% |
| 9. Rare Topics                   | 65%       | 69% |

**Problem Decomposition (47/89 = 52%)**

| Two Components - Binary Search the Answer and Other (5/14) |                                |
|------------------------------------------------------------|--------------------------------|
| 714 - Copying Books                                        | discuss Lev 3 ✓ 0.020s/781(13) |
| 1221 - Against Mammots                                     | discuss Lev 8 --- ? ---        |
| 1280 - Curvy Little Bottles                                | discuss Lev 6 --- ? ---        |
| 10372 - Leaps Tall Buildings (in ...                       | discuss Lev 5 ✓ 0.004s/130     |
| <b>10566 - Crossed Ladders</b>                             | discuss Lev 4 ✓ 0.000s/20      |
| 10606 - Opening Doors                                      | discuss Lev 5 --- ? ---        |
| 10668 - Expanding Rods                                     | discuss Lev 5 --- ? ---        |
| 10804 - Gopher Strategy                                    | discuss Lev 5 Tried (8)        |
| 10816 - Travel in Desert                                   | discuss Lev 4 ✓ 0.700s/391 (4) |
| 10983 - Buy one, get the rest fr...                        | ★ discuss Lev 5 --- ? ---      |
| <b>11262 - Weird Fence</b>                                 | ★ discuss Lev 5 ✓ 0.064s/47    |
| 11516 - WiFi                                               | ★ discuss Lev 5 --- ? ---      |
| 11646 - Athletics Track                                    | discuss Lev 4 --- ? ---        |
| 12428 - Enemy at the Gates                                 | discuss Lev 6 --- ? ---        |

Figura A.4: Los ejercicios de programación de este libro están integrados en uHunt

Construir una herramienta web como uHunt es, en sí mismo, un reto computacional. Hay más de  $\approx 11796315$  envíos de  $\approx 149008$  usuarios ( $\approx$  un envío cada pocos segundos). Es necesario actualizar las estadísticas y clasificaciones con frecuencia, y hacerlo rápido. Para afrontar el reto, Felix utiliza estructuras de datos avanzadas (algunas fuera del ámbito de este libro), como la técnica *database cracking* [29], árbol de Fenwick, compresión de datos, etc.

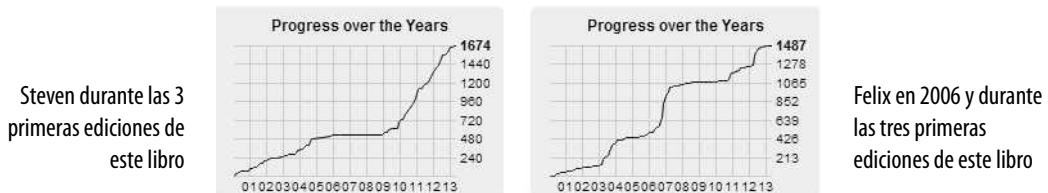


Figura A.5: Progreso de Steven y Felix en el Online Judge (2000-2013)

Nosotros mismos hemos utilizado con asiduidad esta herramienta en diferentes etapas de nuestra vida, como se puede observar en la figura A.5. Dos momentos cruciales que quedan patentes son, el entrenamiento intensivo de Felix para lograr la victoria en el ACM ICPC Kaohsiung 2006, con su equipo del ICPC, y la actividad frenética de Steven durante los años de preparación de este libro (finales de 2009 a 2013).



## Apéndice B

---

### Créditos

Los problemas tratados en este libro proceden, principalmente, del Online Judge [49], el ICPC Live Archive [33] y concursos pasados de la IOI (2009-2012). Hemos podido contactar con los siguientes autores y obtenido su permiso para publicarlos (en orden alfabético):

1. Brian C. Dean (Clemson University, Estados Unidos)
2. Colin Tan Keng Yan (Universidad Nacional de Singapur, Singapur)
3. Derek Kisman (University of Waterloo, Canadá)
4. Gordon V. Cormack (University of Waterloo, Canadá)
5. Howard Cheng (University of Lethbridge, Canadá)
6. Jane Alam Jan (Google)
7. Jim Knisely (Bob Jones University, Estados Unidos)
8. Jittat Fakcharoenphol (Universidad de Kasetsart, Tailandia)
9. Manzurur Rahman Khan (Google)
10. Melvin Zhang Zhiyong (Universidad Nacional de Singapur, Singapur)
11. Michal (Misof) Forišek (Universidad de Comenius, Eslovaquia)
12. Mohammad Mahmudur Rahman (University of South Australia, Australia)
13. Norman Hugh Anderson (Universidad Nacional de Singapur, Singapur)
14. Ondřej Lhoták (University of Waterloo, Canadá)
15. Petr Mitrichev (Google)
16. Piotr Rudnicki (University of Alberta, Canadá)
17. Rob Kolstad (USA Computing Olympiad)
18. Rujia Liu (Universidad de Tsinghua, China)
19. Shahriar Manzoor (Southeast University, Bangladesh)
20. Sohel Hafiz (University of Texas at San Antonio, Estados Unidos)
21. Soo Yuen Jien (Universidad Nacional de Singapur, Singapur)
22. Tan Sun Teck (Universidad Nacional de Singapur, Singapur)
23. TopCoder, Inc (el problema Prime Pairs de la sección 4.7.4)

Sin embargo, debido a que hemos tratado cientos ( $\approx 1675$ ) de problemas a lo largo del libro, ha resultado imposible contactar con todos los autores. Si eres uno de ellos, o conoces al autor de alguno de los problemas no acreditados, te rogamos que nos lo comuniques. La lista más actualizada está en: <https://sites.google.com/site/stevenhalim/home/credits>.

En la siguiente fotografía aparecemos junto a algunos de los autores que hemos podido conocer en persona:



---

## Bibliografía

- [1] Ahmed Shamsul Arefin. *Art of Programming Contest (de la antigua web de Steven)*. Gyankosh Prokashoni (disponible en línea), 2006.
- [2] Wolfgang W. Bein, Mordecai J. Golin, Lawrence L. Larmore y Yan Zhang. The Knuth-Yao Quadrangle-Inequality Speedup is a Consequence of Total-Monotonicity. *ACM Transactions on Algorithms*, 6 (1):17, 2009.
- [3] Richard Peirce Brent. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics*, 20 (2):176-184, 1980.
- [4] Brilliant. Brilliant.  
<https://brilliant.org/>.
- [5] Frank Carrano. *Data Abstraction and Problem Solving with C++: Walls and Mirrors*. Addison Wesley, quinta edición, 2006.
- [6] Yoeng-jin Chu y Tseng-hong Liu. On the Shortest Arborescence of a Directed Graph. *Science Sinica*, 14:1396-1400, 1965.
- [7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Cliff Stein. *Introduction to Algorithms*. MIT Press, segunda edición, 2001.
- [8] Sanjoy Dasgupta, Christos Papadimitriou y Umesh Vazirani. *Algorithms*. McGraw Hill, 2008.
- [9] Mark de Berg, Marc van Kreveld, Mark Overmars y Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, segunda edición, 2000.
- [10] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269-271, 1959.
- [11] Yefim Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii nauk SSSR*, 11:1277-1280, 1970.
- [12] Adam Drozdek. *Data structures and algorithms in Java*. Cengage Learning, tercera edición, 2008.
- [13] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449-467, 1965.
- [14] Jack Edmonds y Richard Manning Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19 (2):248-264, 1972.
- [15] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks-Cole, cuarta edición, 2010.
- [16] Fabian Ernst, Jeroen Moelands y Seppo Pieterse. Teamwork in Prog Contests:  $3 * 1 = 4$ . <http://xrds.acm.org/article.cfm?aid=332139>.

- [17] Project Euler. Project Euler.  
<http://projecteuler.net/>.
- [18] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24 (3):327-336, 1994.
- [19] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5 (6):345, 1962.
- [20] Michal Forišek. IOI Syllabus.  
<http://people.ksp.sk/~misof/oi-syllabus/oi-syllabus-2009.pdf>.
- [21] Michal Forišek. The difficulty of programming contests increases. En *International Conference on Informatics in Secondary Schools*, 2010.
- [22] William Henry. Gates y Christos Papadimitriou. Bounds for Sorting by Prefix Reversal. *Discrete Mathematics*, 27:47-57, 1979.
- [23] Felix Halim, Roland Hock Chuan Yap y Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. En *ICDCS*, 2011.
- [24] Steven Halim y Felix Halim. Competitive Programming in National University of Singapore. En *A new learning paradigm: competition supported by technology*. Ediciones Sello Editorial S.L., 2010.
- [25] Steven Halim, Roland Hock Chuan Yap y Felix Halim. Engineering SLS for the Low Autocorrelation Binary Sequence Problem. En *Constraint Programming*, páginas 640-645, 2008.
- [26] Steven Halim, Roland Hock Chuan Yap y Hoong Chuin Lau. An Integrated White+Black Box Approach for Designing & Tuning SLS. En *Constraint Programming*, páginas 332-347, 2007.
- [27] Steven Halim, Koh Zi Chun, Loh Victor Bo Huai y Felix Halim. Learning Algorithms with Unified and Interactive Visualization. *Olympiad in Informatics*, 6:53-68, 2012.
- [28] John Edward Hopcroft y Richard Manning Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2 (4):225-231, 1973.
- [29] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. Tesis doctoral, CWI y University of Amsterdam, 2010.
- [30] TopCoder Inc. Algorithm Tutorials.  
[http://www.topcoder.com/tc?d1=tutorials&d2=alg\\_index&module=Static](http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static).
- [31] TopCoder Inc. PrimePairs. Copyright 2009 TopCoder, Inc. All rights reserved.  
[http://www.topcoder.com/stat?c=problem\\_statement&pm=10187&rd=13742](http://www.topcoder.com/stat?c=problem_statement&pm=10187&rd=13742).
- [32] TopCoder Inc. Single Round Match (SRM).  
<http://www.topcoder.com/tc>.
- [33] Competitive Learning Institute. ICPC Live Archive.  
<http://icpcarchive.ecs.baylor.edu>.
- [34] IOI. International Olympiad in Informatics.  
<http://ioinformatics.org>.
- [35] Giuseppe F. Italiano, Luigi Laura y Federico Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. *Combinatorial Optimization and Applications*, 6508:157-169, 2010.

- [36] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5 (11):558-562, 1962.
- [37] Juha Kärkkäinen, Giovanni Manzini y Simon J. Puglisi. Permuted Longest-Common-Prefix Array. En *CPM, LNCS 5577*, páginas 181-192, 2009.
- [38] Jon Kleinberg y Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.
- [39] Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83-97, 1955.
- [40] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 2002.
- [41] Rujia Liu. *Algorithm Contests for Beginners (en chino)*. Tsinghua University Press, 2009.
- [42] Rujia Liu y Liang Huang. *The Art of Algorithms and Programming Contests (en chino)*. Tsinghua University Press, 2003.
- [43] Udi Manbers y Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22 (5):935-948, 1993.
- [44] Gary Lee Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13 (3):300-317, 1976.
- [45] James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32-38, 1957.
- [46] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.
- [47] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, segunda edición, 1998.
- [48] Institute of Mathematics y Lithuania Informatics. Olympiads in Informatics. [http://www.mii.lt/olympiads\\_in\\_informatics/](http://www.mii.lt/olympiads_in_informatics/).
- [49] University of Valladolid. Online Judge. <http://onlinejudge.org>.
- [50] USA Computing Olympiad. USACO Training Program Gateway. <http://train.usaco.org/usacogate>.
- [51] David Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33 (3):231-234, 2004.
- [52] John M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15 (3):331-334, 1975.
- [53] George Pólya. *How to Solve It*. Princeton University Press, segunda edición, 1957.
- [54] Janet Prichard y Frank Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*. Addison Wesley, tercera edición, 2010.
- [55] Michael Oser Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12 (1):128-138, 1980.
- [56] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, séptima edición, 2012.
- [57] Kenneth H. Rosen. *Elementary Number Theory and its Applications*. Addison Wesley Longman, cuarta edición, 2000.
- [58] Robert Sedgewick. *Algorithms in C++, Part 1-5*. Addison Wesley, tercera edición, 2002.

- [59] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2008.
- [60] Steven S. Skiena y Miguel A. Revilla. *Programming Challenges*. Springer, 2003.
- [61] SPOJ. Sphere Online Judge.  
<http://www.spoj.pl/>.
- [62] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press (Taylor & Francis Group), primera edición, 2010.
- [63] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1 (2):146-160, 1972.
- [64] Jeffrey Trevers y Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, 32 (4):425-443, 1969.
- [65] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14 (3):249-260, 1995.
- [66] Baylor University. ACM International Collegiate Programming Contest.  
<http://icpc.baylor.edu/icpc>.
- [67] Tom Verhoeff. 20 Years of IOI Competition Tasks. *Olympiads in Informatics*, 3:149-166, 2009.
- [68] Adrian Vladu y Cosmin Negrușeri. Suffix arrays - a programming contest approach. En *GInfo*, 2005.
- [69] Henry S. Warren. *Hacker's Delight*. Pearson, primera edición, 2003.
- [70] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9 (1):11-12, 1962.
- [71] Wikipedia. The Free Encyclopedia.  
<http://wikipedia.org>.

---

# Índice alfabético

2-SAT, 394

A\*, 364

A\* de profundidad iterativa, 366

Álgebra lineal, 404

Árbol, 212

Ancestro común mínimo, 419

APSP, 213

Diámetro, 213

Puntos de articulación y puentes, 212

Recorrido de un árbol, 212

SSSP, 212

Árbol binario indexado, 70

Árbol de búsqueda binaria, 49

Árbol de decisión, 270

Árbol de expansión, 403

Árbol de expansión mínimo, 163

Árbol de expansión ‘máximo’, 167

‘Bosque de expansión’ mínimo, 167

Segundo mejor, 168

Subgrafo de expansión ‘mínimo’, 167

Árbol de Fenwick, 70

Árbol de juego, véase Árbol de decisión

Árbol de matrices de Kirchhoff, teorema, 163

Árbol de segmentos, 65

Adelson-Velskii, Georgii, 65

Alternancia de camino, algoritmo de, 217

Analizador sintáctico descendente

recursivo, 281

Ancestro común mínimo, 213, 419

Anchura, búsqueda en, véase BFS

Aritmética modular, 260

Arista cortada, véase Puentes

Array, 39

atan, 351

atan2, 351

Aumento de camino, algoritmo de, 217

Backtracking, 82, 87, 111, 145, 289

Máscara de bits, 354

Backtracking recursivo, véase Backtracking

Backus-Naur, notación de, 281

Búsqueda, 40

Búsqueda binaria, 40, 98, 308

Búsqueda binaria de la respuesta, 378

Búsqueda completa, 82

Búsqueda de cadenas, véase Cadenas, coincidencia de

Búsqueda de ciclos de Floyd, algoritmo, 267

Base numérica, 231

Bayer, Rudolf, 65

Bellman Ford, algoritmo de, 180

Bellman, Richard Ernest, 172, 180

Berge, Claude, 221

BFS, 151, 173, 196, 360, 362

Bibliotecas, 37

Bidireccional, búsqueda, 362

BigInteger, véase BigInteger, clase de Java

BigInteger, clase de Java, 237

Conversión de bases, 239

Máximo común divisor, 241

modPow, 241

Prueba de primalidad (probabilística), 240

Binet, fórmula de, 245

Binet, Jacques P. M., 251

Bioinformática, véase Cadenas, procesamiento de

Bipartito, emparejamiento, 407

Bisección, método de, 100, 379

bitset, 41, 253

Boole, George, 65

- Brent, Richard P., 256, 264  
Burbuja, ordenación de, 40, 414  
Buscar la respuesta de forma binaria, 101
- Cadenas de matrices, multiplicación de, 370, 422  
Cadenas, alineación de, 290  
Cadenas, coincidencia de, 285  
Cadenas, procesamiento de, 277  
Círculos, 329  
Calculadora posfija, 438  
Cambio de monedas, 104, 128  
Caminos de arista-disjuntos máximos, 413  
Caminos independientes máximos, 413  
Caminos más cortos de origen único, véase SSSP  
Caminos más cortos entre todos los pares, 184, 213  
Ciclo más barato/negativo, 189  
Clausura transitiva, 188  
Diámetro de un grafo, 190  
Impresión de los caminos más cortos, 188  
*Minimax y maximin*, 189  
SCC de un grafo dirigido, 190
- Caminos más largos en un DAG, 204  
Cartero chino, problema del, 399  
Casilleros, ordenación por, 40  
Catalan, Eugène Charles, 251  
Catalan, números de, 246  
Cayley, fórmula de, 403  
Ciclos, búsqueda de, 267  
Cifrado, 280  
Clausura transitiva, 188  
Cobertura de caminos mínima en un DAG, 431  
Cobertura de intervalos, problema de, 107  
Cobertura de vértices, 208, 396  
Cobertura de vértices mínima, 208, 216, 396  
Coeficientes binomiales, 246  
Cola, 44  
Cola de dos extremos, 44  
Cola de prioridad, 50, 176  
Combinatoria, 244  
Componentes conexos, 147  
Componentes fuertemente conexos, véase SCC  
Conjunto dominante, 216
- Conjunto independiente, 358, 429  
Conjunto independiente máximo, 216, 358  
Conjunto independiente ponderado máximo, 429  
Conjuntos disjuntos para unión-buscar, 61  
Consulta de rango mínimo, 65  
Consulta del rango mínimo, 419, 450  
Conteo de caminos en un DAG, 205  
Conversión de bases, 239  
Conversión infija a posfija, 437  
Corte mínimo, 199  
Coseno, problema del, 333  
Criba modificada, 259  
Criptografía, 280  
Cuadrado mágico, 421  
Cuadriláteros, 334  
Cuentas, ordenación por, 40, 448  
*cutPolygon*, 342, 396
- De la Loubère, método de, 421  
*Deque*, 445  
Desarreglo, 265, 403  
Descomposición, 378  
Desigualdad del cuadrángulo, 135  
Diámetro  
    Árbol, 213  
    Grafo, 190  
Difusión, relleno por, 148  
Dijkstra, algoritmo de, 176  
Dijkstra, Edsger Wybe, 172, 176  
Dinic, algoritmo de, 402  
Diofanto de Alejandría, 252, 260  
Distancia de círculo máximo, 411  
Distancia de edición, 290  
Divide y vencerás, 98, 254, 255, 308, 401, 414, 442  
Divisores  
    Número de, 257  
    Suma de, 258
- Ecuación diofántica lineal, 260  
Edmonds Karp, algoritmo de, 195  
Edmonds, Jack R., 193, 195, 409  
Eliminación gausiana, 404  
Eliminación perezosa, 177  
Emparejamiento, 215, 407  
Emparejamiento bipartito, 215  
Emparejamiento de Edmonds, algoritmo de, 409

- Encuentro en el medio, 362  
Envolvente convexa, 343  
Eratóstenes de Cirene, 251, 253  
Eratóstenes, criba de, 253, 259  
Erdős Gallai, teorema de, 403  
Esferas, 411  
Estadísticos de orden, 441  
Estado-espacio, búsqueda, 360  
Estrategia *minimax*, 270  
Estructuras de datos, 37  
Euclídeo extendido, véase Euclídeo, algoritmo  
Euclídeo, algoritmo  
    Euclídeo extendido, 260  
Euclides de Alejandría, 254, 337  
Euclides, algoritmo de, 254  
Euler, fórmula de, 403  
Euler, Fi de, 258  
Euler, Leonhard, 252, 258  
Exponenciación modular, véase Potencia modular  
Expresiones regulares (*regex*), 281  
  
Factores primos, 255–257, 436  
    Número de, 257  
    Número de diferentes, 257  
    Suma de, 257  
Factorial, 255  
Fenwick, Peter M, 75  
Fibonacci, Leonardo, 245, 251  
Fibonacci, sucesión de, 245  
Flores, 409  
Floyd Warshall, algoritmo de, 184  
Floyd, Robert W, 184  
Floyd, Robert W., 193  
Flujo (máximo) de coste mínimo, 430  
Flujo de red, 193  
    Caminos de arista-disjuntos máximos, 413  
    Caminos independientes máximos, 413  
    Capacidad de los vértices, 200  
    Corte mínimo, 199  
    Flujo (máximo) de coste mínimo, 430  
    Multiorigen/multidesagüe, 200  
Flujo máximo, véase Flujo de red  
Ford Fulkerson, método de, 193  
Ford Jr, Lester Randolph, 172, 180, 193  
Forma normal conjuntiva, 394  
Fuerza bruta, véase Búsqueda completa  
  
Fulkerson, Delbert Ray, 172, 193  
  
Galería de arte, problema de la, 396  
Geometría, 319  
Geometría computacional, véase Geometría  
Goldbach, Christian, 252  
Goldbach, conjetura de, 261  
Grafo  
    Estructuras de datos, 57  
Grafo acíclico dirigido, 203  
    Caminos más cortos, 204  
    Caminos más largos, 204  
    Cobertura de caminos mínima, 431  
    Conteo de caminos, 205  
    Grafo general a DAG, 206  
Grafo bipartito, 215  
    Cobertura de caminos mínima en un DAG, 431  
    Cobertura de vértices mínima, 216  
    Comprobación, 151  
    Conjunto dominante, 216  
    Conjunto independiente máximo, 216  
    Emparejamiento bipartito de cardinalidad máxima, 215  
Grafo bipartito completo, 403, 412  
Grafo completo, 400  
Grafo euleriano, 214, 399  
    Comprobación de grafo euleriano, 214  
    Escribir un ciclo euleriano, 214  
Grafo planar, 403  
Grafos, 143  
Grafos especiales, 203  
Grafos, emparejamiento de, 407  
Grafos, modelado de, 145, 149, 178, 197, 200, 217  
Graham, método de, 343  
Graham, Ronald Lewis, 337, 343  
  
Húngaro, algoritmo, 408  
Herón de Alejandría, 337  
Herón, fórmula de, 331  
Hopcroft, John Edward, 155, 171  
  
ICPC, 1  
Índice de inversión, 414  
inPolygon, 340  
Inserción, ordenación por, 40  
Inspección de rutas, problema de, 399  
IOI, 1

- IOI 2003 - Trail Maintenance, 171  
IOI 2008 - Type Printer, 314  
IOI 2009 - Garage, 24  
IOI 2009 - Mecho, 387  
IOI 2009 - POI, 24  
IOI 2010 - Cluedo, 24  
IOI 2010 - Memory, 24  
IOI 2010 - Quality of Living, 104  
IOI 2011 - Alphabets, 236  
IOI 2011 - Crocodile, 183  
IOI 2011 - Elephants, 110  
IOI 2011 - Hottest, 447  
IOI 2011 - Pigeons, 48  
IOI 2011 - Race, 104  
IOI 2011 - Ricehub, 447  
IOI 2011 - Tropical Garden, 161  
IOI 2012 - Tourist Plan, 448  
`isConvex`, 339, 396
- Jarník, Vojtěch, 172  
Josefo, problema de, 415  
Juego de suma cero, 270  
Juego del *nim*, 273
- Kadane, algoritmo de, 122  
Kadane, Jay, 122  
Karp, Richard Manning, 193, 195  
Knuth, Donald Ervin, 280  
Knuth-Morris-Pratt, algoritmo de, 286  
Knuth-Yao, aceleración de DP de, 135  
König, Dénes, 221  
Kosaraju, algoritmo de, 158, 395, 417  
Kosaraju, Sambasiva Rao, 158  
Kruskal, algoritmo de, 164  
Kruskal, Joseph Bernard, 164, 172  
Kuhn Munkres, algoritmo de, 408
- LA 2512 - Art Gallery, 396  
LA 3617 - How I Mathematician ..., 396  
Línea de barrido, 401  
Líneas, 323  
Landis, Evgenii Mikhailovich, 65  
Levenshtein, distancia de, 290  
Lista enlazada, 43  
Live Archive, 18
- Máscara de bits, 41, 129, 354, 369  
Máximo común divisor, 241, 254  
Mínimo común múltiplo, 254
- Manber, Udi, 296  
Matemáticas, 229, 383  
Matrices, potencia de, 424  
MCBM, véase Emparejamiento bipartito  
Mezcla, ordenación de, 40  
Mezcla, ordenación por, 414  
Miller, Gary Lee, 244  
Miller-Rabin, algoritmo de, 240  
*Minimax y maximin*, 189  
Mochila (0-1), 127  
Montículo, 50  
Montículos, ordenación por, 40, 51  
Monty Hall, problema de, 265  
Morris, James Hiram, 280  
Moser, círculo de, 403  
Movimientos del caballo, 416  
Myers, Gene, 296
- Número de cuerda, algoritmo del, 340  
Números compuestos, 255  
Números primos, 252
- Criba de Eratóstenes, 253
  - Factores primos, 255
  - Funciones que implican factores primos, 257
  - Prueba de primalidad, 252
  - Trabajo con factores primos, 256
- Números romanos, 439  
Needleman, Saul B., 280
- Ordenación, 40, 51
- Ordenación de burbuja, 414
  - Ordenación por cuentas, 448
  - Ordenación por mezcla, 414
- Palomar, principio del, 106  
Par más cercano, problema del, 401  
Paréntesis, emparejamiento de, 398  
Partida óptima, véase Partida perfecta  
Partida perfecta, 270  
Pascal, Blaise, 251  
Pascal, triángulo de, 246  
PERT, 205  
Peso negativo, ciclo de, 180, 189  
Pick, Georg Alexander, 403  
Pick, teorema de, 403  
Pila, 44, 398, 437  
Pisano, periodo de, 245, 250  
Pitágoras de Samos, 337

Pitágoras, teorema de, 333  
Polígono  
    Área, 339  
    area, 339  
    cutPolygon, 342, 396  
    Envolvente convexa, 343  
    inPolygon, 340  
    isConvex, 339, 396  
    Perímetro, 338  
    perimeter, 338  
    Representación, 338  
Polinomio, 231  
Pollard, John, 256, 264  
Potencia modular, 241  
Pratt, Vaughan Ronald, 280  
Prefijo común más largo, 310  
Prim, algoritmo de, 165  
Prim, Robert Clay, 165, 172  
Primos gemelos, 261  
Principio del palomar, 236  
Problema del viajante, 129  
Procesamiento previo, 450  
Producto vectorial, 326  
Profunda limitada, búsqueda de, 289  
Profundidad iterativa, búsqueda de, 366  
Profundidad limitada, búsqueda de, 365  
Profundidad, búsqueda en, 144  
Programación competitiva, 1  
Programación dinámica, 111, 204, 246,  
    290, 368, 450  
    Máscara de bits, 369  
Programación dinámica en un árbol, 208  
Progresión aritmética, 231  
Progresión geométrica, 231  
Prueba de giro a la izquierda, 326  
Prueba de primalidad, 240, 252, 436  
Puentes, 154, 212  
Puntos, 321  
Puntos de articulación, 154, 212  
*Quick*, ordenación, 40  
Rabin, Michael Oser, 244  
*Radix*, ordenación, 40  
Razón áurea, 245  
Rejilla, 230  
rho de Pollard, algoritmo, 436  
Satisfacción, 394  
SCC, 158, 190, 382, 394  
Secuencia, 230  
Segundo mejor árbol de expansión, 168  
Selección, ordenación por, 40  
Selección, problema de, 441  
Senos, teorema de los, 333  
Siamés, método, 421  
Sistema numérico, 230  
Smith, Temple F., 280  
SPOJ 0101 - Fishmonger, 222  
SPOJ 0739 - The Moronic Cowmpouter,  
    236  
SPOJ 3944 - Bee Walk, 234  
SPOJ 6409 - Suffix Array, 314  
SSSP, 212, 360, 381  
    Ciclo de peso negativo, 180  
    Detección de ciclos negativos, 180  
    No ponderado, 173  
    Ponderado, 176  
String de Java (expresiones regulares), 281  
Subcadena común más larga, 300, 312  
Subcadena repetida más larga, 299, 312  
Subsecuencia común más larga, 293  
Subsecuencia creciente máxima, 124  
Sufijos, 296  
Sufijos, árbol de, 297  
    Aplicaciones  
        Coincidencia de cadenas, 299  
        Subcadena común más larga, 300  
        Subcadena repetida más larga, 299  
Sufijos, *array* de, 302  
    Aplicaciones  
        Coincidencia de cadenas, 308  
        Prefijo común más largo, 310  
        Subcadena común más larga, 312  
        Subcadena repetida más larga, 312  
        Construcción en  $O(n \log n)$ , 306  
        Construcción en  $O(n^2 \log n)$ , 304  
Sufijos, *trie* de, 296  
Suma de rangos  
    Bidimensional máxima, 123  
    Unidimensional máxima, 122  
Suma de subconjuntos, 127  
Tabla de direccionamiento directo, 51  
Tabla de *hash*, 51  
Tabla dispersa, 450  
Tarjan, Robert Endre, 155, 158, 171, 395  
Teoría de juegos, 270

Teoría de números, 252  
Teoría de probabilidad, 265  
Terna pitagórica, 333  
Top Coder Open 2009: Prime Pairs, 223  
TopCoder, 18  
Topológico, orden, 149  
Torres de Hanoi, 452  
Triángulos, 331

uHunt, 457  
USACO, 18  
UVa, 18  
UVa 00100 - The 3n + 1 problem, 232  
UVa 00101 - The Blocks Problem, 47  
UVa 00102 - Ecological Bin Packing, 94  
UVa 00103 - Stacking Boxes, 221  
UVa 00104 - Arbitrage \*, 192  
UVa 00105 - The Skyline Problem, 95  
UVa 00106 - Fermat vs. Phytagoras, 262  
UVa 00107 - The Cat in the Hat, 235  
UVa 00108 - Maximum Sum, 123  
UVa 00108 - Maximum Sum \*, 136  
UVa 00109 - Scud Busters, 348  
UVa 00110 - Meta-loopless sort, 284  
UVa 00111 - History Grading, 137  
UVa 00112 - Tree Summing, 222  
UVa 00113 - Power Of Cryptography, 235  
UVa 00114 - Simulation Wizardry, 27  
UVa 00115 - Climbing Trees, 222  
UVa 00116 - Unidirectional TSP, 138  
UVa 00117 - The Postal Worker ..., 222  
UVa 00118 - Mutant Flatworld Explorers, 161  
UVa 00119 - Greedy Gift Givers, 24  
UVa 00120 - Stacks Of Flapjacks \*, 435  
UVa 00121 - Pipe Fitters, 336  
UVa 00122 - Trees on the level, 222  
UVa 00123 - Searching Quickly, 47  
UVa 00124 - Following Orders, 162  
UVa 00125 - Numbering Paths, 192  
UVa 00126 - The Errant Physicist, 235  
UVa 00127 - "Accordian" Patience, 48  
UVa 00128 - Software CRC, 263  
UVa 00129 - Krypton Factor, 97  
UVa 00130 - Roman Roulette, 416  
UVa 00131 - The Psychic Poker Player, 367  
UVa 00133 - The Dole Queue, 416  
UVa 00136 - Ugly Numbers, 234  
UVa 00137 - Polygons, 348

UVa 00138 - Street Numbers, 234  
UVa 00139 - Telephone Tangles, 29  
UVa 00140 - Bandwidth, 96  
UVa 00141 - The Spot Game, 28  
UVa 00142 - Mouse Clicks, 389  
UVa 00143 - Orchard Trees, 336  
UVa 00144 - Student Grants, 30  
UVa 00145 - Gondwanaland Telecom, 29  
UVa 00146 - ID Codes \*, 47  
UVa 00147 - Dollars, 137  
UVa 00148 - Anagram Checker, 28  
UVa 00151 - Power Crisis, 416  
UVa 00152 - Tree's a Crowd, 335  
UVa 00153 - Permalex, 285  
UVa 00154 - Recycling, 95  
UVa 00155 - All Squares, 337  
UVa 00156 - Ananagram \*, 28  
UVa 00159 - Word Crosses, 284  
UVa 00160 - Factors and Factorials, 262  
UVa 00161 - Traffic Lights \*, 28  
UVa 00162 - Beggar My Neighbour, 27  
UVa 00164 - String Computer, 295  
UVa 00165 - Stamps, 97  
UVa 00166 - Making Change, 137  
UVa 00167 - The Sultan Successor, 96  
UVa 00168 - Theseus and the ..., 161  
UVa 00170 - Clock Patience, 29  
UVa 00183 - Bit Maps \*, 104  
UVa 00184 - Laser Lines, 389  
UVa 00186 - Trip Routing, 192  
UVa 00187 - Transaction Processing, 28  
UVa 00188 - Perfect Hash, 95  
UVa 00190 - Circle Through Three ..., 336  
UVa 00191 - Intersection, 335  
UVa 00193 - Graph Coloring \*, 97  
UVa 00195 - Anagram \*, 28  
UVa 00196 - Spreadsheet, 138  
UVa 00200 - Rare Order, 162  
UVa 00201 - Square, 389  
UVa 00202 - Repeating Decimals, 270  
UVa 00208 - Firetruck, 97  
UVa 00213 - Message ..., 282  
UVa 00214 - Code Generation, 30  
UVa 00216 - Getting in Line \*, 138  
UVa 00218 - Moth Eradication, 348  
UVa 00220 - Othello, 28  
UVa 00222 - Budget Travel, 97  
UVa 00227 - Puzzle, 28

- UVa 00230 - Borrowers, 46  
UVa 00231 - Testing the Catcher, 137  
UVa 00232 - Crossword Answers, 28  
UVa 00234 - Switching Channels, 96  
UVa 00245 - Uncompress, 282  
UVa 00247 - Calling Circles \*, 163, 190  
UVa 00253 - Cube painting, 95  
UVa 00255 - Correct Move, 27  
UVa 00256 - Quirksome Squares, 94  
UVa 00257 - Palinwords, 295  
UVa 00259 - Software Allocation, 198  
UVa 00259 - Software Allocation \*, 202  
UVa 00260 - Il Gioco dell'X, 161  
UVa 00263 - Number Chains, 285  
UVa 00264 - Count on Cantor \*, 234  
UVa 00270 - Lining Up, 389  
UVa 00271 - Simply Syntax, 283  
UVa 00272 - TEX Quotes, 23  
UVa 00273 - Jack Straw, 388  
UVa 00274 - Cat and Mouse, 192  
UVa 00275 - Expanding Fractions, 270  
UVa 00276 - Egyptian Multiplication, 236  
UVa 00278 - Chess \*, 27  
UVa 00280 - Vertex, 161  
UVa 00290 - Palindroms ↔ ..., 243  
UVa 00291 - The House of Santa ..., 222  
UVa 00294 - Divisors \*, 263  
UVa 00295 - Fatman \*, 390  
UVa 00296 - Safebreaker, 95  
UVa 00297 - Quadtrees, 76  
UVa 00299 - Train Swapping, 415  
UVa 00300 - Maya Calendar, 29  
UVa 00301 - Transportation, 97  
UVa 00305 - Joseph, 416  
UVa 00306 - Cipher, 282  
UVa 00311 - Packets, 111  
UVa 00314 - Robot \*, 182  
UVa 00315 - Network \*, 163  
UVa 00318 - Domino Effect, 161  
UVa 00320 - Border, 284  
UVa 00321 - The New Villa, 367  
UVa 00324 - Factorial Frequencies \*, 262  
UVa 00325 - Identifying Legal ... \*, 284  
UVa 00326 - Extrapolation using a ..., 250  
UVa 00327 - Evaluating Simple C ..., 283  
UVa 00330 - Inventory Maintenance, 284  
UVa 00331 - Mapping the Swaps, 97  
UVa 00332 - Rational Numbers from ..., 262  
UVa 00333 - Recognizing Good ISBNs, 29  
UVa 00334 - Identifying Concurrent ... \*, 192  
UVa 00335 - Processing MX Records, 30  
UVa 00336 - A Node Too Far, 182  
UVa 00337 - Interpreting Control ..., 30  
UVa 00338 - Long Multiplication, 284  
UVa 00339 - SameGame Simulation, 28  
UVa 00340 - Master-Mind Hints, 27  
UVa 00341 - Non-Stop Travel, 192  
UVa 00343 - What Base Is This? \*, 243  
UVa 00344 - Roman Digititis \*, 441  
UVa 00346 - Getting Chorded, 29  
UVa 00347 - Run, Run, Runaround ..., 95  
UVa 00348 - Optimal Array Mult ... \*, 424  
UVa 00349 - Transferable Voting (II), 30  
UVa 00350 - Pseudo-Random Numbers \*, 270  
UVa 00352 - The Seasonal War, 161  
UVa 00353 - Pesky Palindromes, 28  
UVa 00355 - The Bases Are Loaded, 243  
UVa 00356 - Square Pegs And Round ..., 389  
UVa 00357 - Let Me Count The Ways \*, 138  
UVa 00361 - Cops and Robbers, 348  
UVa 00362 - 18,000 Seconds Remaining, 28  
UVa 00369 - Combinations, 250  
UVa 00371 - Ackermann Functions, 232  
UVa 00373 - Romulan Spelling, 284  
UVa 00374 - Big Mod \*, 263  
UVa 00375 - Inscribed Circles and ..., 336  
UVa 00377 - Cowculations \*, 236  
UVa 00378 - Intersecting Lines, 335  
UVa 00379 - HI-Q, 28  
UVa 00380 - Call Forwarding, 96  
UVa 00381 - Making the Grade, 30  
UVa 00382 - Perfection \*, 232  
UVa 00383 - Shipping Routes, 182  
UVa 00384 - Slurphys, 284  
UVa 00386 - Perfect Cubes, 95  
UVa 00388 - Galactic Import, 182  
UVa 00389 - Basically Speaking \*, 243  
UVa 00391 - Mark-up, 283  
UVa 00392 - Polynomial Showdown, 235  
UVa 00394 - Mapmaker, 46  
UVa 00397 - Equation Elation, 283

- UVa 00400 - Unix ls, 47  
UVa 00401 - Palindromes \*, 28  
UVa 00402 - M\*A\*S\*H, 416  
UVa 00403 - Postscript \*, 29  
UVa 00405 - Message Routing, 30  
UVa 00406 - Prime Cuts, 261  
UVa 00408 - Uniform Generator, 262  
UVa 00409 - Excuses, Excuses, 285  
UVa 00410 - Station Balance, 105, 110  
UVa 00412 - Pi, 262  
UVa 00413 - Up and Down Sequences, 234  
UVa 00414 - Machined Surfaces, 46  
UVa 00416 - LED Test \*, 97  
UVa 00417 - Word Index, 56  
UVa 00422 - Word Search Wonder \*, 290  
UVa 00423 - MPI Maelstrom, 192  
UVa 00424 - Integer Inquiry, 243  
UVa 00426 - Fifth Bank of ..., 284  
UVa 00429 - Word Transformation \*, 182  
UVa 00433 - Bank (Not Quite O.C.R.), 97  
UVa 00434 - Matty's Blocks, 47  
UVa 00435 - Block Voting, 96  
UVa 00436 - Arbitrage (II), 192  
UVa 00436 - Arbitrage II, 190  
UVa 00437 - The Tower of Babylon, 137  
UVa 00438 - The Circumference of ..., 336  
UVa 00439 - Knight Moves \*, 417  
UVa 00440 - Eeny Meeny Moo, 416  
UVa 00441 - Lotto, 84  
UVa 00441 - Lotto \*, 95  
UVa 00442 - Matrix Chain Multiplication, 283  
UVa 00443 - Humble Numbers \*, 234  
UVa 00444 - Encoder and Decoder, 282  
UVa 00445 - Marvelous Mazes, 284  
UVa 00446 - Kibbles 'n' Bits 'n' Bits ..., 243  
UVa 00447 - Population Explosion, 29  
UVa 00448 - OOPS, 29  
UVa 00449 - Majoring in Scales, 29  
UVa 00450 - Little Black Book, 47  
UVa 00452 - Project Scheduling, 205  
UVa 00452 - Project Scheduling \*, 221  
UVa 00454 - Anagrams \*, 28  
UVa 00455 - Periodic String, 290  
UVa 00457 - Linear Cellular Automata, 29  
UVa 00458 - The Decoder, 282  
UVa 00459 - Graph Connectivity, 148, 162  
UVa 00460 - Overlapping Rectangles, 337  
UVa 00462 - Bridge Hand Evaluator \*, 27  
UVa 00464 - Sentence/Phrase Generator, 284  
UVa 00465 - Overflow, 243  
UVa 00466 - Mirror Mirror, 47  
UVa 00467 - Synching Signals, 46  
UVa 00468 - Key to Success, 282  
UVa 00469 - Wetlands of Florida, 149, 162  
UVa 00471 - Magic Numbers, 95  
UVa 00473 - Raucous Rockers, 377  
UVa 00474 - Heads Tails Probability, 235  
UVa 00476 - Points in Figures: ... \*, 337  
UVa 00477 - Points in Figures: ..., 337  
UVa 00478 - Points in Figures: ..., 348  
UVa 00481 - What Goes Up? \*, 137  
UVa 00482 - Permutation Arrays, 46  
UVa 00483 - Word Scramble, 282  
UVa 00484 - The Department ..., 56  
UVa 00485 - Pascal Triangle of Death, 250  
UVa 00486 - English-Number Translator, 283  
UVa 00487 - Boggle Blitz, 97  
UVa 00488 - Triangle Wave \*, 284  
UVa 00489 - Hangman Judge \*, 27  
UVa 00490 - Rotating Sentences, 284  
UVa 00492 - Pig Latin, 282  
UVa 00493 - Rational Spiral, 233  
UVa 00494 - Kindergarten ... \*, 284  
UVa 00495 - Fibonacci Freeze, 250  
UVa 00496 - Simply Subsets, 236  
UVa 00497 - Strategic Defense Initiative, 137  
UVa 00498 - Polly the Polynomial \*, 235  
UVa 00499 - What's The Frequency ..., 283  
UVa 00501 - Black Box, 57  
UVa 00507 - Jill Rides Again, 122, 136  
UVa 00514 - Rails \*, 48  
UVa 00516 - Prime Land \*, 262  
UVa 00521 - Gossiping, 388  
UVa 00524 - Prime Ring Problem \*, 97  
UVa 00526 - String Distance ... \*, 295  
UVa 00530 - Binomial Showdown, 250  
UVa 00531 - Compromise, 295  
UVa 00532 - Dungeon Master, 182  
UVa 00534 - Frogger, 171  
UVa 00535 - Globetrotter, 412  
UVa 00536 - Tree Recovery, 222

- UVa 00537 - Artificial Intelligence?, 283  
UVa 00538 - Balancing Bank Accounts, 29  
UVa 00539 - The Settlers ..., 96  
UVa 00540 - Team Queue, 48  
UVa 00541 - Error Correction, 47  
UVa 00542 - France '98, 266  
UVa 00543 - Goldbach's Conjecture \*, 261  
UVa 00544 - Heavy Cargo, 171  
UVa 00545 - Heads, 235  
UVa 00547 - DDF, 264  
UVa 00548 - Tree, 222  
UVa 00550 - Multiplying by Rotation, 233  
UVa 00551 - Nesting a Bunch of ... \*, 399  
UVa 00554 - Caesar Cypher \*, 282  
UVa 00555 - Bridge Hands, 27  
UVa 00556 - Amazing \*, 30  
UVa 00558 - Wormholes, 180  
UVa 00558 - Wormholes \*, 183  
UVa 00562 - Dividing Coins, 137  
UVa 00563 - Crimewave \*, 414  
UVa 00565 - Pizza Anyone?, 97  
UVa 00567 - Risk, 192  
UVa 00568 - Just the Facts, 262  
UVa 00570 - Stats, 284  
UVa 00571 - Jugs, 97  
UVa 00572 - Oil Deposits, 162  
UVa 00573 - The Snail \*, 24  
UVa 00574 - Sum It Up \*, 97  
UVa 00575 - Skew Binary \*, 236  
UVa 00576 - Haiku Review, 284  
UVa 00579 - Clock Hands \*, 29  
UVa 00580 - Critical Mass, 250  
UVa 00583 - Prime Factors \*, 262  
UVa 00584 - Bowling \*, 28  
UVa 00587 - There's treasure everywhere,  
                  335  
UVa 00588 - Video Surveillance \*, 396  
UVa 00590 - Always on the Run, 222  
UVa 00591 - Box of Bricks, 46  
UVa 00594 - One Little, Two Little ..., 48  
UVa 00596 - The Incredible Hull, 348  
UVa 00598 - Bundling Newspaper, 97  
UVa 00599 - The Forrest for the Trees \*, 75  
UVa 00603 - Parking Lot, 30  
UVa 00604 - The Boggle Game, 290  
UVa 00607 - Scheduling Lectures, 376  
UVa 00608 - Counterfeit Dollar \*, 29  
UVa 00610 - Street Directions, 163  
UVa 00612 - DNA Sorting \*, 415  
UVa 00613 - Numbers That Count, 236  
UVa 00614 - Mapping the Route, 161  
UVa 00615 - Is It A Tree?, 222  
UVa 00616 - Coconuts, Revisited \*, 233  
UVa 00617 - Nonstop Travel, 95  
UVa 00619 - Numerically Speaking, 243  
UVa 00620 - Cellular Structure, 284  
UVa 00621 - Secret Research, 23  
UVa 00622 - Grammar Evaluation \*, 284  
UVa 00623 - 500 (factorial) \*, 262  
UVa 00624 - CD \*, 96  
UVa 00626 - Ecosystem, 95  
UVa 00627 - The Net, 182  
UVa 00628 - Passwords, 96  
UVa 00630 - Anagrams (II), 28  
UVa 00632 - Compression (II), 282  
UVa 00634 - Polygon, 348  
UVa 00636 - Squares, 236  
UVa 00637 - Booklet Printing \*, 28  
UVa 00638 - Finding Rectangles, 389  
UVa 00639 - Don't Get Rooked, 96  
UVa 00640 - Self Numbers, 234  
UVa 00641 - Do the Untwist, 282  
UVa 00642 - Word Amalgamation, 28  
UVa 00644 - Immediate Decodability \*,  
                  285  
UVa 00645 - File Mapping, 284  
UVa 00647 - Chutes and Ladders, 28  
UVa 00651 - Deck, 234  
UVa 00652 - Eight, 368  
UVa 00657 - The Die is Cast, 162  
UVa 00658 - It's not a Bug ..., 368  
UVa 00661 - Blowing Fuses, 24  
UVa 00663 - Sorting Slides, 223  
UVa 00665 - False Coin, 46  
UVa 00668 - Parliament, 111  
UVa 00670 - The Dog Task, 223  
UVa 00671 - Spell Checker, 285  
UVa 00673 - Parentheses Balance \*, 399  
UVa 00674 - Coin Change, 129, 138  
UVa 00677 - All Walks of length "n" ..., 96  
UVa 00679 - Dropping Balls, 103  
UVa 00681 - Convex Hull Finding, 348  
UVa 00686 - Goldbach's Conjecture (II),  
                  261  
UVa 00688 - Mobile Phone Coverage, 389  
UVa 00694 - The Collatz Sequence, 234

- UVa 00696 - How Many Knights \*, 27  
UVa 00697 - Jack and Jill, 233  
UVa 00699 - The Falling Leaves, 222  
UVa 00700 - Date Bugs, 48  
UVa 00701 - Archaeologist's Dilemma \*, 235  
UVa 00702 - The Vindictive Coach, 376  
UVa 00703 - Triple Ties: The ..., 95  
UVa 00706 - LC-Display, 29  
UVa 00710 - The Game, 367  
UVa 00711 - Dividing up, 367  
UVa 00712 - S-Trees, 222  
UVa 00713 - Adding Reversed ... \*, 243  
UVa 00714 - Copying Books, 387  
UVa 00719 - Glass Beads, 314  
UVa 00722 - Lakes, 162  
UVa 00725 - Division, 83, 95  
UVa 00726 - Decode, 282  
UVa 00727 - Equation \*, 439  
UVa 00729 - The Hamming Distance ..., 96  
UVa 00732 - Anagram by Stack \*, 48  
UVa 00735 - Dart-a-Mania \*, 95  
UVa 00736 - Lost in Space, 290  
UVa 00737 - Gleaming the Cubes \*, 337  
UVa 00739 - Soundex Indexing, 282  
UVa 00740 - Baudot Data ..., 282  
UVa 00741 - Burrows Wheeler Decoder, 282  
UVa 00743 - The MTM Machine, 284  
UVa 00748 - Exponentiation, 243  
UVa 00750 - 8 Queens Chess Problem, 87, 96  
UVa 00753 - A Plug for Unix, 223  
UVa 00755 - 487-3279, 46  
UVa 00756 - Biorhythms, 264  
UVa 00758 - The Same Game, 162  
UVa 00759 - The Return of the ..., 441  
UVa 00760 - DNA Sequencing \*, 314  
UVa 00762 - We Ship Cheap, 182  
UVa 00763 - Fibinary Numbers \*, 250  
UVa 00775 - Hamiltonian Cucle, 97  
UVa 00776 - Monkeys in a Regular ..., 162  
UVa 00782 - Countour Painting, 162  
UVa 00784 - Maze Exploration, 162  
UVa 00785 - Grid Colouring, 162  
UVa 00787 - Maximum Sub ... \*, 136  
UVa 00790 - Head Judge Headache, 47  
UVa 00793 - Network Connections \*, 75  
UVa 00795 - Sandorf's Cipher, 282  
UVa 00796 - Critical Links \*, 163  
UVa 00808 - Bee Breeding, 234  
UVa 00811 - The Fortified Forest, 391  
UVa 00812 - Trade on Verweggistan, 376  
UVa 00815 - Flooded \*, 337  
UVa 00820 - Internet Bandwidth \*, 193, 202  
UVa 00821 - Page Hopping \*, 192  
UVa 00824 - Coast Tracker, 161  
UVa 00825 - Walking on the Safe Side, 221  
UVa 00830 - Shark, 30  
UVa 00833 - Water Falls, 335  
UVa 00834 - Continued Fractions, 232  
UVa 00836 - Largest Submatrix, 136  
UVa 00837 - Light and Transparencies, 335  
UVa 00839 - Not so Mobile, 222  
UVa 00843 - Crypt Kicker, 390  
UVa 00846 - Steps, 233  
UVa 00847 - A multiplication game, 273  
UVa 00850 - Crypt Kicker II, 283  
UVa 00852 - Deciding victory in Go, 162  
UVa 00855 - Lunch in Grid City, 47  
UVa 00856 - The Vigenère Cipher, 283  
UVa 00857 - Quantiser, 28  
UVa 00858 - Berry Picking, 348  
UVa 00859 - Chinese Checkers, 182  
UVa 00860 - Entropy Text Analyzer, 56  
UVa 00861 - Little Bishops, 98  
UVa 00865 - Substitution Cypher, 282  
UVa 00869 - Airline Comparison, 192  
UVa 00871 - Counting Cells in a Blob, 162  
UVa 00872 - Ordering \*, 162  
UVa 00880 - Cantor Fractions, 234  
UVa 00882 - The Mailbox ..., 376  
UVa 00884 - Factorial Factors, 263  
UVa 00886 - Named Extension Dialing, 290  
UVa 00890 - Maze (II), 284  
UVa 00892 - Finding words, 285  
UVa 00893 - Y3K \*, 29  
UVa 00895 - Word Problem, 283  
UVa 00897 - Annagrammatic Primes, 261  
UVa 00900 - Brick Wall Patterns, 250  
UVa 00902 - Password Search \*, 283  
UVa 00906 - Rational Neighbor, 232  
UVa 00907 - Winterim Backpack... \*, 222  
UVa 00908 - Re-connecting ..., 171  
UVa 00910 - TV Game, 222  
UVa 00911 - Multinomial Coefficients, 250

- UVa 00912 - Live From Mars, 285  
UVa 00913 - Joana and The Odd ..., 234  
UVa 00914 - Jumping Champion, 261  
UVa 00920 - Sunny Mountains \*, 335  
UVa 00922 - Rectangle by the Ocean, 390  
UVa 00924 - Spreading the News \*, 182  
UVa 00925 - No more prerequisites ..., 192  
UVa 00926 - Walking Around Wisely, 221  
UVa 00927 - Integer Sequence from ... \*, 94  
UVa 00928 - Eternal Truths, 368  
UVa 00929 - Number Maze \*, 183  
UVa 00939 - Genes, 56  
UVa 00941 - Permutations \*, 285  
UVa 00944 - Happy Numbers, 270  
UVa 00945 - Loading a Cargo Ship, 30  
UVa 00947 - Master Mind Helper, 27  
UVa 00948 - Fibonaccimal Base, 250  
UVa 00949 - Getaway, 182  
UVa 00957 - Popes, 103  
UVa 00960 - Gaussian Primes, 244  
UVa 00962 - Taxicab Numbers, 235  
UVa 00967 - Circular, 387  
UVa 00974 - Kaprekar Numbers, 235  
UVa 00976 - Bridge Building \*, 388  
UVa 00978 - Lemmings Battle \*, 57  
UVa 00983 - Localized Summing for ..., 136  
UVa 00985 - Round and Round ... \*, 368  
UVa 00986 - How Many?, 221  
UVa 00988 - Many paths, one ... \*, 221  
UVa 00988 - Many paths, one destination,  
    205  
UVa 00989 - Su Doku, 367  
UVa 00990 - Diving For Gold, 137  
UVa 00991 - Safe Salutations \*, 250  
UVa 00993 - Product of digits, 262  
UVa 01039 - Simplified GSM Network, 388  
UVa 01040 - The Traveling Judges \*, 391  
UVa 01047 - Zones \*, 96  
UVa 01052 - Bit Compression, 367  
UVa 01056 - Degrees of Separation \*, 190,  
    192  
UVa 01057 - Routing, 368  
UVa 01061 - Consanguine Calculations \*,  
    29  
UVa 01062 - Containers \*, 48  
UVa 01064 - Network, 96  
UVa 01079 - A Careful Approach, 391  
UVa 01092 - Tracking Bio-bots \*, 388  
UVa 01093 - Castles, 391  
UVa 01096 - The Islands \*, 398  
UVa 01098 - Robots on Ice \*, 368  
UVa 01099 - Sharing Chocolate \*, 377  
UVa 01103 - Ancient Messages \*, 162  
UVa 01111 - Trash Removal \*, 348  
UVa 01112 - Mice and Maze \*, 183  
UVa 01121 - Subsequence \*, 447  
UVa 01124 - Celebrity Jeopardy, 23  
UVa 01148 - The mysterious X network,  
    182  
UVa 01160 - X-Plosives, 171  
UVa 01172 - The Bridges of ... \*, 376  
UVa 01174 - IP-TV, 171  
UVa 01184 - Air Raid \*, 432  
UVa 01185 - BigNumber, 235  
UVa 01193 - Radar Install..., 110  
UVa 01194 - Machine Schedule, 223  
UVa 01195 - Calling Extraterrestrial ...,  
    389  
UVa 01196 - Tiling Up Blocks, 137  
UVa 01197 - The Suspects, 75  
UVa 01198 - Geodetic Set Problem, 192  
UVa 01200 - A DP Problem, 283  
UVa 01201 - Taxi Cab Scheme \*, 432  
UVa 01202 - Finding Nemo, 183  
UVa 01203 - Argus \*, 57  
UVa 01206 - Boundary Points, 348  
UVa 01207 - AGTC, 295  
UVa 01208 - Oreon, 171  
UVa 01209 - Wordfish, 47  
UVa 01210 - Sum of Consecutive ... \*, 244  
UVa 01211 - Atomic Car Race \*, 376  
UVa 01213 - Sum of Different Primes, 137  
UVa 01215 - String Cutting, 285  
UVa 01216 - The Bug Sensor Problem, 171  
UVa 01217 - Route Planning, 368  
UVa 01219 - Team Arrangement, 284  
UVa 01220 - Party at Hali-Bula \*, 377  
UVa 01221 - Against Mammoths, 387  
UVa 01222 - Bribing FIPA, 377  
UVa 01223 - Editor, 314  
UVa 01224 - Tile Code, 251  
UVa 01225 - Digit Counting \*, 232  
UVa 01226 - Numerical surprises, 243  
UVa 01229 - Sub-dictionary, 163, 190  
UVa 01230 - MODEX \*, 244  
UVa 01231 - ACORN \*, 376

- UVa 01232 - SKYLINE, 76  
UVa 01233 - USHER, 192  
UVa 01234 - RACING, 167, 171  
UVa 01235 - Anti Brute Force Lock, 171  
UVa 01237 - Expert Enough \*, 94  
UVa 01238 - Free Parentheses \*, 376  
UVa 01239 - Greatest K-Palindrome ..., 285  
UVa 01240 - ICPC Team Strategy, 377  
UVa 01241 - Jollybee Tournament, 48  
UVa 01242 - Necklace \*, 414  
UVa 01243 - Polynomial-time Red..., 388  
UVa 01244 - Palindromic paths, 377  
UVa 01246 - Find Terrorists, 263  
UVa 01247 - Interstar Transport, 192  
UVa 01249 - Euclid, 335  
UVa 01250 - Robot Challenge, 391  
UVa 01251 - Repeated Substitution ..., 368  
UVa 01252 - Twenty Questions \*, 377  
UVa 01253 - Infected Land, 368  
UVa 01254 - Top 10, 314  
UVa 01258 - Nowhere Money, 250  
UVa 01260 - Sales \*, 95  
UVa 01261 - String Popping, 138  
UVa 01262 - Password \*, 98  
UVa 01263 - Mines, 388  
UVa 01266 - Magic Square \*, 422  
UVa 01280 - Curvy Little Bottles, 387  
UVa 01347 - Tour \*, 398  
UVa 01388 - Graveyard, 336  
UVa 10000 - Longest Paths, 221  
UVa 10001 - Garden of Eden, 97  
UVa 10002 - Center of Mass?, 348  
UVa 10003 - Cutting Sticks, 133, 138  
UVa 10004 - Bicoloring \*, 151, 162  
UVa 10005 - Packing polygons \*, 336  
UVa 10006 - Carmichael Numbers, 235  
UVa 10007 - Count the Trees \*, 250  
UVa 10008 - What's Cryptanalysis?, 283  
UVa 10009 - All Roads Lead Where?, 182  
UVa 10010 - Where's Waldorf? \*, 290  
UVa 10012 - How Big Is It? \*, 389  
UVa 10013 - Super long sums, 243  
UVa 10014 - Simple calculations, 233  
UVa 10015 - Joseph's Cousin, 416  
UVa 10016 - Flip-flop the Squarelotron, 47  
UVa 10017 - The Never Ending ... \*, 453  
UVa 10018 - Reverse and Add, 28  
UVa 10019 - Funny Encryption Method, 282  
UVa 10020 - Minimal Coverage, 110  
UVa 10023 - Square root, 244  
UVa 10025 - The ? 1 ? 2 ? ..., 233  
UVa 10026 - Shoemaker's Problem, 110  
UVa 10029 - Edit Step Ladders, 377  
UVa 10032 - Tug of War, 377  
UVa 10033 - Interpreter, 30  
UVa 10034 - Freckles, 171  
UVa 10035 - Primary Arithmetic, 233  
UVa 10036 - Divisibility, 138  
UVa 10037 - Bridge, 110  
UVa 10038 - Jolly Jumpers \*, 46  
UVa 10041 - Vito's Family, 95  
UVa 10042 - Smith Numbers \*, 235  
UVa 10044 - Erdos numbers, 182  
UVa 10047 - The Monocycle, 368  
UVa 10048 - Audiophobia \*, 169, 171  
UVa 10049 - Self-describing Sequence, 235  
UVa 10050 - Hartals, 46  
UVa 10051 - Tower of Cubes, 221  
UVa 10054 - The Necklace \*, 222  
UVa 10055 - Hashmat the Brave Warrior, 232  
UVa 10056 - What is the Probability?, 266  
UVa 10057 - A mid-summer night ..., 47  
UVa 10058 - Jimmi's Riddles \*, 284  
UVa 10060 - A Hole to Catch a Man, 348  
UVa 10061 - How many zeros & how ..., 262  
UVa 10062 - Tell me the frequencies, 283  
UVa 10063 - Knuth's Permutation, 97  
UVa 10065 - Useless Tile Packers, 348  
UVa 10066 - The Twin Towers, 295  
UVa 10067 - Playing with Wheels, 182  
UVa 10069 - Distinct Subsequences, 376  
UVa 10070 - Leap Year or Not Leap ..., 29  
UVa 10071 - Back to High School ..., 232  
UVa 10074 - Take the Land, 137  
UVa 10075 - Airlines, 388  
UVa 10077 - The Stern-Brocot ..., 103  
UVa 10078 - Art Gallery \*, 396  
UVa 10079 - Pizza Cutting, 251  
UVa 10080 - Gopher II, 223  
UVa 10081 - Tight Words, 376  
UVa 10082 - WERTYU, 28  
UVa 10083 - Division, 243

- UVa 10086 - Test the Rods, 138  
UVa 10088 - Trees on My Island, 404  
UVa 10090 - Marbles \*, 264  
UVa 10092 - The Problem with the ..., 202  
UVa 10093 - An Easy Problem, 236  
UVa 10094 - Place the Guards, 98  
UVa 10097 - The Color game, 368  
UVa 10098 - Generating Fast, Sorted ..., 28  
UVa 10099 - Tourist Guide, 171  
UVa 10100 - Longest Match, 295  
UVa 10101 - Bangla Numbers, 235  
UVa 10102 - The Path in the ... \*, 95  
UVa 10104 - Euclid Problem \*, 264  
UVa 10105 - Polynomial Coefficients, 250  
UVa 10106 - Product, 243  
UVa 10107 - What is the Median? \*, 47  
UVa 10110 - Light, more light \*, 264  
UVa 10111 - Find the Winning ... \*, 273  
UVa 10112 - Myacm Triangles, 348  
UVa 10113 - Exchange Rates, 161  
UVa 10114 - Loansome Car Buyer \*, 23  
UVa 10115 - Automatic Editing, 285  
UVa 10116 - Robot Motion, 161  
UVa 10125 - Sumsets, 95  
UVa 10126 - Zipf's Law, 285  
UVa 10127 - Ones, 263  
UVa 10128 - Queue, 98  
UVa 10129 - Play on Words, 222  
UVa 10130 - SuperSale, 137  
UVa 10131 - Is Bigger Smarter?, 137  
UVa 10132 - File Fragmentation, 56  
UVa 10134 - AutoFish, 30  
UVa 10136 - Chocolate Chip Cookies, 336  
UVa 10137 - The Trip \*, 236  
UVa 10138 - CDVII, 56  
UVa 10139 - Factovisors \*, 263  
UVa 10140 - Prime Distance \*, 261  
UVa 10141 - Request for Proposal \*, 24  
UVa 10142 - Australian Voting, 30  
UVa 10147 - Highways, 167, 171  
UVa 10149 - Yahtzee, 377  
UVa 10150 - Doublets, 182  
UVa 10152 - ShellSort, 111  
UVa 10154 - Weights and Measures, 377  
UVa 10158 - War, 75  
UVa 10161 - Ant on a Chessboard \*, 234  
UVa 10162 - Last Digit, 270  
UVa 10163 - Storage Keepers, 377  
UVa 10164 - Number Game, 377  
UVa 10165 - Stone Game, 273  
UVa 10166 - Travel, 183  
UVa 10167 - Birthday Cake, 389  
UVa 10168 - Summation of Four Primes, 261  
UVa 10170 - The Hotel with Infinite ..., 233  
UVa 10171 - Meeting Prof. Miguel, 192  
UVa 10172 - The Lonesome Cargo ... \*, 48  
UVa 10174 - Couple-Bachelor-Spinster ..., 263  
UVa 10176 - Ocean Deep; Make it ... \*, 263  
UVa 10177 - (2/3/4)-D Sqr/Rects/..., 95  
UVa 10178 - Count the Faces, 404  
UVa 10179 - Irreducible Basic ... \*, 263  
UVa 10180 - Rope Crisis in Ropeland, 336  
UVa 10181 - 15-Puzzle Problem \*, 368  
UVa 10182 - Bee Maja \*, 234  
UVa 10183 - How many Fibs?, 250  
UVa 10187 - From Dusk Till Dawn, 183  
UVa 10188 - Automated Judge Script, 30  
UVa 10189 - Minesweeper \*, 27  
UVa 10190 - Divide, But Not Quite ..., 236  
UVa 10191 - Longest Nap, 28  
UVa 10192 - Vacation \*, 295  
UVa 10193 - All You Need Is Love, 244  
UVa 10194 - Football a.k.a. Soccer, 47  
UVa 10195 - The Knights Of The ..., 336  
UVa 10196 - Check The Check, 27  
UVa 10197 - Learning Portuguese, 285  
UVa 10198 - Counting, 243  
UVa 10199 - Tourist Guide, 163  
UVa 10200 - Prime Time, 387  
UVa 10201 - Adventures in Moving ..., 222  
UVa 10203 - Snow Clearing \*, 222  
UVa 10205 - Stack 'em Up, 27  
UVa 10209 - Is This Integration?, 336  
UVa 10210 - Romeo & Juliet, 336  
UVa 10212 - The Last Non-zero ... \*, 264  
UVa 10213 - How Many Pieces ... \*, 404  
UVa 10215 - The Largest/Smallest Box, 235  
UVa 10218 - Let's Dance, 266  
UVa 10219 - Find the Ways \*, 250  
UVa 10220 - I Love Big Numbers, 262  
UVa 10221 - Satellites, 336  
UVa 10222 - Decode the Mad Man, 282  
UVa 10223 - How Many Nodes?, 250

- UVa 10226 - Hardwood Species \*, 56  
UVa 10227 - Forests, 75  
UVa 10229 - Modular Fibonacci, 429  
UVa 10233 - Dermuba Triangle \*, 234  
UVa 10235 - Simply Emirp \*, 244  
UVa 10238 - Throw the Dice, 266  
UVa 10242 - Fourth Point, 335  
UVa 10243 - Fire; Fire; Fire \*, 396  
UVa 10245 - The Closest Pair Problem \*,  
    401  
UVa 10249 - The Grand Dinner, 110  
UVa 10250 - The Other Two Trees, 335  
UVa 10252 - Common Permutation \*, 283  
UVa 10257 - Dick and Jane, 233  
UVa 10258 - Contest Scoreboard \*, 47  
UVa 10259 - Hippity Hopscotch, 221  
UVa 10260 - Soundex, 46  
UVa 10261 - Ferry Loading, 137  
UVa 10263 - Railway \*, 335  
UVa 10264 - The Most Potent Corner \*, 48  
UVa 10267 - Graphical Editor, 30  
UVa 10268 - 498' \*, 235  
UVa 10271 - Chopsticks, 377  
UVa 10276 - Hanoi Tower Troubles Again,  
    96  
UVa 10278 - Fire Station, 183  
UVa 10279 - Mine Sweeper, 27  
UVa 10281 - Average Speed, 232  
UVa 10282 - Babelfish, 56  
UVa 10283 - The Kissing Circles, 336  
UVa 10284 - Chessboard in FEN \*, 27  
UVa 10285 - Longest Run ... \*, 221  
UVa 10286 - The Trouble with a ..., 336  
UVa 10293 - Word Length and Frequency,  
    283  
UVa 10295 - Hay Points, 56  
UVa 10296 - Jogging Trails \*, 400  
UVa 10297 - Beavergnaw \*, 337  
UVa 10298 - Power Strings \*, 290  
UVa 10299 - Relatives, 263  
UVa 10300 - Ecological Premium, 23  
UVa 10301 - Rings and Glue, 389  
UVa 10302 - Summation of Polynomials,  
    236  
UVa 10303 - How Many Trees, 250  
UVa 10304 - Optimal Binary ..., 377  
UVa 10305 - Ordering Tasks \*, 162  
UVa 10306 - e-Coins \*, 138  
UVa 10307 - Killing Aliens in Borg Maze,  
    388  
UVa 10308 - Roads in the North, 222  
UVa 10309 - Turn the Lights Off \*, 367  
UVa 10310 - Dog and Gopher, 389  
UVa 10311 - Goldbach and Euler, 262  
UVa 10312 - Expression Bracketing \*, 250  
UVa 10313 - Pay the Price, 138  
UVa 10315 - Poker Hands, 27  
UVa 10316 - Airline Hub, 412  
UVa 10318 - Security Panel, 367  
UVa 10319 - Manhattan \*, 395  
UVa 10323 - Factorial. You Must ..., 262  
UVa 10324 - Zeros and Ones, 24  
UVa 10325 - The Lottery, 389  
UVa 10326 - The Polynomial Equation,  
    236  
UVa 10327 - Flip Sort \*, 415  
UVa 10328 - Coin Toss, 266  
UVa 10330 - Power Transmission, 203  
UVa 10333 - The Tower of ASCII, 284  
UVa 10334 - Ray Through Glasses \*, 250  
UVa 10336 - Rank the Languages, 162  
UVa 10337 - Flight Planner \*, 138  
UVa 10338 - Mischievous Children \*, 262  
UVa 10339 - Watching Watches, 29  
UVa 10340 - All in All, 111  
UVa 10341 - Solve It \*, 103  
UVa 10344 - 23 Out of 5, 97  
UVa 10346 - Peter's Smoke \*, 233  
UVa 10347 - Medians, 336  
UVa 10349 - Antenna Placement \*, 223  
UVa 10350 - Liftless Eme \*, 221  
UVa 10354 - Avoiding Your Boss, 192  
UVa 10356 - Rough Roads, 183  
UVa 10357 - Playball, 335  
UVa 10359 - Tiling, 251  
UVa 10360 - Rat Attack, 91, 95  
UVa 10361 - Automatic Poetry, 285  
UVa 10363 - Tic Tac Toe, 28  
UVa 10364 - Square, 376  
UVa 10365 - Blocks, 95  
UVa 10368 - Euclid's Game, 273  
UVa 10369 - Arctic Networks \*, 171  
UVa 10370 - Above Average, 233  
UVa 10371 - Time Zones, 29  
UVa 10372 - Leaps Tall Buildings ..., 387  
UVa 10374 - Election, 283

- UVa 10375 - Choose and Divide, 250  
UVa 10377 - Maze Traversal, 161  
UVa 10382 - Watering Grass, 107, 110  
UVa 10387 - Billiard, 336  
UVa 10389 - Subway, 183  
UVa 10391 - Compound Words, 285  
UVa 10392 - Factoring Large Numbers, 262  
UVa 10393 - The One-Handed Typist \*, 285  
UVa 10394 - Twin Primes \*, 262  
UVa 10397 - Connect the Campus, 171  
UVa 10400 - Game Show Math, 138  
UVa 10401 - Injured Queen Problem \*, 221  
UVa 10404 - Bachet's Game, 273  
UVa 10405 - Longest Common ..., 295  
UVa 10406 - Cutting tabletops, 348  
UVa 10407 - Simple Division \*, 262  
UVa 10408 - Farey Sequences \*, 235  
UVa 10409 - Die Game, 27  
UVa 10415 - Eb Alto Saxophone Player, 29  
UVa 10419 - Sum-up the Primes, 376  
UVa 10420 - List of Conquests, 283  
UVa 10422 - Knights in FEN, 182  
UVa 10424 - Love Calculator, 24  
UVa 10427 - Naughty Sleepy ..., 389  
UVa 10430 - Dear GOD, 243  
UVa 10432 - Polygon Inside A Circle, 336  
UVa 10433 - Automorphic Numbers, 243  
UVa 10440 - Ferry Loading II, 111  
UVa 10443 - Rock, Scissors, Paper \*, 28  
UVa 10446 - The Marriage Interview, 138  
UVa 10449 - Traffic \*, 183  
UVa 10450 - World Cup Noise, 250  
UVa 10451 - Ancient ..., 336  
UVa 10452 - Marcus, help, 97  
UVa 10456 - Make Palindrome, 295  
UVa 10459 - The Tree Root \*, 222  
UVa 10460 - Find the Permuted String, 97  
UVa 10462 - Is There A Second ..., 171  
UVa 10464 - Big Big Real Numbers, 244  
UVa 10465 - Homer Simpson, 138  
UVa 10466 - How Far?, 335  
UVa 10469 - To Carry or not to Carry, 232  
UVa 10473 - Simple Base Conversion, 243  
UVa 10474 - Where is the Marble?, 103  
UVa 10475 - Help the Leaders, 97  
UVa 10480 - Sabotage, 199, 203  
UVa 10482 - The Candyman Can, 378  
UVa 10483 - The Sum Equals ..., 95  
UVa 10484 - Divisibility of Factors, 263  
UVa 10487 - Closest Sums \*, 95  
UVa 10489 - Boxes of Chocolates, 264  
UVa 10490 - Mr. Azad and his Son, 262  
UVa 10491 - Cows and Cars \*, 266  
UVa 10493 - Cats, with or without Hats, 234  
UVa 10494 - If We Were a Child Again, 243  
UVa 10496 - Collecting Beepers \*, 138  
UVa 10497 - Sweet Child Make Trouble, 250  
UVa 10499 - The Land of Justice, 233  
UVa 10500 - Robot maps, 284  
UVa 10502 - Counting Rectangles, 95  
UVa 10503 - The dominoes solitaire \*, 97  
UVa 10505 - Montesco vs Capuleto, 162  
UVa 10506 - Ouroboros, 97  
UVa 10507 - Waking up brain \*, 76  
UVa 10508 - Word Morphing, 285  
UVa 10509 - R U Kidding Mr. ..., 234  
UVa 10511 - Councilling, 202  
UVa 10515 - Power et al, 270  
UVa 10518 - How Many Calls? \*, 429  
UVa 10519 - Really Strange, 243  
UVa 10520 - Determine it, 138  
UVa 10522 - Height to Area, 336  
UVa 10523 - Very Easy \*, 243  
UVa 10525 - New to Bangladesh?, 192  
UVa 10527 - Persistent Numbers, 263  
UVa 10528 - Major Scales, 28  
UVa 10530 - Guessing Game, 27  
UVa 10532 - Combination, Once Again, 250  
UVa 10533 - Digit Primes, 387  
UVa 10534 - Wavio Sequence, 137  
UVa 10536 - Game of Euler, 376  
UVa 10539 - Almost Prime Numbers \*, 389  
UVa 10541 - Stripe \*, 250  
UVa 10543 - Traveling Politician, 222  
UVa 10550 - Combination Lock, 23  
UVa 10551 - Basic Remains \*, 243  
UVa 10554 - Calories from Fat, 28  
UVa 10557 - XYZZY \*, 183  
UVa 10566 - Crossed Ladders, 387  
UVa 10567 - Helping Fill Bates \*, 103  
UVa 10573 - Geometry Paradox, 336

- UVa 10576 - Y2K Accounting Bug \*, 97  
UVa 10577 - Bounding box \*, 336  
UVa 10578 - The Game of 31, 273  
UVa 10579 - Fibonacci Numbers, 250  
UVa 10582 - ASCII Labyrinth, 98  
UVa 10583 - Ubiquitous Religions, 76  
UVa 10585 - Center of symmetry, 335  
UVa 10586 - Polynomial Remains \*, 236  
UVa 10589 - Area \*, 336  
UVa 10591 - Happy Number, 270  
UVa 10594 - Data Flow, 431  
UVa 10596 - Morning Walk \*, 222  
UVa 10600 - ACM Contest and ... \*, 171  
UVa 10602 - Editor Nottobad, 111  
UVa 10603 - Fill, 183  
UVa 10604 - Chemical Reaction, 377  
UVa 10606 - Opening Doors, 387  
UVa 10608 - Friends, 76  
UVa 10610 - Gopher and Hawks, 182  
UVa 10611 - Playboy Chimp, 103  
UVa 10616 - Divisible Group Sum \*, 137  
UVa 10617 - Again Palindrome, 295  
UVa 10620 - A Flea on a Chessboard, 234  
UVa 10622 - Perfect P-th Power, 263  
UVa 10624 - Super Number, 233  
UVa 10625 - GNU = GNU'sNotUnix, 283  
UVa 10626 - Buying Coke, 378  
UVa 10633 - Rare Easy Problem, 264  
UVa 10635 - Prince and Princess \*, 295  
UVa 10637 - Coprimes \*, 389  
UVa 10642 - Can You Solve It?, 234  
UVa 10643 - Facing Problems With ..., 250  
UVa 10645 - Menu, 377  
UVa 10646 - What is the Card? \*, 27  
UVa 10650 - Determinate Prime, 262  
UVa 10651 - Pebble Solitaire, 376  
UVa 10652 - Board Wrapping \*, 349  
UVa 10653 - Bombs; NO they ... \*, 182  
UVa 10655 - Contemplation  
    Algebra \*, 429  
UVa 10656 - Maximum Sum (II) \*, 111  
UVa 10659 - Fitting Text into Slides, 29  
UVa 10660 - Citizen attention ... \*, 95  
UVa 10662 - The Wedding, 95  
UVa 10664 - Luggage, 137  
UVa 10666 - The Eurocup is here, 234  
UVa 10667 - Largest Block, 137  
UVa 10668 - Expanding Rods, 387  
UVa 10669 - Three powers, 243  
UVa 10670 - Work Reduction, 110  
UVa 10672 - Marbles on a tree, 111  
UVa 10673 - Play with Floor and Ceil \*,  
    264  
UVa 10677 - Base Equality, 236  
UVa 10678 - The Grazing Cows \*, 336  
UVa 10679 - I Love Strings, 285  
UVa 10680 - LCM \*, 263  
UVa 10681 - Teobaldo's Trip, 222  
UVa 10683 - The decadary watch, 29  
UVa 10684 - The Jackpot \*, 136  
UVa 10685 - Nature, 76  
UVa 10686 - SQF Problem, 56  
UVa 10687 - Monitoring the Amazon, 161  
UVa 10688 - The Poor Giant, 138  
UVa 10689 - Yet Another Number ... \*, 250  
UVa 10690 - Expression Again, 376  
UVa 10693 - Traffic Volume, 234  
UVa 10696 - f91, 233  
UVa 10698 - Football Sort, 47  
UVa 10699 - Count the ... \*, 263  
UVa 10700 - Camel Trading, 111  
UVa 10701 - Pre, in and post, 222  
UVa 10702 - Traveling Salesman, 222  
UVa 10703 - Free spots, 47  
UVa 10706 - Number Sequence, 103  
UVa 10707 - 2D - Nim, 162  
UVa 10710 - Chinese Shuffle, 234  
UVa 10714 - Ants, 111  
UVa 10717 - Mint \*, 389  
UVa 10718 - Bit Mask \*, 111  
UVa 10719 - Quotient Polynomial, 236  
UVa 10720 - Graph Construction \*, 404  
UVa 10721 - Bar Codes \*, 138  
UVa 10722 - Super Lucky Numbers, 378  
UVa 10724 - Road Construction, 192  
UVa 10730 - Antiarithmetic?, 95  
UVa 10731 - Test, 163, 190  
UVa 10733 - The Colored Cubes, 251  
UVa 10734 - Triangle Partitioning, 390  
UVa 10738 - Riemann vs. Mertens \*, 263  
UVa 10739 - String to Palindrome, 295  
UVa 10742 - New Rule in Euphonmia, 103  
UVa 10746 - Crime Wave - The Sequel \*,  
    431  
UVa 10751 - Chessboard \*, 233  
UVa 10755 - Garbage Heap \*, 136

- UVa 10759 - Dice Throwing \*, 266  
UVa 10761 - Broken Keyboard, 284  
UVa 10763 - Foreign Exchange, 110  
UVa 10765 - Doves and Bombs \*, 163  
UVa 10771 - Barbarian tribes \*, 416  
UVa 10773 - Back to Intermediate ... \*, 232  
UVa 10774 - Repeated Josephus \*, 416  
UVa 10777 - God, Save me, 266  
UVa 10779 - Collectors Problem, 203  
UVa 10780 - Again Prime? No time., 263  
UVa 10783 - Odd Sum, 233  
UVa 10784 - Diagonal \*, 251  
UVa 10785 - The Mad Numerologist, 110  
UVa 10789 - Prime Frequency, 283  
UVa 10790 - How Many Points of ..., 251  
UVa 10791 - Minimum Sum LCM, 263  
UVa 10792 - The Laurel-Hardy Story, 336  
UVa 10793 - The Orc Attack, 192  
UVa 10800 - Not That Kind of Graph \*,  
    284  
UVa 10801 - Lift Hopping \*, 183  
UVa 10803 - Thunder Mountain, 192  
UVa 10804 - Gopher Strategy, 387  
UVa 10805 - Cockroach Escape ... \*, 222  
UVa 10806 - Dijkstra, Dijkstra, 431  
UVa 10810 - Ultra Quicksort, 415  
UVa 10812 - Beat the Spread \*, 28  
UVa 10813 - Traditional BINGO \*, 28  
UVa 10814 - Simplifying Fractions \*, 244  
UVa 10815 - Andy's First Dictionary, 57  
UVa 10816 - Travel in Desert, 387  
UVa 10817 - Headmaster's Headache, 377  
UVa 10819 - Trouble of 13-Dots \*, 137  
UVa 10820 - Send A Table, 263  
UVa 10823 - Of Circles and Squares, 389  
UVa 10827 - Maximum Sum on ... \*, 137  
UVa 10832 - Yoyodyne Propulsion ..., 335  
UVa 10842 - Traffic Flow, 171  
UVa 10843 - Anne's game, 404  
UVa 10849 - Move the bishop, 27  
UVa 10851 - 2D Hieroglyphs ... \*, 282  
UVa 10852 - Less Prime, 262  
UVa 10854 - Number of Paths \*, 284  
UVa 10855 - Rotated squares \*, 47  
UVa 10856 - Recover Factorial, 391  
UVa 10858 - Unique Factorization, 48  
UVa 10862 - Connect the Cable Wires, 250  
UVa 10865 - Brownie Points, 335  
UVa 10870 - Recurrences, 429  
UVa 10871 - Primed Subsequence, 388  
UVa 10874 - Segments, 222  
UVa 10875 - Big Math, 284  
UVa 10876 - Factory Robot, 391  
UVa 10878 - Decode the Tape \*, 282  
UVa 10879 - Code Refactoring, 233  
UVa 10880 - Colin and Ryan, 47  
UVa 10882 - Koerner's Pub, 234  
UVa 10888 - Warehouse \*, 431  
UVa 10890 - Maze, 367  
UVa 10891 - Game of Sum \*, 388  
UVa 10892 - LCM Cardinality \*, 262  
UVa 10894 - Save Hridoy, 284  
UVa 10895 - Matrix Transpose \*, 75  
UVa 10896 - Known Plaintext Attack, 282  
UVa 10897 - Travelling Distance, 412  
UVa 10898 - Combo Deal, 376  
UVa 10901 - Ferry Loading III \*, 48  
UVa 10902 - Pick-up sticks, 335  
UVa 10903 - Rock-Paper-Scissors ..., 28  
UVa 10905 - Children's Game, 47  
UVa 10906 - Strange Integration \*, 283  
UVa 10908 - Largest Square, 95  
UVa 10910 - Mark's Distribution, 138  
UVa 10911 - Forming Quiz Teams, 1  
UVa 10911 - Forming Quiz Teams \*, 376  
UVa 10912 - Simple Minded ..., 138  
UVa 10913 - Walking ... \*, 222  
UVa 10916 - Factstone Benchmark \*, 235  
UVa 10917 - A Walk Through the Forest,  
    388  
UVa 10918 - Tri Tiling, 251  
UVa 10919 - Prerequisites?, 24  
UVa 10920 - Spiral Tap \*, 47  
UVa 10921 - Find the Telephone, 282  
UVa 10922 - 2 the 9s, 264  
UVa 10923 - Seven Seas, 368  
UVa 10924 - Prime Words, 244  
UVa 10925 - Krakovia, 243  
UVa 10926 - How Many Dependencies?,  
    221  
UVa 10927 - Bright Lights \*, 335  
UVa 10928 - My Dear Neighbours, 75  
UVa 10929 - You can say 11, 264  
UVa 10930 - A-Sequence, 235  
UVa 10931 - Parity \*, 236  
UVa 10935 - Throwing cards away I, 48

- UVa 10937 - Blackbeard the Pirate, 388  
UVa 10938 - Flea circus, 421  
UVa 10940 - Throwing Cards Away II \*, 233  
UVa 10943 - How do you add?, 132, 211  
UVa 10943 - How do you add? \*, 138  
UVa 10944 - Nuts for nuts.., 388  
UVa 10945 - Mother Bear \*, 28  
UVa 10946 - You want what filled?, 162  
UVa 10947 - Bear with me, again.., 192  
UVa 10948 - The Primary Problem, 262  
UVa 10950 - Bad Code, 97  
UVa 10954 - Add All \*, 57  
UVa 10957 - So Doku Checker, 367  
UVa 10958 - How Many Solutions?, 263  
UVa 10959 - The Party, Part I, 182  
UVa 10961 - Chasing After Don Giovanni, 30  
UVa 10963 - The Swallowing Ground, 23  
UVa 10964 - Strange Planet, 234  
UVa 10967 - The Great Escape, 183  
UVa 10970 - Big Chocolate, 234  
UVa 10973 - Triangle Counting, 95  
UVa 10976 - Fractions Again ? \*, 94  
UVa 10977 - Enchanted Forest, 182  
UVa 10978 - Let's Play Magic, 46  
UVa 10980 - Lowest Price in Town, 138  
UVa 10982 - Troublemakers, 111  
UVa 10983 - Buy one, get ... \*, 387  
UVa 10986 - Sending email \*, 183  
UVa 10990 - Another New Function \*, 263  
UVa 10991 - Region, 336  
UVa 10992 - The Ghost of Programmers, 243  
UVa 10994 - Simple Addition, 234  
UVa 11000 - Bee, 250  
UVa 11001 - Necklace, 94  
UVa 11002 - Towards Zero, 377  
UVa 11003 - Boxes, 137  
UVa 11005 - Cheapest Base, 236  
UVa 11015 - 05-32 Rendezvous, 192  
UVa 11021 - Tribbles, 266  
UVa 11022 - String Factoring \*, 295  
UVa 11026 - A Grouping Problem, 139  
UVa 11028 - Sum of Product, 235  
UVa 11029 - Leading and Trailing, 264  
UVa 11034 - Ferry Loading IV \*, 48  
UVa 11036 - Eventually periodic ..., 270  
UVa 11039 - Building Designing, 47  
UVa 11040 - Add bricks in the wall, 47  
UVa 11042 - Complex, difficult and ..., 264  
UVa 11044 - Searching for Nessy, 23  
UVa 11045 - My T-Shirt Suits Me, 203  
UVa 11047 - The Scrooge Co Problem, 192  
UVa 11048 - Automatic Correction ... \*, 285  
UVa 11049 - Basic Wall Maze, 182  
UVa 11053 - Flavius Josephus ... \*, 270  
UVa 11054 - Wine Trading in Gergovia, 111  
UVa 11055 - Homogeneous Square, 236  
UVa 11056 - Formula 1 \*, 285  
UVa 11057 - Exact Sum \*, 103  
UVa 11059 - Maximum Product, 95  
UVa 11060 - Beverages \*, 150, 162  
UVa 11062 - Andy's Second Dictionary, 57  
UVa 11063 - B2 Sequences, 235  
UVa 11064 - Number Theory, 263  
UVa 11065 - A Gentlemen's Agreement \*, 367  
UVa 11067 - Little Red Riding Hood, 221  
UVa 11068 - An Easy Task, 335  
UVa 11069 - A Graph Problem \*, 251  
UVa 11070 - The Good Old Times, 284  
UVa 11074 - Draw Grid, 284  
UVa 11078 - Open Credit System, 94  
UVa 11080 - Place the Guards \*, 162  
UVa 11084 - Anagram Division, 377  
UVa 11085 - Back to the 8-Queens \*, 97  
UVa 11086 - Composite Prime, 263  
UVa 11088 - End up with More Teams, 376  
UVa 11089 - Fi-binary Number, 250  
UVa 11090 - Going in Cycle, 98  
UVa 11093 - Just Finish it up, 46  
UVa 11094 - Continents \*, 162  
UVa 11096 - Nails, 349  
UVa 11100 - The Trip, 2007 \*, 110  
UVa 11101 - Mall Mania \*, 182  
UVa 11103 - WFF'N Proof, 110  
UVa 11105 - Semi-prime H-numbers \*, 388  
UVa 11107 - Life Forms \*, 314  
UVa 11108 - Tautology, 96  
UVa 11110 - Equidivisions, 162  
UVa 11111 - Generalized Matrioshkas \*, 399  
UVa 11115 - Uncle Jack, 251

- UVa 11121 - Base -2, 236  
UVa 11125 - Arrange Some Marbles, 378  
UVa 11127 - Triple-Free Binary Strings,  
367  
UVa 11130 - Billiard bounces \*, 233  
UVa 11131 - Close Relatives, 222  
UVa 11133 - Eigensequence, 378  
UVa 11136 - Hoax or what \*, 57  
UVa 11137 - Ingenuous Cubrency, 138  
UVa 11138 - Nuts and Bolts \*, 223  
UVa 11140 - Little Ali's Little Brother, 30  
UVa 11148 - Moliu Fractions, 283  
UVa 11150 - Cola, 233  
UVa 11151 - Longest Palindrome \*, 295  
UVa 11152 - Colourful ... \*, 336  
UVa 11157 - Dynamic Frog \*, 111  
UVa 11159 - Factors and Multiples \*, 223  
UVa 11161 - Help My Brother (II), 250  
UVa 11163 - Jaguar King, 368  
UVa 11164 - Kingdom Division, 336  
UVa 11167 - Monkeys in the Emei ... \*, 203  
UVa 11172 - Relational Operators \*, 23  
UVa 11173 - Grey Codes, 45, 48  
UVa 11176 - Winning Streak \*, 266  
UVa 11181 - Probability (bar) Given, 266  
UVa 11185 - Ternary, 243  
UVa 11192 - Group Reverse, 46  
UVa 11195 - Another n-Queen Problem, 88  
UVa 11195 - Another n-Queen Problem \*,  
367  
UVa 11198 - Dancing Digits \*, 368  
UVa 11201 - The Problem with the ..., 97  
UVa 11202 - The least possible effort, 233  
UVa 11203 - Can you decide it ... \*, 283  
UVa 11204 - Musical Instruments, 251  
UVa 11205 - The Broken Pedometer, 96  
UVa 11207 - The Easiest Way, 337  
UVa 11212 - Editing a Book \*, 368  
UVa 11218 - KTV, 376  
UVa 11219 - How old are you?, 29  
UVa 11220 - Decoding the message, 282  
UVa 11221 - Magic Square Palindrome \*,  
28  
UVa 11222 - Only I did it, 46  
UVa 11223 - O: dah, dah, dah, 29  
UVa 11225 - Tarot scores, 27  
UVa 11226 - Reaching the fix-point, 263  
UVa 11227 - The silver bullet \*, 389  
UVa 11228 - Transportation ... \*, 171  
UVa 11230 - Annoying painting tool, 111  
UVa 11231 - Black and White Painting \*,  
234  
UVa 11233 - Deli Deli, 285  
UVa 11234 - Expressions, 222  
UVa 11235 - Frequent Values \*, 76  
UVa 11236 - Grocery Store \*, 96  
UVa 11239 - Open Source, 56  
UVa 11240 - Antimonotonicity, 111  
UVa 11241 - Humidex, 236  
UVa 11242 - Tour de France \*, 95  
UVa 11244 - Counting Stars, 162  
UVa 11246 - K-Multiple Free Set, 234  
UVa 11247 - Income Tax Hazard, 233  
UVa 11254 - Consecutive Integers \*, 233  
UVa 11258 - String Partition \*, 295  
UVa 11262 - Weird Fence \*, 387  
UVa 11264 - Coin Collector \*, 110  
UVa 11265 - The Sultan's Problem \*, 349  
UVa 11267 - The 'Hire-a-Coder' ..., 388  
UVa 11269 - Setting Problems, 110  
UVa 11270 - Tiling Dominoes, 251  
UVa 11278 - One-Handed Typist \*, 282  
UVa 11280 - Flying to Fredericton, 183  
UVa 11281 - Triangular Pegs in ..., 336  
UVa 11282 - Mixing Invitations, 389  
UVa 11283 - Playing Boggle \*, 290  
UVa 11284 - Shopping Trip \*, 138  
UVa 11285 - Exchange Rates, 377  
UVa 11286 - Conformity \*, 56  
UVa 11287 - Pseudoprime Numbers \*, 244  
UVa 11291 - Smeech \*, 284  
UVa 11292 - Dragon of Loowater, 108  
UVa 11292 - Dragon of Loowater \*, 110  
UVa 11296 - Counting Solutions to an ...,  
234  
UVa 11297 - Census, 76  
UVa 11298 - Dissecting a Hexagon, 234  
UVa 11301 - Great Wall of China \*, 431  
UVa 11307 - Alternative Arborescence, 222  
UVa 11308 - Bankrupt Baker, 56  
UVa 11309 - Counting Chaos, 28  
UVa 11310 - Delivery Debacle \*, 251  
UVa 11311 - Exclusively Edible \*, 273  
UVa 11313 - Gourmet Games, 233  
UVa 11319 - Stupid Sequence? \*, 406  
UVa 11321 - Sort Sort and Sort, 48

- UVa 11324 - The Largest Clique \*, 388  
UVa 11326 - Laser Pointer, 336  
UVa 11327 - Enumerating Rational ..., 263  
UVa 11329 - Curious Fleas \*, 368  
UVa 11332 - Summing Digits, 23  
UVa 11335 - Discrete Pursuit, 111  
UVa 11338 - Minefield, 183  
UVa 11340 - Newspaper \*, 46  
UVa 11341 - Term Strategy, 137  
UVa 11342 - Three-square, 96  
UVa 11343 - Isolated Segments, 335  
UVa 11344 - The Huge One \*, 264  
UVa 11345 - Rectangles, 337  
UVa 11346 - Probability, 266  
UVa 11347 - Multifactorials, 263  
UVa 11348 - Exhibition, 56  
UVa 11349 - Symmetric Matrix, 47  
UVa 11350 - Stern-Brocot Tree, 76  
UVa 11351 - Last Man Standing \*, 416  
UVa 11352 - Crazy King, 182  
UVa 11353 - A Different kind of Sorting, 263  
UVa 11356 - Dates, 29  
UVa 11357 - Ensuring Truth \*, 283  
UVa 11360 - Have Fun with Matrices, 47  
UVa 11362 - Phone List, 290  
UVa 11364 - Parking, 23  
UVa 11367 - Full Tank?, 178, 183  
UVa 11368 - Nested Dolls, 137  
UVa 11369 - Shopaholic, 110  
UVa 11371 - Number Theory for ... \*, 264  
UVa 11377 - Airport Setup, 183  
UVa 11378 - Bey Battle \*, 401  
UVa 11380 - Down Went The ..., 200  
UVa 11380 - Down Went The Titanic \*, 203  
UVa 11384 - Help is needed for Dexter, 235  
UVa 11385 - Da Vinci Code \*, 283  
UVa 11387 - The 3-Regular Graph, 234  
UVa 11388 - GCD LCM, 262  
UVa 11389 - The Bus Driver Problem \*, 110  
UVa 11391 - Blobs in the Board \*, 377  
UVa 11393 - Tri-Isomorphism, 234  
UVa 11395 - Sigma Function, 263  
UVa 11396 - Claw Decomposition \*, 162  
UVa 11398 - The Base-1 Number System, 236  
UVa 11401 - Triangle Counting \*, 251  
UVa 11402 - Ahoy Pirates, 70  
UVa 11402 - Ahoy, Pirates \*, 76  
UVa 11405 - Can U Win? \*, 388  
UVa 11407 - Squares, 139  
UVa 11408 - Count DePrimes \*, 388  
UVa 11412 - Dig the Holes, 96  
UVa 11413 - Fill the ... \*, 103  
UVa 11414 - Dreams, 404  
UVa 11415 - Count the Factorials, 389  
UVa 11417 - GCD, 262  
UVa 11418 - Clever Naming Patterns, 203  
UVa 11419 - SAM I AM, 223  
UVa 11420 - Chest of ..., 139  
UVa 11428 - Cubes, 389  
UVa 11432 - Busy Programmer, 378  
UVa 11437 - Triangle Fun, 336  
UVa 11439 - Maximizing the ICPC \*, 410  
UVa 11447 - Reservoir Logs, 349  
UVa 11448 - Who said crisis?, 243  
UVa 11450 - Wedding Shopping, 139  
UVa 11452 - Dancing the Cheeky ... \*, 285  
UVa 11455 - Behold My Quadrangle, 337  
UVa 11456 - Trainsorting \*, 137  
UVa 11459 - Snakes and Ladders \*, 27  
UVa 11461 - Square Numbers, 235  
UVa 11462 - Age Sort \*, 450  
UVa 11463 - Commandos, 184  
UVa 11463 - Commandos \*, 192  
UVa 11464 - Even Parity, 367  
UVa 11466 - Largest Prime Divisor \*, 262  
UVa 11470 - Square Sums, 162  
UVa 11471 - Arrange the Tiles, 367  
UVa 11472 - Beautiful Numbers, 378  
UVa 11473 - Campus Roads, 349  
UVa 11474 - Dying Tree \*, 390  
UVa 11475 - Extend to Palindromes \*, 290  
UVa 11476 - Factoring Large(t) ... \*, 437  
UVa 11479 - Is this the easiest problem?, 336  
UVa 11480 - Jimmy's Balls, 251  
UVa 11482 - Building a Triangular ..., 284  
UVa 11483 - Code Creator, 285  
UVa 11486 - Finding Paths in Grid \*, 429  
UVa 11487 - Gathering Food \*, 222  
UVa 11489 - Integer Game \*, 273  
UVa 11491 - Erasing and Winning, 388  
UVa 11492 - Babel \*, 183

- UVa 11494 - Queen, 27  
UVa 11495 - Bubbles and Buckets, 415  
UVa 11496 - Musical Loop, 46  
UVa 11498 - Division of Nlogonia \*, 23  
UVa 11500 - Vampires, 266  
UVa 11503 - Virtual Friends \*, 76  
UVa 11504 - Dominos \*, 163  
UVa 11505 - Logo, 335  
UVa 11506 - Angry Programmer \*, 203  
UVa 11507 - Bender B. Rodriguez ... \*, 24  
UVa 11512 - GATTACA \*, 314  
UVa 11513 - 9 Puzzle, 368  
UVa 11515 - Cranes, 389  
UVa 11516 - WiFi \*, 387  
UVa 11517 - Exact Change \*, 138  
UVa 11518 - Dominos 2, 162  
UVa 11519 - Logo 2, 335  
UVa 11520 - Fill the Square, 111  
UVa 11525 - Permutation \*, 390  
UVa 11526 - H(n) \*, 236  
UVa 11530 - SMS Typing, 28  
UVa 11532 - Simple Adjacency ..., 111  
UVa 11536 - Smallest Sub-Array \*, 447  
UVa 11538 - Chess Queen \*, 251  
UVa 11541 - Decoding, 282  
UVa 11545 - Avoiding ..., 222  
UVa 11547 - Automatic Answer, 23  
UVa 11548 - Blackboard Bonanza, 96  
UVa 11549 - Calculator Conundrum, 270  
UVa 11550 - Demanding Dilemma, 75  
UVa 11552 - Fewest Flops, 295  
UVa 11553 - Grid Game \*, 96  
UVa 11554 - Hapless Hedonism, 251  
UVa 11556 - Best Compression Ever, 235  
UVa 11559 - Event Planning \*, 23  
UVa 11561 - Getting Gold, 162  
UVa 11565 - Simple Equations, 84  
UVa 11565 - Simple Equations \*, 96  
UVa 11566 - Let's Yum Cha \*, 137  
UVa 11567 - Moliu Number Generator, 111  
UVa 11572 - Unique Snowflakes \*, 56  
UVa 11574 - Colliding Traffic \*, 390  
UVa 11576 - Scrolling Sign \*, 290  
UVa 11577 - Letter Frequency, 283  
UVa 11579 - Triangle Trouble, 336  
UVa 11581 - Grid Successors \*, 47  
UVa 11586 - Train Tracks, 24  
UVa 11588 - Image Coding, 48  
UVa 11597 - Spanning Subtree \*, 251  
UVa 11608 - No Problem, 46  
UVa 11609 - Teams, 251  
UVa 11610 - Reverse Prime \*, 391  
UVa 11614 - Etruscan Warriors Never ..., 232  
UVa 11615 - Family Tree, 222  
UVa 11616 - Roman Numerals \*, 441  
UVa 11621 - Small Factors, 103  
UVa 11624 - Fire, 182  
UVa 11626 - Convex Hull, 349  
UVa 11628 - Another lottery, 266  
UVa 11629 - Ballot evaluation, 56  
UVa 11631 - Dark Roads \*, 171  
UVa 11634 - Generate random ... \*, 270  
UVa 11635 - Hotel Booking \*, 388  
UVa 11636 - Hello World, 235  
UVa 11639 - Guard the Land, 337  
UVa 11643 - Knight Tour \*, 417  
UVa 11646 - Athletics Track, 387  
UVa 11650 - Mirror Clock, 29  
UVa 11655 - Waterland, 221  
UVa 11658 - Best Coalition, 137  
UVa 11660 - Look-and-Say sequences, 235  
UVa 11661 - Burger Time?, 24  
UVa 11664 - Langton's Ant, 243  
UVa 11666 - Logarithms, 235  
UVa 11677 - Alarm Clock, 29  
UVa 11678 - Card's Exchange, 27  
UVa 11679 - Sub-prime, 23  
UVa 11683 - Laser Sculpture, 24  
UVa 11686 - Pick up sticks, 162  
UVa 11687 - Digits, 24  
UVa 11689 - Soda Surpler, 233  
UVa 11690 - Money Matters, 76  
UVa 11692 - Rain Fall, 236  
UVa 11693 - Speedy Escape, 388  
UVa 11695 - Flight Planning \*, 222  
UVa 11697 - Playfair Cipher \*, 283  
UVa 11701 - Cantor, 103  
UVa 11703 - sqrt log sin, 139  
UVa 11709 - Trust Groups, 163  
UVa 11710 - Expensive Subway, 171  
UVa 11713 - Abstract Names, 285  
UVa 11714 - Blind Sorting, 235  
UVa 11715 - Car, 236  
UVa 11716 - Digital Fortress, 282  
UVa 11717 - Energy Saving Micro..., 30

- UVa 11718 - Fantasy of a Summation \*, 234  
UVa 11719 - Gridlands Airports \*, 404  
UVa 11721 - Instant View ..., 388  
UVa 11723 - Numbering Road \*, 232  
UVa 11727 - Cost Cutting \*, 23  
UVa 11728 - Alternate Task \*, 263  
UVa 11729 - Commando War, 110  
UVa 11730 - Number Transformation, 389  
UVa 11733 - Airports, 171  
UVa 11734 - Big Number of ..., 285  
UVa 11742 - Social Constraints, 85, 96  
UVa 11743 - Credit Check, 29  
UVa 11747 - Heavy Cycle Edges \*, 171  
UVa 11749 - Poor Trade Advisor, 162  
UVa 11752 - The Super ..., 262  
UVa 11760 - Brother Arif, ..., 48  
UVa 11764 - Jumping Mario, 23  
UVa 11770 - Lighting Away, 163  
UVa 11774 - Doom's Day, 262  
UVa 11777 - Automate the Grades, 48  
UVa 11780 - Miles 2 Km, 250  
UVa 11782 - Optimal Cut, 222  
UVa 11787 - Numeral Hieroglyphs, 282  
UVa 11790 - Murcia's Skyline \*, 137  
UVa 11792 - Krochanska is Here, 182  
UVa 11799 - Horror Dash \*, 23  
UVa 11800 - Determine the Shape \*, 337  
UVa 11804 - Argentina, 96  
UVa 11805 - Bafana Bafana, 232  
UVa 11813 - Shopping, 388  
UVa 11816 - HST, 236  
UVa 11817 - Tunnelling The Earth, 412  
UVa 11821 - High-Precision Number \*, 244  
UVa 11824 - A Minimum Land Price, 48  
UVa 11827 - Maximum GCD \*, 262  
UVa 11830 - Contract revision, 243  
UVa 11831 - Sticker Collector ... \*, 161  
UVa 11832 - Account Book, 376  
UVa 11833 - Route Change, 183  
UVa 11834 - Elevator, 337  
UVa 11835 - Formula 1, 47  
UVa 11838 - Come and Go \*, 158, 163  
UVa 11839 - Optical Reader, 285  
UVa 11847 - Cut the Silver Bar \*, 235  
UVa 11849 - CD \*, 57  
UVa 11850 - Alaska, 47  
UVa 11854 - Egypt, 336  
UVa 11857 - Driving Range, 171  
UVa 11858 - Frosh Week \*, 415  
UVa 11860 - Document Analyzer, 56  
UVa 11875 - Brick Game \*, 232  
UVa 11876 - N + NOD (N), 103  
UVa 11877 - The Coco-Cola Store, 233  
UVa 11878 - Homework Checker \*, 283  
UVa 11879 - Multiple of 17 \*, 243  
UVa 11881 - Internal Rate of Return, 104  
UVa 11888 - Abnormal 89's, 290  
UVa 11889 - Benefit \*, 263  
UVa 11894 - Genius MJ, 335  
UVa 11900 - Boiled Eggs, 111  
UVa 11902 - Dominator, 145, 161  
UVa 11906 - Knight in a War Grid \*, 161  
UVa 11909 - Soya Milk \*, 336  
UVa 11917 - Do Your Own Homework, 57  
UVa 11926 - Multitasking \*, 48  
UVa 11933 - Splitting Numbers \*, 48  
UVa 11934 - Magic Formula, 233  
UVa 11935 - Through the Desert, 101, 104  
UVa 11936 - The Lazy Lumberjacks, 336  
UVa 11942 - Lumberjack Sequencing, 23  
UVa 11945 - Financial Management, 28  
UVa 11946 - Code Number, 282  
UVa 11947 - Cancer or Scorpio \*, 30  
UVa 11951 - Area \*, 137  
UVa 11952 - Arithmetic, 243  
UVa 11953 - Battleships \*, 162  
UVa 11955 - Binomial Theorem \*, 250  
UVa 11956 - Brain\*\*\*\*, 24  
UVa 11957 - Checkers \*, 221  
UVa 11958 - Coming Home, 30  
UVa 11959 - Dice, 96  
UVa 11960 - Divisor Game \*, 390  
UVa 11961 - DNA, 97  
UVa 11962 - DNA II, 285  
UVa 11965 - Extra Spaces, 284  
UVa 11966 - Galactic Bonding, 390  
UVa 11967 - Hic-Hac-Hoe, 390  
UVa 11968 - In The Airport, 233  
UVa 11970 - Lucky Numbers, 235  
UVa 11974 - Switch The Lights, 368  
UVa 11975 - Tele-loto, 95  
UVa 11984 - A Change in Thermal Unit, 28  
UVa 11986 - Save from Radiation, 235  
UVa 11988 - Broken Keyboard ... \*, 48

- UVa 11991 - Easy Problem from ... \*, 75  
UVa 11995 - I Can Guess ... \*, 57  
UVa 12004 - Bubble Sort \*, 233  
UVa 12005 - Find Solutions, 263  
UVa 12015 - Google is Feeling Lucky, 23  
UVa 12019 - Doom's Day Algorithm, 30  
UVa 12022 - Ordering T-shirts, 251  
UVa 12024 - Hats, 266  
UVa 12027 - Very Big Perfect Square, 233  
UVa 12028 - A Gift from ..., 388  
UVa 12030 - Help the Winners, 377  
UVa 12032 - The Monkey ... \*, 104  
UVa 12036 - Stable Grid \*, 236  
UVa 12043 - Divisors, 263  
UVa 12047 - Highest Paid Toll \*, 183  
UVa 12049 - Just Prune The List, 57  
UVa 12060 - All Integer Average \*, 30  
UVa 12068 - Harmonic Mean, 262  
UVa 12070 - Invite Your Friends, 389  
UVa 12083 - Guardian of Decency, 217,  
                223  
UVa 12085 - Mobile Casanova \*, 30  
UVa 12086 - Potentiometers, 76  
UVa 12100 - Printer Queue, 48  
UVa 12101 - Prime Path, 389  
UVa 12114 - Bachelor Arithmetic, 266  
UVa 12125 - March of the Penguins \*, 203  
UVa 12135 - Switch Bulbs, 368  
UVa 12136 - Schedule of a Married Man,  
                30  
UVa 12143 - Stopping Doom's Day, 243  
UVa 12144 - Almost Shortest Path, 183  
UVa 12148 - Electricity, 30  
UVa 12149 - Feynman, 232  
UVa 12150 - Pole Position, 47  
UVa 12155 - ASCII Diamondi \*, 284  
UVa 12157 - Tariff Plan, 23  
UVa 12159 - Gun Fight \*, 389  
UVa 12160 - Unlock the Lock \*, 182  
UVa 12168 - Cat vs. Dog, 223  
UVa 12186 - Another Crisis, 222  
UVa 12187 - Brothers, 47  
UVa 12190 - Electric Bill, 104  
UVa 12192 - Grapevine \*, 103  
UVa 12195 - Jingle Composing, 29  
UVa 12207 - This is Your Queue, 48  
UVa 12210 - A Match Making Problem \*,  
                111
- UVa 12238 - Ants Colony, 421  
UVa 12239 - Bingo, 27  
UVa 12243 - Flowers Flourish ..., 285  
UVa 12247 - Jollo \*, 27  
UVa 12249 - Overlapping Scenes, 96  
UVa 12250 - Language Detection, 23  
UVa 12256 - Making Quadrilaterals \*, 337  
UVa 12279 - Emoogle Balance, 23  
UVa 12289 - One-Two-Three, 23  
UVa 12290 - Counting Game, 233  
UVa 12291 - Polyomino Composer, 47  
UVa 12293 - Box Game, 273  
UVa 12318 - Digital Roulette, 390  
UVa 12319 - Edgetown's Traffic Jams, 192  
UVa 12321 - Gas Station, 110  
UVa 12324 - Philip J. Fry Problem, 376  
UVa 12342 - Tax Calculator, 29  
UVa 12346 - Water Gate Management, 96  
UVa 12347 - Binary Search Tree, 222  
UVa 12348 - Fun Coloring, 96  
UVa 12356 - Army Buddies \*, 47  
UVa 12364 - In Braille, 284  
UVa 12372 - Packing for Holiday, 23  
UVa 12376 - As Long as I Learn, I Live,  
                161  
UVa 12397 - Roman Numerals \*, 441  
UVa 12398 - NumPuzz I, 47  
UVa 12403 - Save Setu, 23  
UVa 12405 - Scarecrow \*, 110  
UVa 12406 - Help Dexter, 96  
UVa 12414 - Calculating Yuan Fen, 285  
UVa 12416 - Excessive Space Remover, 235  
UVa 12428 - Enemy at the Gates, 387  
UVa 12439 - February 29, 30  
UVa 12442 - Forwarding Emails \*, 161  
UVa 12455 - Bars \*, 96  
UVa 12457 - Tennis contest, 266  
UVa 12459 - Bees' ancestors, 243  
UVa 12460 - Careful teacher, 390  
UVa 12461 - Airplane, 266  
UVa 12463 - Little Nephew, 251  
UVa 12464 - Professor Lazy, Ph.D., 270  
UVa 12467 - Secret word, 290  
UVa 12468 - Zapping, 23  
UVa 12469 - Stones, 273  
UVa 12470 - Tribonacci, 429  
UVa 12478 - Hardest Problem ..., 24  
UVa 12482 - Short Story Competition, 111

- UVa 12485 - Perfect Choir, 111  
UVa 12488 - Start Grid, 95  
UVa 12498 - Ant's Shopping Mall, 95  
UVa 12502 - Three Families, 232  
UVa 12503 - Robot Instructions, 23  
UVa 12504 - Updating a Dictionary, 57  
UVa 12515 - Movie Police, 95  
UVa 12527 - Different Digits, 233  
UVa 12531 - Hours and Minutes, 30  
UVa 12532 - Interval Product \*, 76  
UVa 12541 - Birthdates, 48  
UVa 12542 - Prime Substring, 244  
UVa 12543 - Longest Word, 283  
UVa 12554 - A Special ... Song, 24  
UVa 12555 - Baby Me, 29  
UVa 12577 - Hajj-e-Akbar, 23  
UVa 12578 - 10:6:2, 336  
UVa 12582 - Wedding of Sultan, 161  
UVa 12583 - Memory Overflow, 95  
UVa 12592 - Slogan Learning of Princess,  
                  57
- UVa 12602 - Nice Licence Plates, 236  
UVa 12608 - Garbage Collection, 30
- Václav Chvátal, 396  
Vértice cortado, véase Puntos de articulación  
Vértices, capacidad de los, 200  
Vértices, división de, 200  
Vector, 39  
Vector (geometría), 324  
Ventana corredera, 44, 445  
Viajante bitónico, problema del, 397  
Viajante, problema del, 397  
Voraz, algoritmo, 104, 245
- Warshall, Stephen, 184, 188, 193  
Waterman, Michael S., 280  
Wunsch, Christian D., 280
- Zeckendorf, Edouard, 251  
Zeckendorf, teorema de, 245



