Análisis y diseño de algoritmos

0. Presentación

José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

Datos

Asignatura:

- **Titulación:** Grado en Ingeniería Informática
- **6 créditos ECTS:** 3 teóricos y 3 prácticos
- **Área de conocimiento:** Lenguajes y Sistemas Informáticos

Personales:

- **Horario de tutorias:**
  - lunes de 15:00 a 19:00
- **Correo electrónico:** oncina@ua.es

Metodología docente

Teoría

- Explicaciones del profesor
- Resolución de ejercicios y problemas por parte del alumno

Práctica

- Explicaciones del profesor
- Tipos de clase:
  - Cuaderno de prácticas:
    - problemas e implementacion que el alumno podrá resolver en cada sesión
    - cada trabajo tendrá una fecha límite de entrega
    - se presentará mediante Moodle
  - Ejercicio práctico:
    - resolución de un problema propuesto e implementación de su solución
    - durante las últimas semanas del curso

Evaluación convocatoria de junio (C3)

Prueba	Descripción	Ponderación
Cuaderno de prácticas	Resolución de problemas e implementación de algunos algoritmos propuestos	20 %
Práctica final	Durante las últimas semanas: Resolución, implementación y defensa de un ejercicio propuesto	10 %
Exámenes parciales	Se realizarán dos exámenes parciales	20 %
Examen final	Abarca todos los contenidos estudiados durante el curso	50 %

Evaluación convocatoria de junio (C3)

Prueba	Descripción	Ponderación
Cuaderno de prácticas	Resolución de problemas e implementación de algunos algoritmos propuestos	20 %
Práctica final	Durante las últimas semanas: Resolución, implementación y defensa de un ejercicio propuesto	10 %
Exámenes parciales	Se realizarán dos exámenes parciales	20 %
Examen final	Abarca todos los contenidos estudiados durante el curso	50 %

Nota final

- Para optar al aprobado hay que tener como mínimo un 4 en el examen final
  - si no se supera, la nota final de la convocatoria saturará en 4
- Para las demás pruebas no se establece mínimo alguno
- Los ejercicios de prácticas se presentarán exclusivamente en la semana que corresponda. Son trabajos no recuperables

Conv. extraordinarias: julio y diciembre (C4 y C1)

Prueba	**Descripción**	**(A)**	**(B)**
Cuaderno de prácticas	Resolución de problemas e implementación de algunos algoritmos propuestos	20 %	20 %
Práctica final	Durante las últimas semanas: Resolución, implementación y defensa de un ejercicio propuesto	10 %	10 %
Exámenes parciales	Se realizarán dos exámenes parciales	20 %	0 %
Examen final	Abarca todos los contenidos estudiados durante el curso	50 %	70 %

Conv. extraordinarias: julio y diciembre (C4 y C1)

Prueba	Descripción	(A)	(B)
Cuaderno de prácticas	Resolución de problemas e implementación de algunos algoritmos propuestos	20 %	20 %
Práctica final	Durante las últimas semanas: Resolución, implementación y defensa de un ejercicio propuesto	10 %	10 %
Exámenes parciales	Se realizarán dos exámenes parciales	20 %	0 %
Examen final	Abarca todos los contenidos estudiados durante el curso	50 %	70 %

Nota final

- Para estas convocatorias puedes:
  - volver a hacer un examen final
  - volver a entregar la práctica final
- Las notas obtenidas remplazan las anteriores
- El resto de las notas se guardan (dentro del mismo curso académico)
- La nota final es el máximo de las dos modalidades

FAQ

FAQ

- ¿Puedo cambiarme de turno?
  - Eso no depende de mi, solicitalo en secretaría

FAQ

- ¿Puedo cambiarme de turno?
  - Eso no depende de mi, solicitalo en secretaría
- ¡Se me ha pasado la entrega de la práctica por un segundo!
  - Lo siento ... Deberías haber hecho entregas parciales

FAQ

- ¿Puedo cambiarme de turno?
  - Eso no depende de mi, solicitalo en secretaría
- ¡Se me ha pasado la entrega de la práctica por un segundo!
  - Lo siento ... Deberías haber hecho entregas parciales
- ¿Cuándo saldrán las notas de las prácticas?
  - Eso no depende de mi, habla con el coordinador

FAQ

- ¿Puedo cambiarme de turno?
  - Eso no depende de mi, solicitalo en secretaría
- ¡Se me ha pasado la entrega de la práctica por un segundo!
  - Lo siento ... Deberías haber hecho entregas parciales
- ¿Cuándo saldrán las notas de las prácticas?
  - Eso no depende de mi, habla con el coordinador
- ¿Qué pasa si no puedo venir a un turno de prácticas?
  - Que os perdéis la explicación y posibles pistas que se den en clase
  - No se pasa lista

FAQ

- ¿Puedo cambiarme de turno?
  - Eso no depende de mi, solicitalo en secretaría
- ¡Se me ha pasado la entrega de la práctica por un segundo!
  - Lo siento ... Deberías haber hecho entregas parciales
- ¿Cuándo saldrán las notas de las prácticas?
  - Eso no depende de mi, habla con el coordinador
- ¿Qué pasa si no puedo venir a un turno de prácticas?
  - Que os perdéis la explicación y posibles pistas que se den en clase
  - No se pasa lista
- ¿Qué pasa si no entrego la práctica tal como se dice en el enunciado?
  - Normalmete eso conlleva un cero en la práctica

FAQ

- ¿Puedo cambiarme de turno?
  - Eso no depende de mi, solicitalo en secretaría
- ¡Se me ha pasado la entrega de la práctica por un segundo!
  - Lo siento ... Deberías haber hecho entregas parciales
- ¿Cuándo saldrán las notas de las prácticas?
  - Eso no depende de mi, habla con el coordinador
- ¿Qué pasa si no puedo venir a un turno de prácticas?
  - Que os perdéis la explicación y posibles pistas que se den en clase
  - No se pasa lista
- ¿Qué pasa si no entrego la práctica tal como se dice en el enunciado?
  - Normalmete eso conlleva un cero en la práctica
- ¿Qué pasa si me copio las prácticas?
  - Que os arriesgáis a que se os abra un expediente
  - Es el coordinador quien decide si hay copia o no

Análisis y diseño de algoritmos

1. Introducción

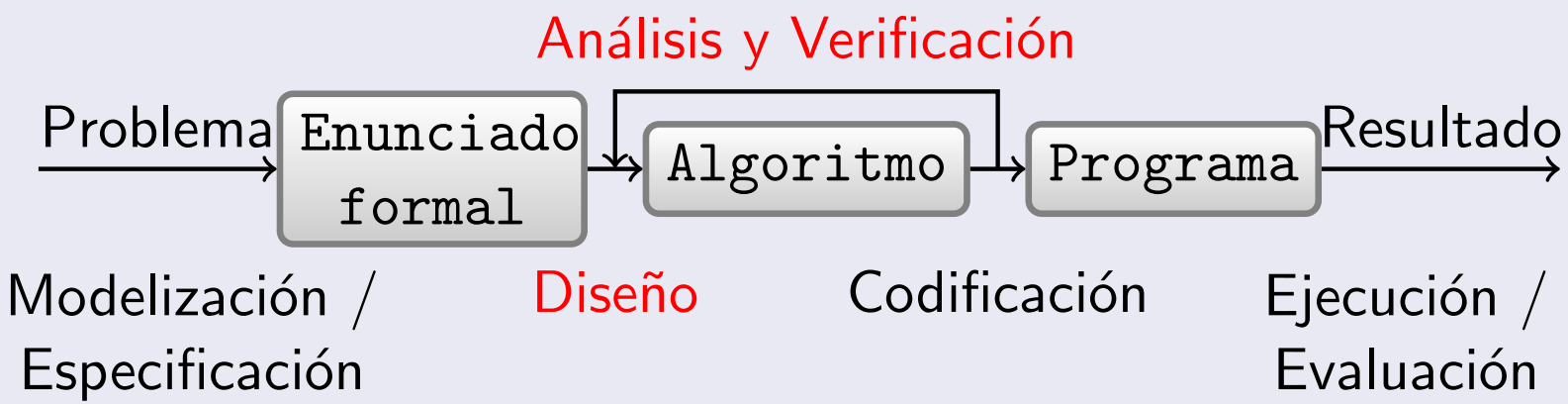
José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

Resolución de problemas

Etapas:



- Diseño y análisis de algoritmos.
  - Estudio de metodologías y técnicas que facilitan el diseño, análisis y la verificación de algoritmos

Verificación de algoritmos

- Finalidad: demostrar que un algoritmo funciona correctamente
  - Termina en un tiempo finito
  - Devuelve un resultado de acuerdo a su especificación

El análisis de algoritmos

- Finalidad: medir de forma cuantitativa la cantidad de recursos que un algoritmo necesita para su ejecución
- Recursos a analizar:
  - Tiempo que un algoritmo necesita para su ejecución
  - Espacio (memoria) que un algoritmo consume
- Finalidad:
  - Valoraciones: el algoritmo A es “bueno”, “el mejor”, “prohibitivo”
  - Comparaciones: el algoritmo A es mejor que el B

Diseño de algoritmos: paradigmas

- La resolución de problemas:
  - Diseño ad hoc (para ese fin)
    - Algoritmos dependientes del problema y no generalizables
    - Dificultad de adecuar cambios de especificación
  - Basado en paradigmas (= metodologías, esquemas, estrategias)
    - Cada esquema representa un grupo de algoritmos con características comunes (análogos)
    - Permiten la generalización y reutilización de algoritmos
    - Cada instanciación de un esquema da lugar a un algoritmo diferente
- El diseño de algoritmos estudia la aplicación de diferentes metodologías o paradigmas a la resolución de problemas en programación

Objetivos de la asignatura

- Dar a conocer las familias más importantes de problemas algorítmicos y estudiar diferentes esquemas o paradigmas de diseño aplicables para resolverlos
- Aprender a instanciar (particularizar) un esquema genérico para un problema concreto, identificando los datos y operaciones del esquema con las del problema, previa comprobación de que se satisfacen los requisitos necesarios para su aplicación
- Justificar la elección de un determinado esquema cuando varios de ellos pueden ser aplicables a un mismo problema

Paradigmas (o esquemas) que vamos a estudiar

- Divide y vencerás (*divide and conquer*)
- Programación dinámica (*dynamic programming*)
- Algoritmos voraces (*greedy methods*)
- Algoritmos de búsqueda y enumeración
  - Algoritmos de vuelta atrás (*backtracking*)
  - Ramificación y poda (*branch and bound*)

Programa de la asignatura

1. Introducción
2. Eficiencia
3. Divide y vencerás
4. Programación dinámica
5. Algoritmos voraces
6. Vuelta atrás
7. Ramificación y poda

Bibliografía básica

- **"Introduction to Algorithms (Third Edition)"**
  T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein
  MIT Press,2009
- **"Introducció a l'anàlisi i disseny d'algorismes"**
  Francesc J. Ferri, Jesús V. Albert, Gregorio Martín
  Universitat de València, 1998
- **"Técnicas de diseño de algoritmos"**
  Rosa Guerequeta y Antonio Vallecillo
  Universidad de Málaga, 1998
  - Disponible en formato pdf en:
    `http://www.lcc.uma.es/~av/Libro/Libro.zip`

Otros recursos disponibles

- Clases en vídeo
  - Coursera (`http://www.coursera.org`)
  - Youtube: "Lecture *: Data Structures and Algorithms - Richard Buckland, UNSW"
- Material propio
  - Estas transparencias: `http://www.dlsi.ua.es/~oncina/ada.pdf`
  - Por capítulo: `http://www.dlsi.ua.es/~oncina/ada-[0-7].pdf`
  - Apoyo matemáticas: `http://www.dlsi.ua.es/~oncina/am.pdf`
  - Apoyo programación: `http://www.dlsi.ua.es/~oncina/ap.pdf`
  - Vídeos del curso 2021: `https://bit.ly/3ApuN09`

Otros recursos disponibles

- Clases en vídeo
  - Coursera (`http://www.coursera.org`)
  - Youtube: "Lecture *: Data Structures and Algorithms - Richard Buckland, UNSW"
- Material propio
  - Estas transparencias: `http://www.dlsi.ua.es/~oncina/ada.pdf`
  - Por capítulo: `http://www.dlsi.ua.es/~oncina/ada-[0-7].pdf`
  - Apoyo matemáticas: `http://www.dlsi.ua.es/~oncina/am.pdf`
  - Apoyo programación: `http://www.dlsi.ua.es/~oncina/ap.pdf`
  - Vídeos del curso 2021: `https://bit.ly/3ApuN09`

¡Atencion!

El material propio está siempre en desarrollo, puede cambiar en cualquier momento

Más recursos

- Campus Virtual (y Moodle)
  - Materiales y anuncios
    - Apuntes, transparencias utilizadas por los otros profesores, ejercicios, etc.
    - Guía docente de la asignatura
    - Anuncios y avisos al alumnado
  - Tutorías electrónicas
- Tutorías presenciales
  - consultar en `http://www.dlsi.ua.es`
  - reservas en `http://www.dlsi.ua.es/alumnes/`

Análisis y diseño de algoritmos

2. Eficiencia

José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

Objetivos

- Proporcionar la capacidad para analizar con rigor la eficiencia de los algoritmos
  - Distinguir los conceptos de eficiencia en tiempo y en espacio
  - Entender y saber aplicar criterios asintóticos a los conceptos de eficiencia
  - Saber calcular la complejidad temporal o espacial de un algoritmo
  - Saber comparar, en cuanto a su eficiencia, distintas soluciones algorítmicas a un mismo problema

Contenido

Índice

¿Qué es un algoritmo?

Definición (Algoritmo)

Un algoritmo es una serie finita de instrucciones no ambiguas que expresa un método de resolución de un problema

Importante:

- la máquina sobre la que se define el algoritmo debe estar bien definida
- los recursos (usualmente tiempo y memoria) necesarios para cada paso elemental deben estar acotados
- El algoritmo debe terminar en un número finito de pasos

Noción de complejidad

Definición (Complejidad algorítmica)

Es una medida de los recursos que necesita un algoritmo para resolver un problema

Los recursos mas usuales son:

- Tiempo: complejidad temporal
- Memoria: complejidad espacial

Se suele expresar en función de la dificultad *a priori* del problema:

- Tamaño del problema: lo que ocupa su representación
- Parámetro representativo: *i.e.* la dimensión de una matriz

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	
Decir cuál es el mayor de 2 enteros	
Ordenar un vector de n enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 enteros	
Ordenar un vector de n enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 enteros	$2 \cdot 32$
Ordenar un vector de n enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 enteros	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 enteros	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 enteros	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

- Usualmente se omite el tamaño de enteros, reales, punteros, etc. si se asume que su tamaño está acotado

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 enteros	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

- Usualmente se omite el tamaño de enteros, reales, punteros, etc. si se asume que su tamaño está acotado
- ¿Cuántos bits se necesitan para codificar un entero positivo n arbitrariamente grande?

Atención:

La complejidad puede depender de cómo se codifique el problema

Ejemplo

Sumar uno a un entero arbitrariamente grande

- Complejidad constante si el entero se codifica en base uno
- Complejidad lineal si el entero se codifica en base dos

Normalmente se prohíben:

- codificaciones en base uno
- codificaciones no compactas

El tiempo de ejecución

El tiempo de ejecución de un algoritmo depende de:

Factores externos

- La máquina en la que se va a ejecutar
- El compilador
- Los datos de entrada suministrados en cada ejecución

Factores internos

- El número de instrucciones que ejecuta el algoritmo y su duración

¿Cómo estudiamos el tiempo de ejecución?

Definición (Análisis empírico o *a posteriori*)

Programar el algoritmo, ejecutar el programa para distintos valores de entrada y cronometrar el tiempo de ejecución

- ▲ Es una medida del comportamiento del algoritmo en su entorno
- ▼ El resultado depende de los factores externos e internos

Definición (Análisis teórico o *a priori*)

Obtener una función que represente el tiempo de ejecución (en operaciones elementales) del algoritmo para cualquier valor de entrada

- ▲ El resultado depende sólo de los factores internos
- ▲ No es necesario implementar y ejecutar los algoritmos
- ▼ No obtiene una medida real del comportamiento del algoritmo en el entorno de aplicación

Tiempo de ejecución de un algoritmo

Definición (Operaciones elementales)

Son las operaciones que realiza el ordenador. Su duración debe estar acotada por una constante

Ejemplo (Operaciones elementales)

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador
- Los saltos (llamadas a funciones, retorno desde ellos . . .)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores o matrices)

Tiempo de ejecución de un algoritmo

Para simplificar, se suele considerar que el coste temporal de las operaciones elementales es unitario

Definición (Tiempo de ejecución de un algoritmo)

Una función ($T(n)$) que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de problema n

Ejemplo: Suma de los elementos de un vector

Ejemplo (sintaxis de la STL)

```
1 int acc( const vector<int> &v){
2   int s = 0;
3
4   for(size_t i = 0; i < v.size(); i++)
5     s += v[i];
6
7   return s;
8 }
```

Ejemplo: Suma de los elementos de un vector

Ejemplo (sintaxis de la STL)

```
1 int acc( const vector<int> &v){
2   int s = 0;
3
4   for(size_t i = 0; i < v.size(); i++)
5     s += v[i];
6
7   return s;
8 }
```

Si estudiamos el bucle ($n = \texttt{v.size()}$):

n	asign.	comp.	inc.	total
0	1	1	0	2
1	1	2	1	4
2	1	3	2	6
⋮	⋮	⋮	⋮	⋮
n	1	$n+1$	n	$2+2n$

Ejemplo: Suma de los elementos de un vector

Ejemplo (sintaxis de la STL)

```
1 int acc( const vector<int> &v){
2   int s = 0;
3
4   for(size_t i = 0; i < v.size(); i++)
5     s += v[i];
6
7   return s;
8 }
```

Si estudiamos el bucle ($n = \texttt{v.size()}$):

n	asign.	comp.	inc.	total
0	1	1	0	2
1	1	2	1	4
2	1	3	2	6
⋮	⋮	⋮	⋮	⋮
n	1	$n+1$	n	$2+2n$

La complejidad del algoritmo será:

$$T(n) = \underbrace{1}_{\text{primera asignación}} + \underbrace{2+2n}_{\textit{bucle}} + \underbrace{n}_{\text{interior del bucle}} = 3+3n$$

Ejercicio: Traspuesta de una matriz cuadrada

Traspuesta de una matriz $d \times d$ (Sintaxis de la librería armadillo)

```
1 void transpose( mat& A){  // I assume that A.n_rows == A.n_cols
2   for( size_t i = 1; i < A.n_rows; i++ )
3     for( size_t j = 0; j < i ; j++ )
4       swap( A(i,j), A(j,i) );
5 }
```

Ejercicio: Traspuesta de una matriz cuadrada

Traspuesta de una matriz $d \times d$ (Sintaxis de la librería `armadillo`)

```
1 void transpose( mat& A){  // I assume that A.n_rows == A.n_cols
2   for( size_t i = 1; i < A.n_rows; i++ )
3     for( size_t j = 0; j < i ; j++ )
4       swap( A(i,j), A(j,i) );
5 }
```

Como la complejidad del bucle interior es: $2 + 3i$ veces

$$T_d(d) = \underbrace{2(d-1)+2}_{\text{bucle exterior}} + \underbrace{\sum_{i=1}^{d-1}(2+3i)}_{\text{interior}} = \cdots = O(d^2)$$

Ejercicio: Traspuesta de una matriz cuadrada

Traspuesta de una matriz $d \times d$ (Sintaxis de la librería `armadillo`)

```
1 void transpose( mat& A){  // I assume that A.n_rows == A.n_cols
2   for( size_t i = 1; i < A.n_rows; i++ )
3     for( size_t j = 0; j < i ; j++ )
4       swap( A(i,j), A(j,i) );
5 }
```

Como la complejidad del bucle interior es: $2 + 3i$ veces

$$T_d(d) = \underbrace{2(d-1)+2}_{\text{bucle exterior}} + \underbrace{\sum_{i=1}^{d-1}(2+3i)}_{\text{interior}} = \cdots = O(d^2)$$

Si queremos la complejidad con respecto al tamaño del problema ($s = d^2$):

$$T_s(s) = T_d(d) = O(d^2) = O(s)$$

Ejercicio: Producto de dos matrices cuadradas

Producto de dos matrices $d \times d$ (Sintaxis de la librería `armadillo`)

```
1  mat product( const mat &A, const mat &B ){
2    mat R(A.n_rows, B.n_cols);
3    for( size_t i = 0; i < A.n_rows; i++ )
4      for( size_t j = 0; j < B.n_cols; j++ ) {
5        double acc = 0.0;
6        for( size_t k = 0; k < A.n_cols; k++ )
7          acc += A(i,k) * B(k,j);
8        R(i,j) = acc;
9      }
10   }
11   return R;
12 }
```

Ejercicio: Producto de dos matrices cuadradas

Producto de dos matrices $d \times d$ (Sintaxis de la librería armadillo)

```
1  mat product( const mat &A, const mat &B ){
2    mat R(A.n_rows, B.n_cols);
3    for( size_t i = 0; i < A.n_rows; i++ )
4      for( size_t j = 0; j < B.n_cols; j++ ) {
5        double acc = 0.0;
6        for( size_t k = 0; k < A.n_cols; k++ )
7          acc += A(i,k) * B(k,j);
8        R(i,j) = acc;
9      }
10   }
11   return R;
12 }
```

- La complejidad de las líneas 6-7 es $O(d)$
- La complejidad de las líneas 4-9 es $O(d) + d \cdot O(d) = O(d^2)$
- La complejidad de las líneas 3-10 es $O(d) + d \cdot O(d^2) = O(d^3)$

La complejidad del algoritmo será: $T_d(d) = O(d^3)$

Ejercicio: Producto de dos matrices cuadradas

¿Cual es la complejidad con respecto al tamaño?

El tamaño del problema es $s = 2d^2$ por lo que $d = \sqrt{s/2}$

$$T_s(s) = T_d(d) = O\left(d^3\right) = O\left(\left(\sqrt{s/2}\right)^3\right) = O(s^{3/2})$$

¿Cual es la complejidad espacial?

- En la complejidad espacial no se tiene en cuenta lo que ocupa la codificación del problema.
- Solo se tiene en cuenta lo que es imputable al algoritmo.

$$M_s(s) = M_d(d) = O(d^2) = O\left(\left(\sqrt{s/2}\right)^2\right) = O(s)$$

Si el resultado es un argumento, la complejidad espacial sería $O(1)$

Ejercicios

- Dado un vector de enteros positivos v y el entero z
  - Devuelve el primer índice `i` tal que `v[i] == z`
  - Devuelve `NOT_FOUND` en caso de no encontrarlo

Búsqueda de un elemento

```
1 const int NOT_FOUND = -1;
2
3 int find( const vector<int> &v, int z ){
4   for( size_t i = 0; i < v.size(); i++ )
5     if( v[i] == z )
6       return i;
7   return NOT_FOUND;
8 }
```

Problema

- No podemos contar el número de pasos porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- En el ejemplo de la transparencia anterior:

v	z	Pasos
$(1, 0, 2, 4)$	1	3
$(1, 0, 2, 4)$	0	6
$(1, 0, 2, 4)$	2	9
$(1, 0, 2, 4)$	4	12
$(1, 0, 2, 4)$	5	14

- ¿Qué podemos hacer?
  - Acotar el coste mediante dos funciones que expresen respectivamente, el coste máximo y el coste mínimo del algoritmo (cotas de complejidad)

Índice

Cotas de complejidad

- Cuando aparecen diferentes casos para una misma talla n, se introducen las siguientes medidas de la complejidad
  - Caso peor: cota superior del algoritmo $\rightarrow C_{\max}(n)$
  - Caso mejor: cota inferior del algoritmo $\rightarrow C_{\min}(n)$
  - Caso promedio: coste promedio $\rightarrow C_{\text{avg}}(n)$
- Todas son funciones del tamaño del problema
- El coste promedio es difícil de evaluar a priori
  - Es necesario conocer la distribución de probabilidad de la entrada
  - ¡No es la media de la cota inferior y de la cota superior!

Cotas superior e inferior

Buscar elemento

```
1  #include <limits>
2
3  const size_t NOT_FOUND = numeric_limits<size_t>::max();
4
5  size_t find( const vector<int> &v, int z ) {
6    for( size_t i = 0; i < v.size(); i++ )
7      if( v[i] == z )
8        return i;
9    return NOT_FOUND;
10 }
```

Cotas superior e inferior

Buscar elemento

```
1  #include <limits>
2
3  const size_t NOT_FOUND = numeric_limits<size_t>::max();
4
5  size_t find( const vector<int> &v, int z ) {
6    for( size_t i = 0; i < v.size(); i++ )
7      if( v[i] == z )
8        return i;
9    return NOT_FOUND;
10 }
```

- En este caso el tamaño del problema es $n =$ `v.size()`

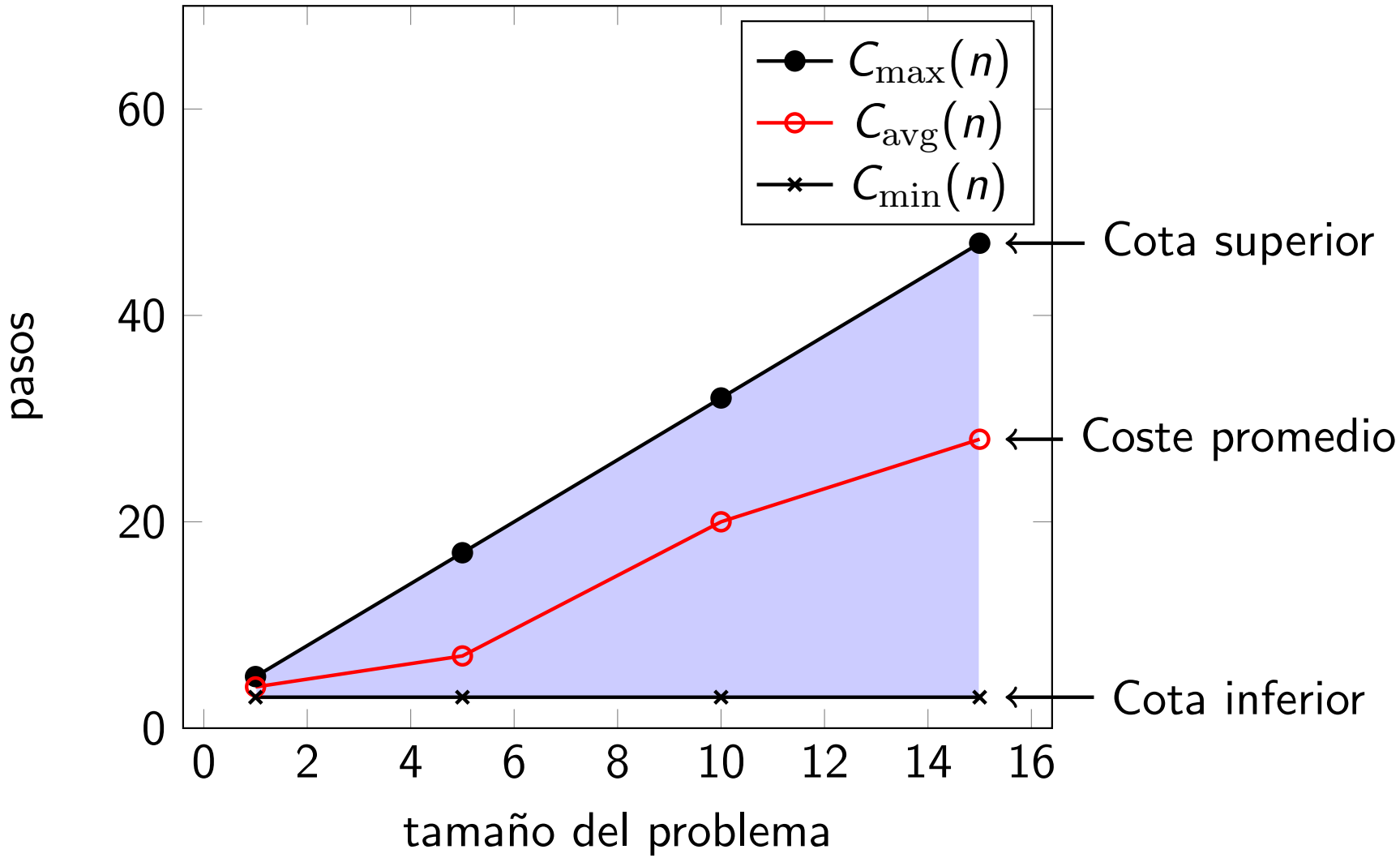
	Mejor caso	Peor caso
	$1+1+1+1$	$1+3n+1$
Suma	4	$3n+2$

Cotas:

$$C_{\max} = 3n+2 \quad \in O(n)$$
$$C_{\min} = 4 \quad \in O(1)$$

Cotas superior e inferior

- Coste de la función `find`



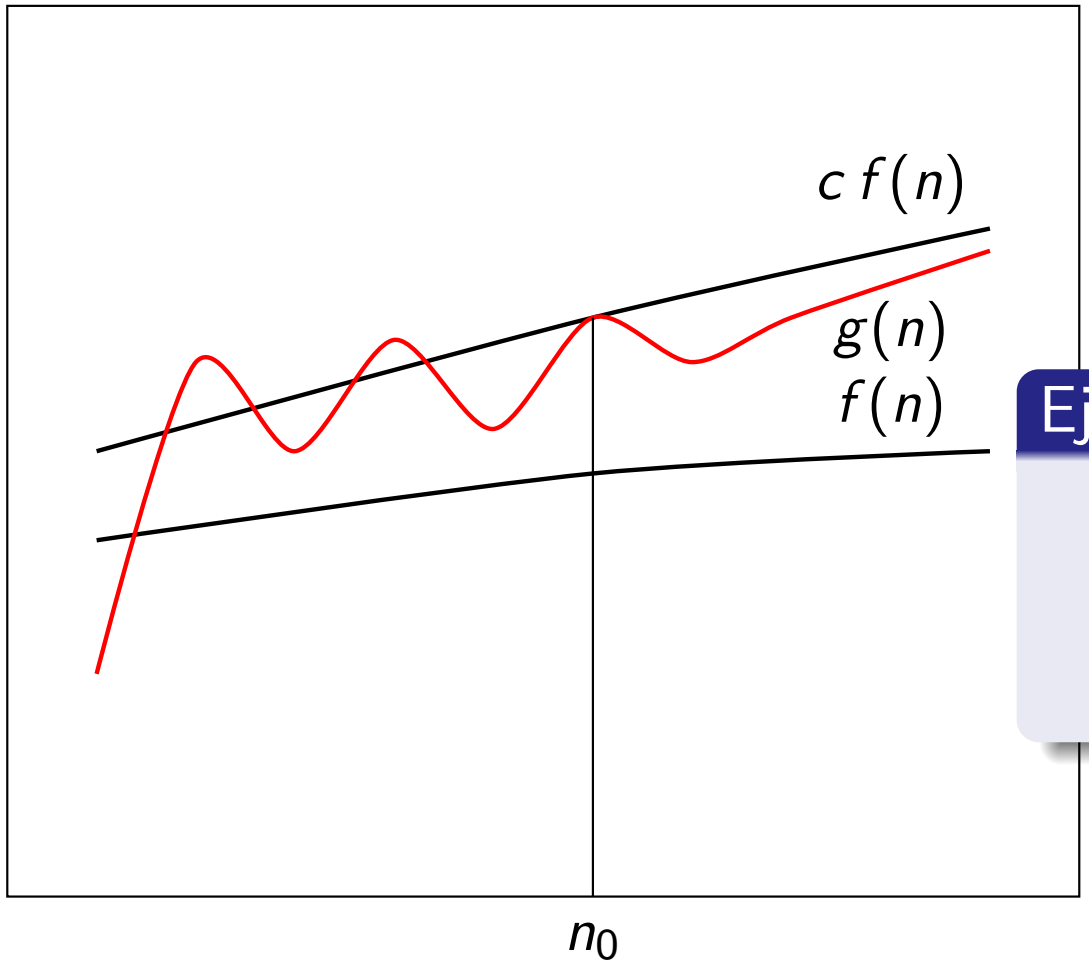
Análisis asintótico

- El estudio de la complejidad resulta interesante para tamaños grandes de problema por varios motivos:
  - Las diferencias "reales" en tiempo de ejecución de algoritmos con diferente coste para tamaños pequeños del problema no suelen ser significativas
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones
- Al estudio de la complejidad para tamaños grandes de problemas se le denomina análisis asintótico
  - Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente
  - Para ello, se definen clases de equivalencia que engloban a las funciones que "crecen de la misma forma".
- Se emplea la notación asintótica

Notación asintótica:

- Notación matemática utilizada para representar la complejidad cuando el tamaño de problema (n) crece ($n \to \infty$)
- Se definen tres tipos de notación:
  - Notación O (ómicron mayúscula, *big omicron* o *big O*) $\Rightarrow$ cota superior
  - Notación Ω (omega mayúscula o *big omega*) $\Rightarrow$ cota inferior
  - Notación Θ (zeta mayúscula o *big theta*)$\Rightarrow$ coste exacto

Cota superior. Notación O



Ejemplos:

- ¿$3n + 1 \in O(n)$?
- ¿$3n^2 + 1 \in O(n)$?
- ¿$3n^2 + 2 \in O(n^2)$?

Cota superior. Notación O

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define[1] el conjunto $O(f)$ como el conjunto de funciones acotadas superiormente por un múltiplo de f:

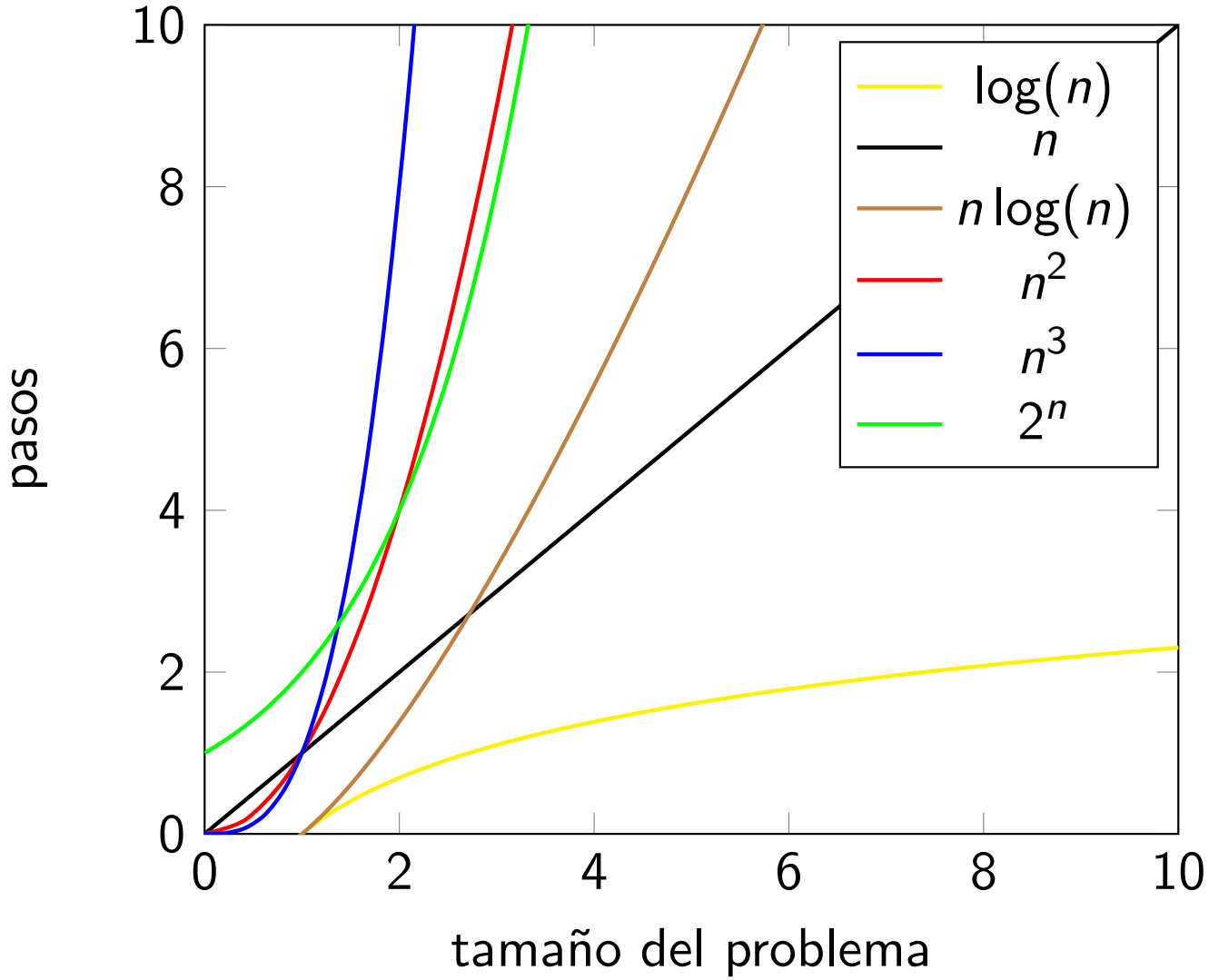
$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c > 0, \exists n_0 : \forall n > n_0 \\ 0 < g(n) \leqslant cf(n)\}$$

- Dada una función $g : \mathbb{N} \to \mathbb{R}^+$ se dice que $g \in O(f)$ si existe un múltiplo de f que es cota superior de g para valores grandes de n

[1] Según: `https://xlinux.nist.gov/dads/HTML/bigOnotation.html`

¿Para qué sirven?

- Permite agrupar en clases funciones con el mismo crecimiento



Propiedades

$$f \in O(f)$$

$$f \in O(g) \Rightarrow O(f) \subseteq O(g)$$

$$O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$$

$$f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$

$$f \in O(g) \wedge f \in O(h) \Rightarrow f \in O(\min\{g, h\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

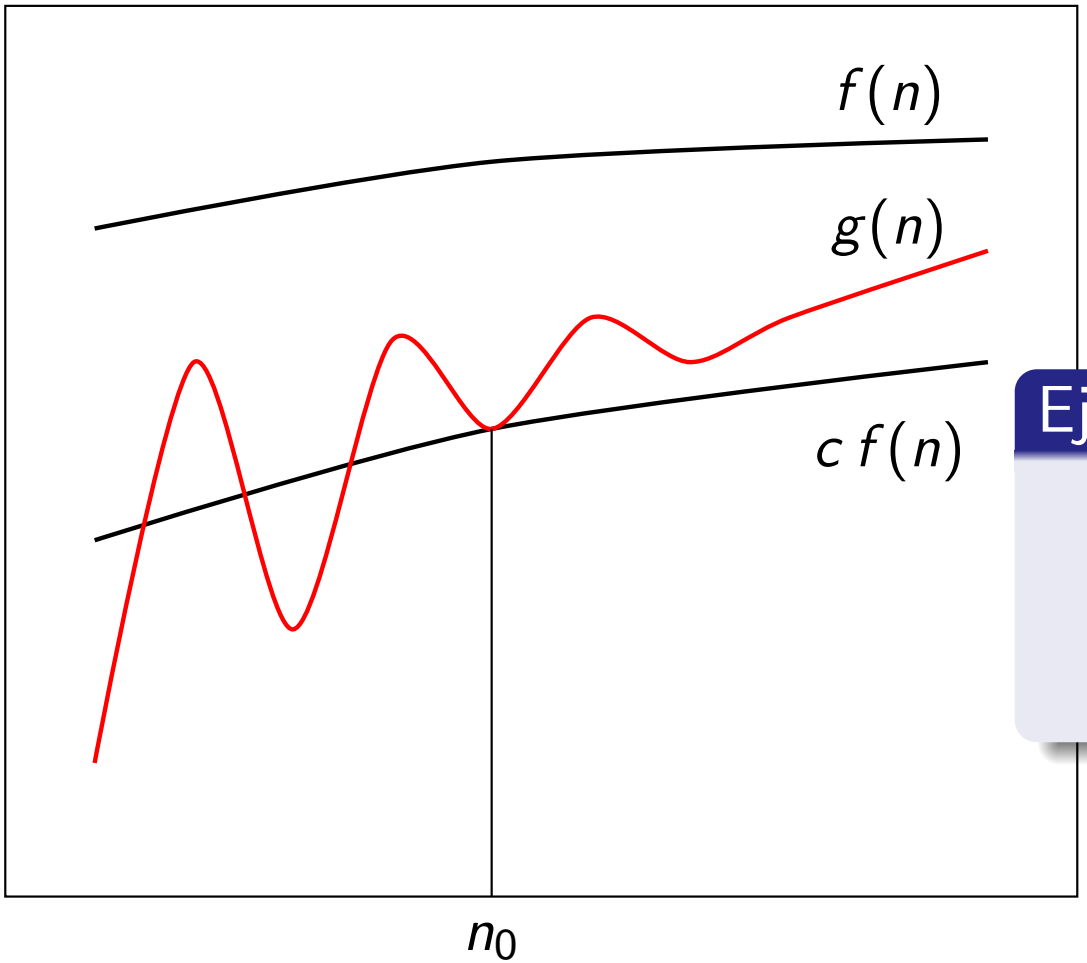
$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$$

$$f(n) = a_m n^m + \cdots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in O(n^m)$$

$$O(f) \subset O(g) \Rightarrow f \in O(g) \wedge g \notin O(f)$$

Cota inferior. Notación Ω



Ejemplos:

- ¿$3n + 1 \in \Omega(n)$?
- ¿$3n^2 + 1 \in \Omega(n)$?
- ¿$3n^2 + 2 \in \Omega(n^2)$?

Cota inferior. Notación Ω

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define[2] el conjunto $\Omega(f)$ como el conjunto de funciones acotadas inferiormente por un múltiplo de f:

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c > 0, \exists n_0 : \forall n > n_0, \\ 0 < cf(n) \leqslant g(n)\}$$

- Dada una función $g : \mathbb{N} \to \mathbb{R}^+$ se dice que $g \in \Omega(f)$ si existe un múltiplo de f que es cota inferior de g para valores grandes de n

[2] Según: texttthttps://xlinux.nist.gov/dads/HTML/omegaCapital.html

Propiedades

$$f \in \Omega(f)$$

$$f \in \Omega(g) \Rightarrow \Omega(f) \subseteq \Omega(g)$$

$$\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)$$

$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$$

$$f \in \Omega(g) \wedge f \in \Omega(h) \Rightarrow f \in \Omega(\max\{g, h\})$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(\min\{g_1, g_2\})$$

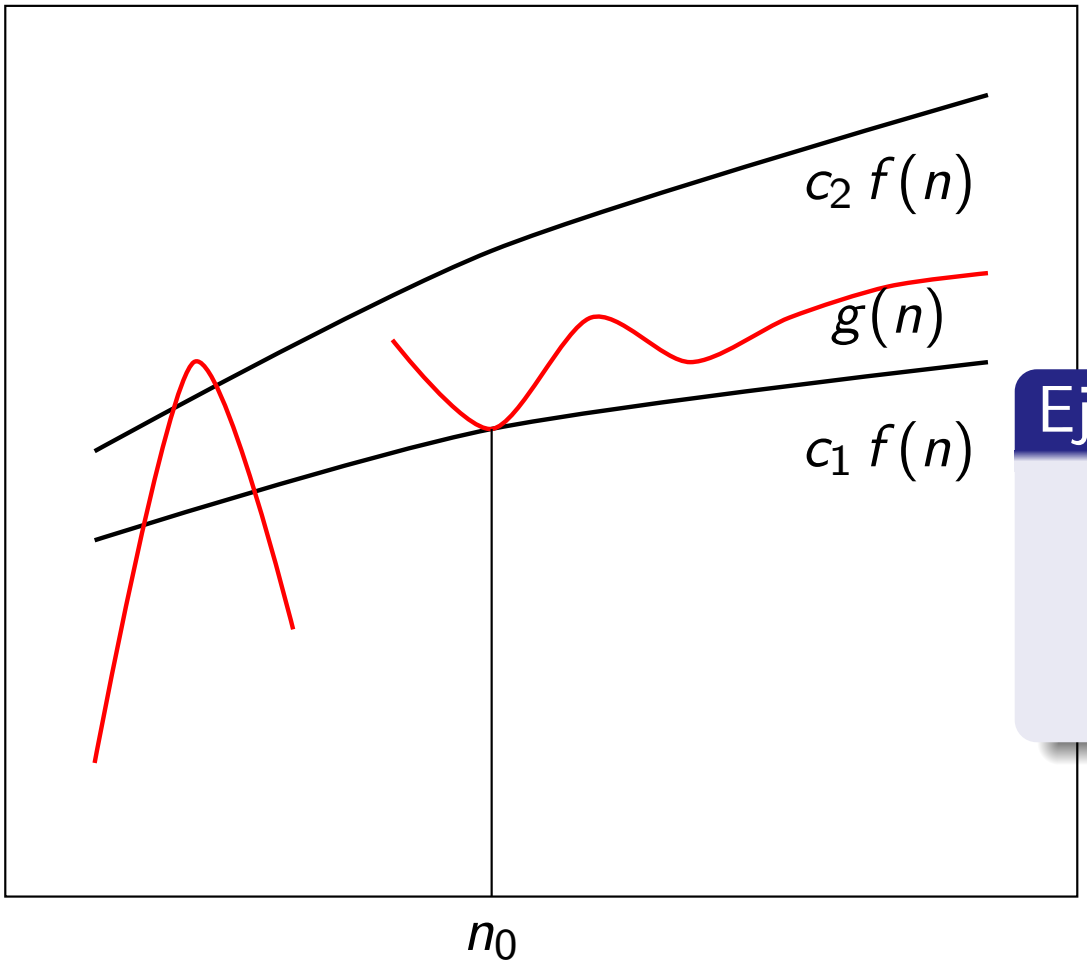
$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(g_1 + g_2)$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 f_2 \in \Omega(g_1 g_2)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f)$$

$$f(n) = a_m n^m + \cdots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in \Omega(n^m)$$

Coste exacto. Notación Θ



Ejemplos:

- ¿$3n + 1 \in \Theta(n)$?
- ¿$3n^2 + 1 \in \Theta(n)$?
- ¿$3n^2 + 2 \in \Theta(n^2)$?

Coste exacto. Notación Θ

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define[3] el conjunto $\Theta(f)$ como el conjunto de funciones acotadas superior e inferiormente por un múltiplo de f:

$$\Theta(f) = \{g : \mathbb{N} \to R^+ | \exists c_1 > 0, c_2 > 0, \exists n_0 : \forall n \geqslant n_0, \\ 0 < c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)\}$$

- O lo que es lo mismo: $\Theta(f) = O(f) \cap \Omega(f)$
- Dada una función $g : \mathbb{N} \to \mathbb{R}^+$ se dice que $g \in \Theta(f)$ si existen múltiplos de f que son a la vez cota superior y cota inferior de t para valores grandes de n

[3] Según: `https://xlinux.nist.gov/dads/HTML/theta.html`

Propiedades

$$f \in \Theta(f)$$

$$f \in \Theta(g) \Rightarrow \Theta(g) = \Theta(f)$$

$$\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g) \wedge g \in \Theta(f)$$

$$f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$$

$$f \in \Theta(g) \wedge f \in \Theta(h) \Rightarrow f \in \Theta(\max\{g, h\})$$

$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 + f_2 \in \Theta(g_1 + g_2)$$

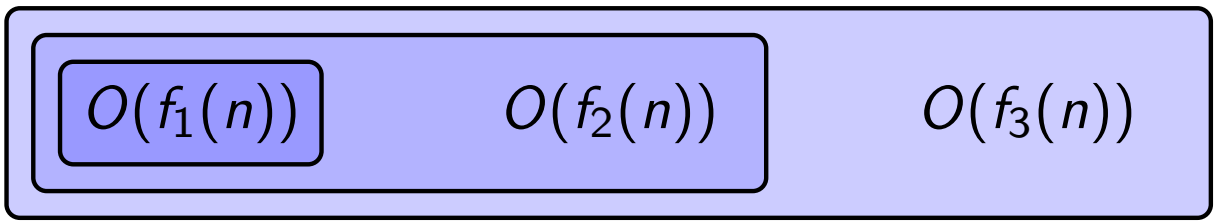
$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 f_2 \in \Theta(g_1 g_2)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = k, k \neq 0, k \neq \infty \Rightarrow \Theta(f) = \Theta(g)$$

$$f(n) = a_m n^m + \cdots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in \Theta(n^m)$$

Jerarquias de funciones

- Los conjuntos de funciones están incluidos unos en otros generando una ordenación de las diferentes funciones. Por ejemplo, para $O(\cdot)$,



- Las clases más utilizadas en la expresión de complejidades son:

$$\underbrace{O(1)}_{\text{constantes}} \subset \underbrace{O(\log\log n)}_{\text{sublogarítmicas}} \subset \underbrace{O(\log n) \subset O(log^{a(>1)} n)}_{\text{logarítmicas}}$$

$$\subset \underbrace{O(\sqrt{n})}_{\text{sublineales}} \subset \underbrace{O(n)}_{\text{lineales}} \subset \underbrace{O(n\log n)}_{\text{lineal-logarítmicas}}$$

$$\subset \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinómicas}} \subset \underbrace{O(2^n)}_{\text{exponenciales}} \subset \underbrace{O(n!) \subset O(n^n)}_{\text{superexponenciales}}$$

Índice

Cálculo de complejidades

- Pasos para obtener las cotas de complejidad
  1. Determinar la talla o tamaño del problema
  2. Determinar los casos mejor y peor (instancias para las que el algoritmo tarda más o menos)
     - Para algunos algoritmos, el caso mejor y el caso peor tienen la misma complejidad ya que se comportan de igual forma para cualquier instancia del mismo tamaño
  3. Obtención de las cotas para cada caso
     - Algoritmos iterativos
     - Algoritmos recursivos

Índice

Buscar elemento

```
1 size_t find( const vector<int> &v, int val ) {
2   for( size_t i = 0; i < v.size(); i++ )
3     if( v[i] == val )
4       return i;
5   return NOT_FOUND;
6 }
```

Línea	Cuenta Pasos		C. Asintótica	
	Mejor caso	Peor caso	Mejor caso	Peor caso
2	2	$2+2n$	$\Omega(1)$	$O(n)$
3	1	n	$\Omega(1)$	$O(n)$
4	1	0	$\Omega(1)$	–
5	0	1	–	$O(1)$
Suma	4	$3+3n$	$\Omega(1)$	$O(n)$

$$C_{\max}(n) = 3 + 3n$$
$$C_{\min}(n) = 4$$

$$C_{\max}(n) \in O(n)$$
$$C_{\min}(n) \in \Omega(1)$$

Ejercicio

Elemento máximo de un vector

```
1 int max( const vector<int> &v ) {  // assuming v.size() > 0
2   int max = v[0];
3   for( size_t i = 1; i < v.size(); i++ )
4     if( v[i] > max )
5       max = v[i];
6   return max;
7 }
```

Ejemplo

Ordenación por selección directa

```
1 void selection_sort( vector<T> &v) {
2     for( size_t i = 0; i < v.size()-1; i++) {
3         size_t i_min = i;
4         for( size_t j = i+1; j < v.size(); j++)
5             if (v[j] < v[i_min])
6                 i_min = j;
7         swap(v[i],v[i_min]);
8     }
9 }
```

Ejemplo

Búsqueda binaria en un vector ordenado

```
1  const size_t NOT_FOUND = numeric_limits<size_t>::max();
2
3  int binary_search( const vector<int> &v, int val ) {  // assume v is sorted
4    size_t first = 0;
5    size_t count = v.size();
6    while( count > 0 ) {
7      size_t step = count/2;
8      size_t med = first + step;
9      if( v[med] < val ) {
10       first = med + 1;
11       count -= step + 1;
12     } else
13       count = step;
14   }
15
16   if( first < v.size() && v[first] == val )
17     return first;
18   else
19     return NOT_FOUND;
20 }
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```

$$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```

$$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```

$$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```

$$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
for( int i = 1; i < n; i*=2 )
  {...}
```

$$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$$

$$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
for( int i = 1; i < n; i*=2 )
  {...}
```

$$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$$

$$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$$

$$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
for( int i = 1; i < n; i*=2 )
  {...}
for( int i = n; i > 0; i/=2 )
  {...}
```

$$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$$

$$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$$

$$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```

$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```

$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```

$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```

$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```

$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```

$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```

$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```

$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```

$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```

$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```

$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```

$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```

$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 1; j < n; j*=2 )
    {...}
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(\log^2(n))$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(\log^2(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < i; j*=2 )
    {...}
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(\log^2(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < i; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(i)} 1 \in \Theta(n \log(n))$

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(\log^2(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < i; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(i)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < i; j++ )
    {...}
```

Situaciones típicas

```
for( int i = 0; i < n; i+=2 )
  {...}
```
$\equiv \sum_{i=0}^{\frac{n}{2}} 1 \in \Theta(n)$

```
for( int i = 1; i < n; i++ )
  for( int j = 0; j < i; j++)
    {...}
```
$\equiv \sum_{i=1}^{n-1} \sum_{j=0}^{j=i-1} 1 = \sum_{i=1}^{n-1} i \in \Theta(n^2)$

```
for( int i = 1; i < n; i*=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = n; i > 0; i/=2 )
  {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} 1 = \log_2(n) \in \Theta(\log(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < n; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{n-1} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 1; j < n; j*=2 )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=1}^{\log_2(n)} 1 \in \Theta(\log^2(n))$

```
for( int i = 0; i < n; i++ )
  for( int j = 1; j < i; j*=2 )
    {...}
```
$\equiv \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2(i)} 1 \in \Theta(n \log(n))$

```
for( int i = 1; i < n; i*=2 )
  for( int j = 0; j < i; j++ )
    {...}
```
$\equiv \sum_{i=1}^{\log_2(n)} \sum_{j=0}^{2^i} 1$
$= 2^0 + 2^1 + 2^2 + \cdots + \frac{n}{2} + n < 2n \in \Theta(n)$

Índice

Ejemplo

Búsqueda en un vector ordenado

```
1  // Returns the index of the first element which is lower or equal to val.
2  size_t lower_bound( const vector<int> &v, int val, size_t first, int count) {
3    if( count == 0 )
4      return first;
5
6    size_t step = count/2;
7    size_t med = first + step;
8    if( v[med] < val ) {
9      return lower_bound( v, val, med+1, count - (step + 1));
10   } else
11     return lower_bound( v, val, first, step );
12 }
13
14 int binary_search( const vector<int> &v, int val ) {
15   size_t first = lower_bound( v, val, 0, v.size() );
16
17   if( first < v.size() && v[first] == val )
18     return first;
19   else
20     return NOT_FOUND;
21 }
```

Algoritmos recursivos

- Dado un algoritmo recursivo:

Búsqueda binaria

```
1  size_t lower_bound( const vector<int> &v, int val, size_t first, size_t count) {
2    if( count == 0 )
3      return first;
4
5    size_t step = count/2;
6    size_t med = first + step;
7    if( v[med] < val ) {
8      return lower_bound( v, val, med+1, count - (step + 1));
9    } else
10     return lower_bound( v, val, first, step );
11 }
```

- El coste depende de las llamadas recursivas, y, por tanto, debe definirse recursivamente:

$$T(n) \in \begin{cases} \Theta(1) & n = 0 \\ \Theta(1) + T(n/2) & n > 0 \end{cases} \qquad (n = \texttt{count})$$

Relaciones de recurrencia

- Una relación de recurrencia es una expresión que relaciona el valor de una función f definida para un entero n con uno o más valores de la misma función para valores menores que n
- Ejemplo: Los números de Fibonacci:

$$f(n) = \begin{cases} n & n \leqslant 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

- Esta es una ecuación lineal homogénea de segundo grado
- En este curso trabajaremos con lineales de primer grado:

$$f(n) = \begin{cases} P'(n) & n \leqslant n_0 \\ a\, f(F(n)) + P(n) & n > n_0 \end{cases}$$

Donde:

- $a \in \mathbb{N}$ es una constante
- $P(b), P'(n)$ son funciones de n
- $F(n) < n$ (normalmente $n - b$ con $b > 0$, o n/b con $b > 1$)

Algoritmos recursivos

- Las relaciones de recurrencia se usan para expresar la complejidad de un algoritmo recursivo aunque también son aplicables a los iterativos
- Si el algoritmo dispone de mejor y peor caso, puede haber una relación de recurrencia para cada caso
- La complejidad de un algoritmo se obtiene en tres pasos:
  1. Determinación de la talla del problema
  2. Obtención de las relaciones de recurrencia del algoritmo
  3. Resolución de las relaciones
- Para resolverlas, usaremos el método de sustitución:
  - Es un método sencillo
  - Sólo para funciones lineales (sólo una vez en función de sí mismas)
  - Consiste en sustituir cada $f(n)$ por su valor al aplicarle de nuevo la función hasta obtener un término general

Ordenación por selección

- Ejemplo: Ordenar un vector a partir del elemento `first`:

Ordenación por selección (recursivo)

```
1  void sort( vector<int> &v, size_t first) {
2    if( first == v.size() )
3      return;
4    int min = first;
5    for( size_t i = first + 1; i < v.size(); i++ )
6      if( v[i] < v[min] )
7        min = i;
8    swap( v[min], v[first]);
9    sort(v, first + 1);
10 }
```

- Obtener ecuación de recurrencia a partir del algoritmo:

$$T(n) = \begin{cases} \Theta(1) & n = 0 \\ \Theta(n) + T(n-1) & n > 0 \end{cases}$$

donde $n = \texttt{v.size()} - \texttt{first}$.

Ecuación de recurrencia

- Resolviendo la recurrencia por sustitución

$$\begin{aligned} T(n) &\overset{1}{=} n + T(n-1) \\ &\overset{2}{=} n + (n-1) + T(n-2) \\ &\overset{3}{=} n + (n-1) + (n-2) + T(n-3) = \ldots \\ &\overset{k}{=} \sum_{i=0}^{k-1} (n-i) + T(n-k) \end{aligned}$$

Pararemos cuando $n - k = 0$, o sea, cuando $k = n$

$$\overset{n}{=} \sum_{i=0}^{n-1} (n-i) + T(0) = \sum_{j=1}^{n} j + 1 = \frac{n(n+1)}{2} + 1$$

Entonces:

$$T(n) \in \Theta(n^2)$$

Algoritmo de ordenación por partición o *Quicksort*

- Elemento pivote: sirve para dividir en dos partes el vector. Su elección define variantes del algoritmo
  - Al azar
  - Primer elemento (Quicksort primer elemento)
  - Elemento central (Quicksort central)
  - Elemento mediana (Quicksort mediana)
- Pasos:
  - Elección del pivote
  - Se divide el vector en dos partes:
    - parte izquierda del pivote (elementos menores)
    - parte derecha del pivote (elementos mayores)
  - Se hacen dos llamadas recursivas. Una con cada parte del vector

Quicksort (pivote primer elemento)

Quicksort

```
1  void quicksort( vector<int> &v,
2    size_t first /* included */, size_t last /* not included (past-the-end) */
3  ) {
4    if( last - first < 2 ) return;
5
6    size_t p = first, l = last;
7
8    while( p+1 != l ) {
9      if( v[p + 1] < v[p] ) {
10       swap( v[p+1], v[p] );
11       p++;
12     } else {
13       l--;
14       swap( v[p+1], v[l] );
15     }
16   }
17   quicksort( v, first, p );
18   quicksort( v, p+1, last);
19 }
20
21 void quicksort( vector<int> &v ) { quicksort( v, 0, v.size() ); }
```

Quicksort. Complejidad temporal

- Tamaño del problema: n
  - Mejor caso: subproblemas $(n/2,\ n/2)$

$$T(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2}) & n > 1 \end{cases}$$

  - Peor caso: subproblemas $(0,\ n-1)$ o $(n-1,\ 0)$

$$T(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(n) + T(0) + T(n-1) & n > 1 \end{cases}$$

Quicksort. Complejidad temporal

- Mejor caso:

$$T(n) \overset{1}{=} n + 2T\left(\frac{n}{2}\right)$$
$$\overset{2}{=} n + 2\left(\frac{n}{2} + 2T\left(\frac{n}{2^2}\right)\right) = 2n + 2^2 T\left(\frac{n}{2^2}\right)$$
$$\overset{3}{=} 2n + 2^2\left(\frac{n}{2^2} + 2T\left(\frac{n}{2^3}\right)\right) = 3n + 2^3 T\left(\frac{n}{2^3}\right)$$
$$\overset{k}{=} kn + 2^k T\left(\frac{n}{2^k}\right)$$

La recursión termina cuando $n/2^k = 1$ por lo que habrá $k = \log_2 n$ llamadas recursivas

$$= n\log_2 n + nT(1) = n\log_2 n + n$$

Por tanto,

$$T(n) \in \Omega(n\log_2 n)$$

Quicksort. Complejidad temporal

- Peor caso:

$$\begin{aligned} T(n) &\overset{1}{=} n + T(n-1) \\ &\overset{2}{=} n + (n-1) + T(n-2) \\ &\overset{3}{=} n + (n-1) + (n-2) + T(n-3) \\ &\overset{k}{=} n + (n-1) + (n-2) + \cdots + T(n-k) \end{aligned}$$

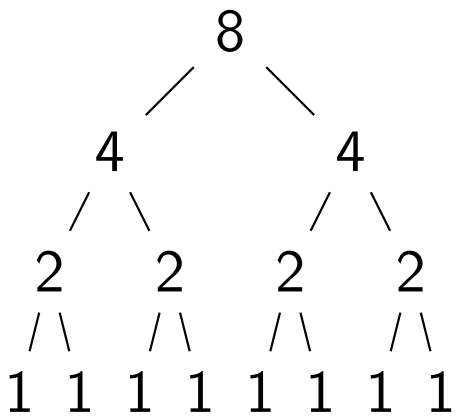
La recursión termina cuando $n - k = 1$ por lo que habrá $k = n - 1$ llamadas recursivas

$$= n + (n-1) + \cdots + 3 + 2 + T(1) = \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

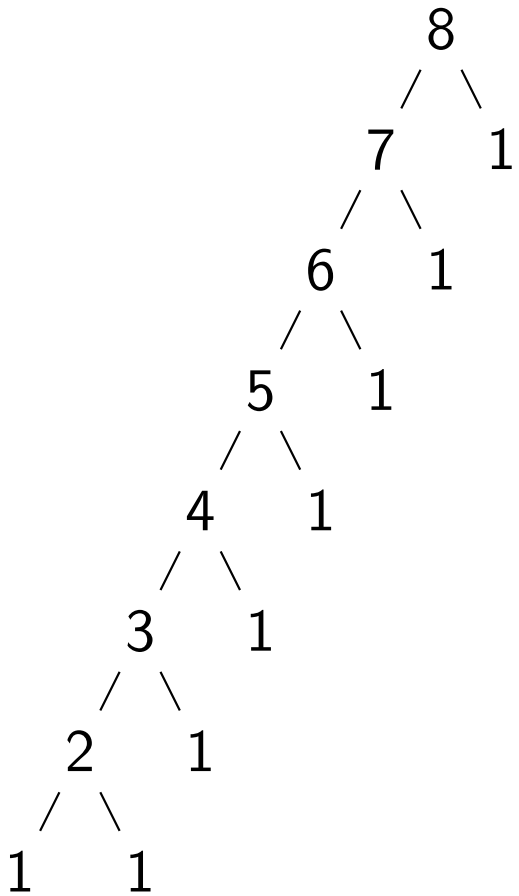
Por tanto,

$$T(n) \in O(n^2)$$

Quicksort



Caso mejor
$\Omega(n \log_2 n)$



Peor caso
$O(n^2)$

Quicksort mediana

- El caso mejor es cuando el pivote es la mediana
- Obtener la mediana
  - Coste menor que $O(n \log n)$
  - Se aprovecha el recorrido para reorganizar elementos y para encontrar la mediana en la siguiente llamada
  - Su complejidad es por tanto de $\Theta(n \log n)$

Quicksort. Complejidad espacial

- Tamaño del problema: n
  - Mejor caso: subproblemas $(n/2,\ n/2)$

$$M(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(1) + M(\frac{n}{2}) & n > 1 \end{cases}$$

  - Peor caso: subproblemas $(0,\ n-1)$ o $(n-1,\ 0)$

$$M(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(1) + M(n-1) & n > 1 \end{cases}$$

Quicksort. Complejidad espacial

- Mejor caso:

$$\begin{aligned} M(n) &\overset{1}{=} 1 + M\left(\frac{n}{2}\right) \\ &\overset{2}{=} 1 + \left(1 + M\left(\frac{n}{2^2}\right)\right) = 2 + M\left(\frac{n}{2^2}\right) \\ &\overset{3}{=} 2 + \left(1 + M\left(\frac{n}{2^3}\right)\right) = 3 + M\left(\frac{n}{2^3}\right) \\ &\overset{k}{=} k + M\left(\frac{n}{2^k}\right) \end{aligned}$$

La recursión termina cuando $n/2^k = 1$ por lo que habrá $k = \log_2 n$ llamadas recursivas

$$= \log_2 n + M(1) = \log_2 n + 1$$

Por tanto,

$$M(n) \in \Omega(\log_2 n)$$

Quicksort. Complejidad espacial

- Peor caso:

$$\begin{aligned} M(n) &\overset{1}{=} 1 + M(n-1) \\ &\overset{2}{=} 2 + M(n-2) \\ &\overset{3}{=} 3 + M(n-3) \\ &\overset{k}{=} k + M(n-k) \end{aligned}$$

La recursión termina cuando $n - k = 1$ por lo que habrá $k = n - 1$ llamadas recursivas

$$= n - 1 + M(1) = n$$

Por tanto,

$$M(n) \in O(n)$$

Algoritmo de ordenación por montículo o *Heapsort*

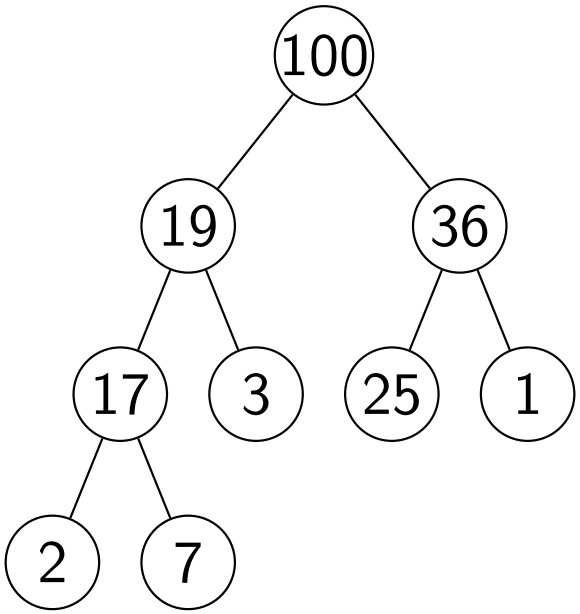
Monticulo (Max heap)

- Implementación del tipo de datos abtracto llamado *cola de prioridad* (`priority_queue` en las `STL`)
- Operaciones:
  - inserción (`push`)
  - extracción del elemento mas grande (`pop` y `top`)
- Es una estructura de árbol binario completo
  - niveles llenos (menos el último que lo está de izquierda a derecha)
- cumple la *propiedad del montículo*
  - El valor del padre siempre es mayor que los valores de sus hijos
  - El valor de la raiz del arbol es el elemento mayor de la estructura

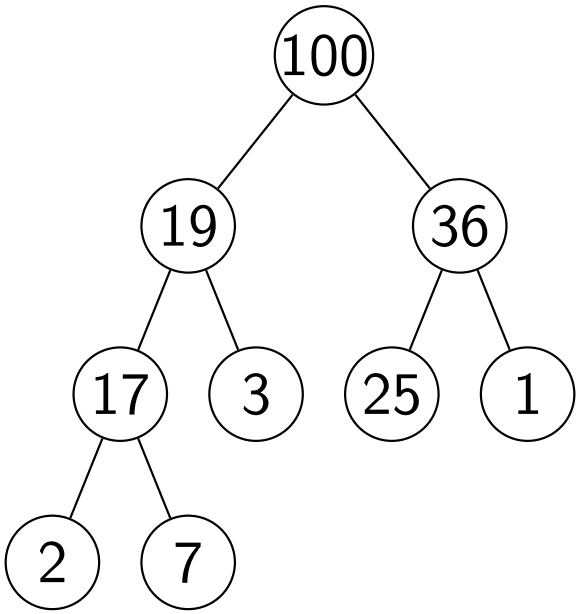
Nota:

Tambien existe el *Min heap*

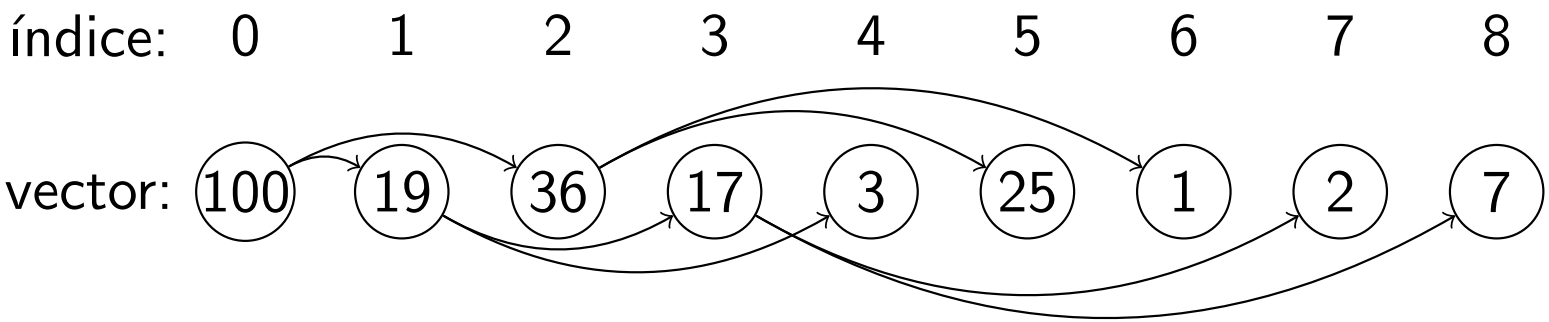
Ejemplo de max heap



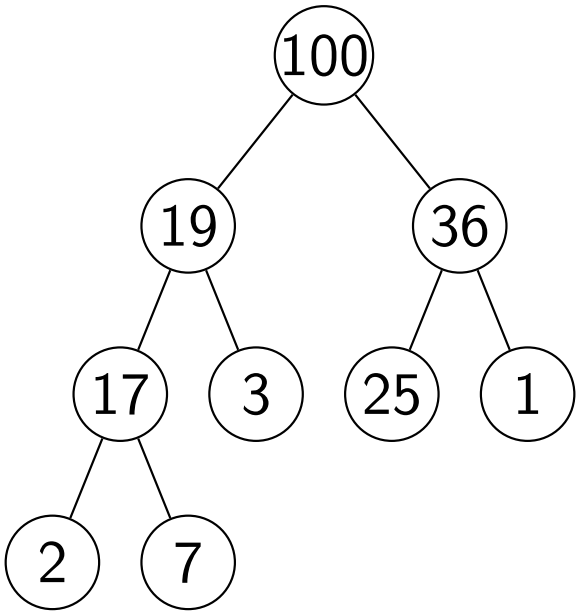
Ejemplo de max heap



Este árbol se almacena en un vector:



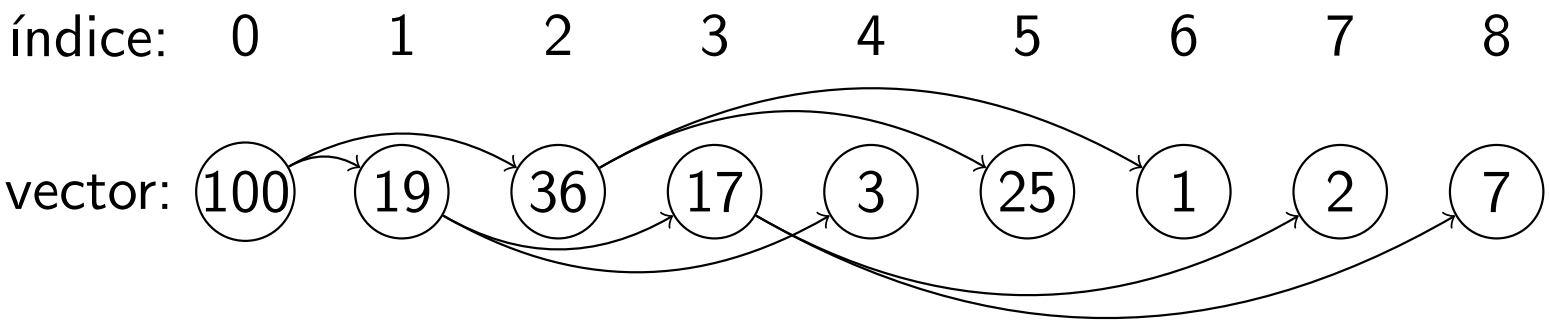
Ejemplo de max heap



relaciones:

```
1    root() = 0
2    left_child(i) = 2 * i + 1
3    right_child(i) = 2*i+2
4    parent(i) = (i-1)/2
```

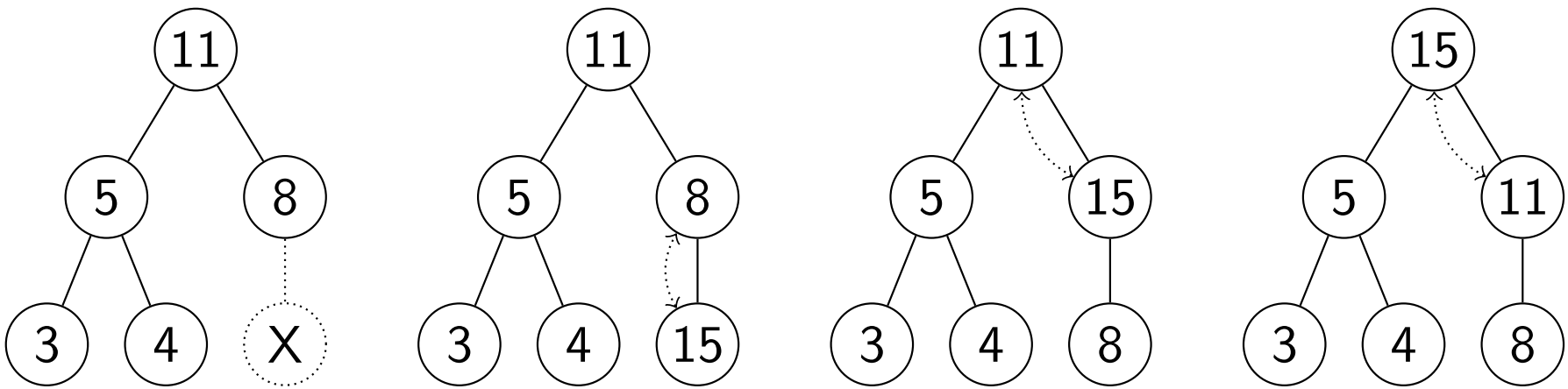
Este árbol se almacena en un vector:



Inserción

- añade el elemento al final del array que representa el heap
- compara el elemente con su padre. Si es menor, para
- si no, itercambialo con su padre y vuelve al paso anterior

Ejemplo: insercion de 15

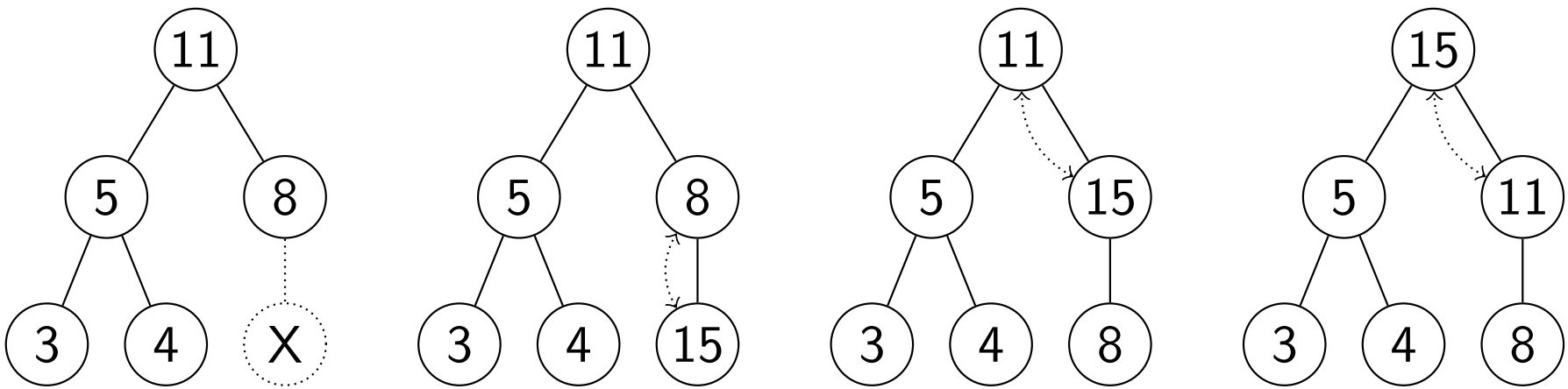


¿Complejidad?

Inserción

- añade el elemento al final del array que representa el heap
- compara el elemente con su padre. Si es menor, para
- si no, itercambialo con su padre y vuelve al paso anterior

Ejemplo: insercion de 15

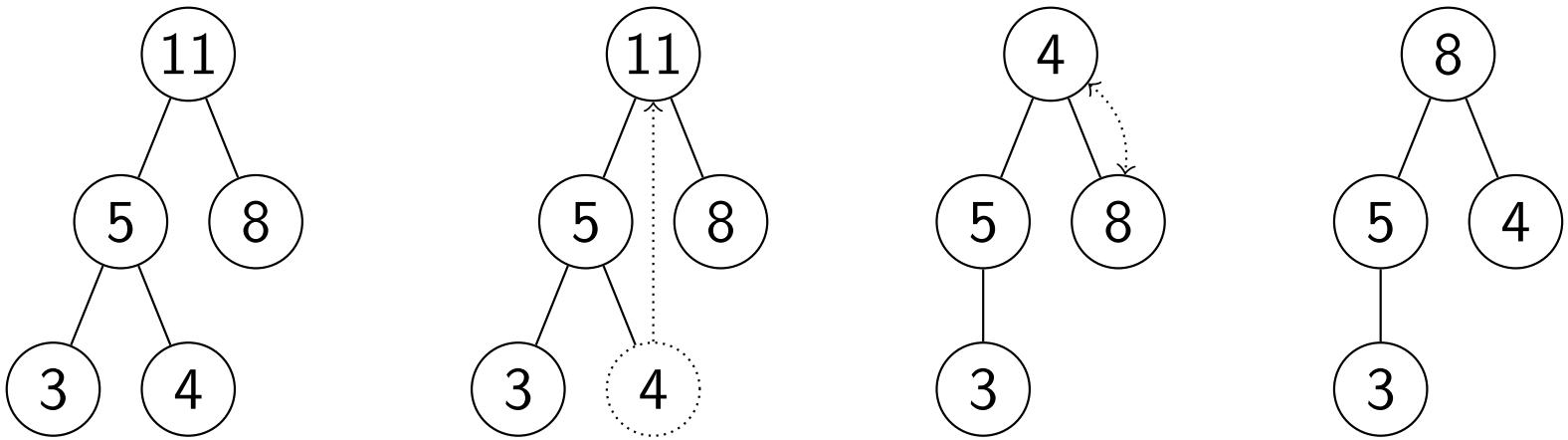


¿Complejidad? $O(\log(n))$

Extraccion (top)

- remplaza la raiz del nodo por el último elemento
- compara el nuevo elemento con sus hijos. Si es mayor, para
- intercambia el elemento con el menor de sus hijos y vuelve al paso anterior

Ejemplo: extracción

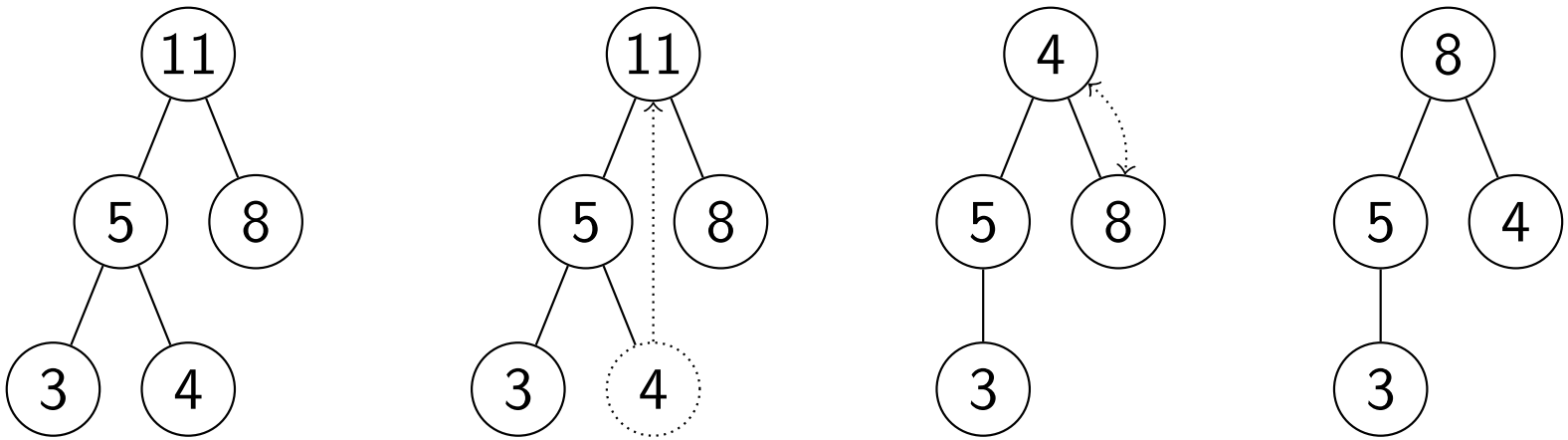


¿Complejidad?

Extraccion (top)

- remplaza la raiz del nodo por el último elemento
- compara el nuevo elemento con sus hijos. Si es mayor, para
- intercambia el elemento con el menor de sus hijos y vuelve al paso anterior

Ejemplo: extracción



¿Complejidad? $O(\log(n))$

Algoritmo Sink

Sink (segunda parte de extraer)

```
1  void sink(int *v, size_t n, size_t root) {
2    do{
3      size_t largest = root;  // Initialize largest as root
4      size_t l = 2*root + 1;  // left child
5      size_t r = 2*root + 2;  // right child
6
7      if (l < n && v[l] > v[largest]) // Is left child larger than root?
8        largest = l;
9
10     if (r < n && v[r] > v[largest]) // Is right child larger than largest so far?
11       largest = r;
12
13     if (largest == root) break; // If largest is still root then end
14
15         // If not, swap new largest with current node i and repeat.
16     swap(v[root], v[largest]);
17     root=largest;
18   } while (true);
19 }
```

¿Complejidad?

Algoritmo Sink

Sink (segunda parte de extraer)

```
1  void sink(int *v, size_t n, size_t root) {
2    do{
3      size_t largest = root;  // Initialize largest as root
4      size_t l = 2*root + 1;  // left child
5      size_t r = 2*root + 2;  // right child
6
7      if (l < n && v[l] > v[largest]) // Is left child larger than root?
8        largest = l;
9
10     if (r < n && v[r] > v[largest]) // Is right child larger than largest so far?
11       largest = r;
12
13     if (largest == root) break; // If largest is still root then end
14
15         // If not, swap new largest with current node i and repeat.
16     swap(v[root], v[largest]);
17     root=largest;
18   } while (true);
19 }
```

¿Complejidad? $T_{\text{sink}}(n) \in O(\log(n))$

Algoritmo de ordenación por montículo o *Heapsort*

Heapsort

- Construir un max heap con los datos
- ir extrayendo elementos del heap y colocarlos en el hueco que queda
- repetir hasta que se vacie el heap

¿Cómo construir el heap?

- Una forma es ir insertando los elementos de uno en uno en el heap
- complejidad $O(n \log(n))$

Algoritmo de ordenación por montículo o *Heapsort*

Heapsort

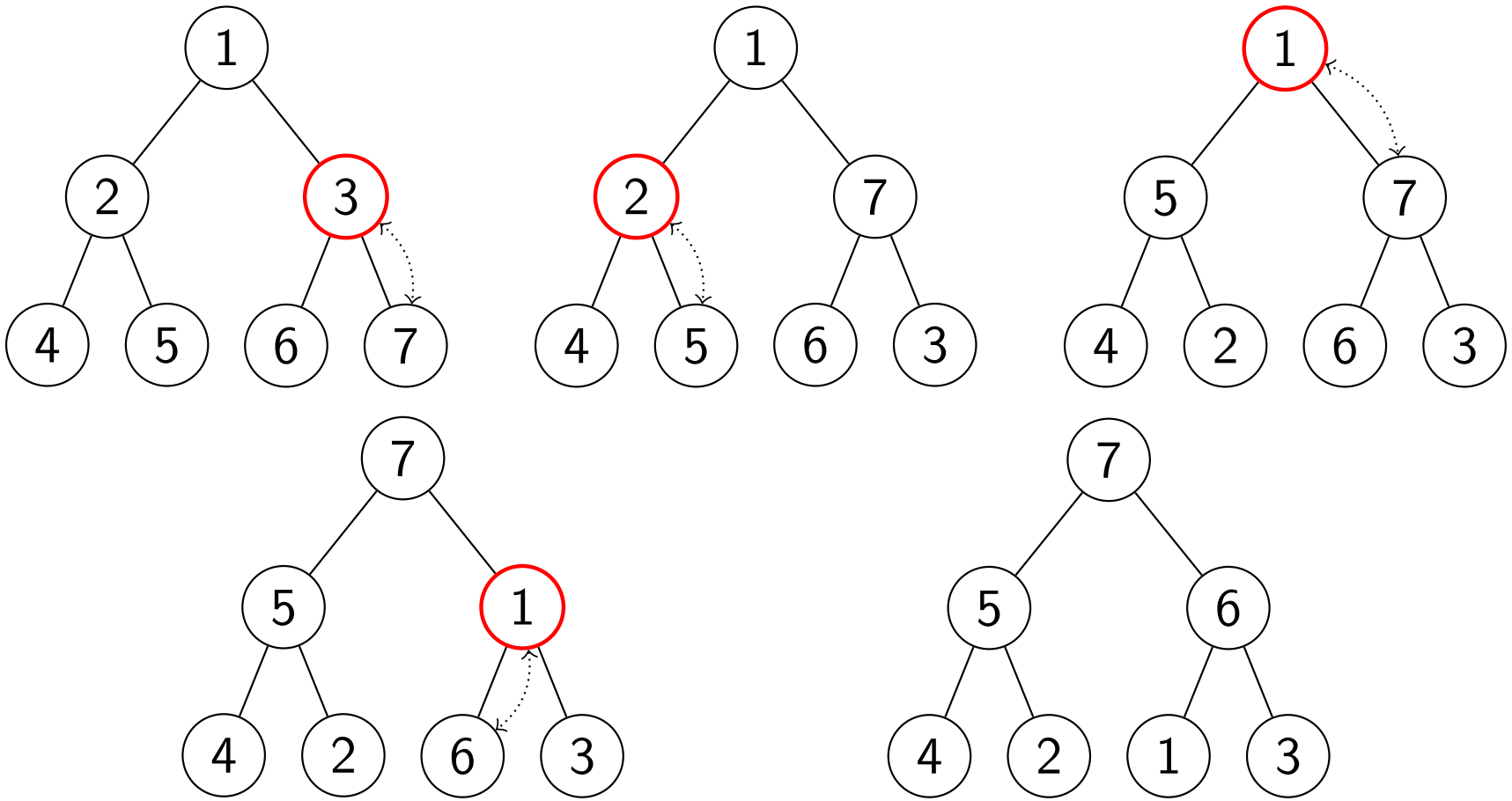
- Construir un max heap con los datos
- ir extrayendo elementos del heap y colocarlos en el hueco que queda
- repetir hasta que se vacie el heap

¿Cómo construir el heap?

- Una forma es ir insertando los elementos de uno en uno en el heap
- complejidad $O(n \log(n))$

- Otra forma es aplicando el algoritmo `sink` a todos los elementos del vector, desde el final al principio (basta con la mitad).
- tiene una complejidad $O(n)$

Ejemplo de Heapification



Complejidad: (h = nivel del árbol, $(n+1)/2^h$ = nodos por nivel)

$$\sum_{h=2}^{\log(n)} \frac{n}{2^h} T_{\text{sink}}(2^h - 1) \in \sum_{h=2}^{\log(n)} \frac{n}{2^h} O(h) \subseteq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Algoritmo de ordenación por montículo o Heapsort

Heap sort

```
1  void heapSort(int *v, size_t n) {
2    if( n < 2) return;  // do nothing
3      // Build a MAX-HEAP with the input arra
4    for (size_t i = n / 2 - 1; true; i--){
5      sink(v, n, i);
6      if (i==0) break; // size_t is unsigned type
7    }
8
9    for (size_t i=n-1; i>0; i--) {
10     swap(v[0], v[i]); // Move current root to the end.
11     sink(v, i, 0);
12   }
13 }
```

¿Complejidad temporal?

Algoritmo de ordenación por montículo o Heapsort

Heap sort

```
1  void heapSort(int *v, size_t n) {
2    if( n < 2) return;  // do nothing
3      // Build a MAX-HEAP with the input arra
4    for (size_t i = n / 2 - 1; true; i--){
5      sink(v, n, i);
6      if (i==0) break; // size_t is unsigned type
7    }
8
9    for (size_t i=n-1; i>0; i--) {
10     swap(v[0], v[i]); // Move current root to the end.
11     sink(v, i, 0);
12   }
13 }
```

¿Complejidad temporal?

$$O(n) + \sum_{i=1}^{n-1} \log(i) \in O(n \log(n))$$

Ejercicios:

¿Cierto o falso?

$$O(n^2) \subset O(2^n) \tag{1}$$
$$3n^2 + 3 \in O(n^3) \tag{2}$$
$$O(2^{\log(n)}) \subset O(n^2) \tag{3}$$
$$n + n\log(n) \in \Omega(n) \tag{4}$$
$$n + n\log(n) \in \Theta(n) \tag{5}$$
$$\Theta(n/2) = \Theta(n) \tag{6}$$
$$\Theta(n) \subseteq O(n) \tag{7}$$
$$\Theta(n) \subseteq \Theta(n^2) \tag{8}$$
$$O(n^{log(n)}) \subseteq O(2^n) \tag{9}$$

Análisis y diseño de algoritmos

3. Divide y vencerás

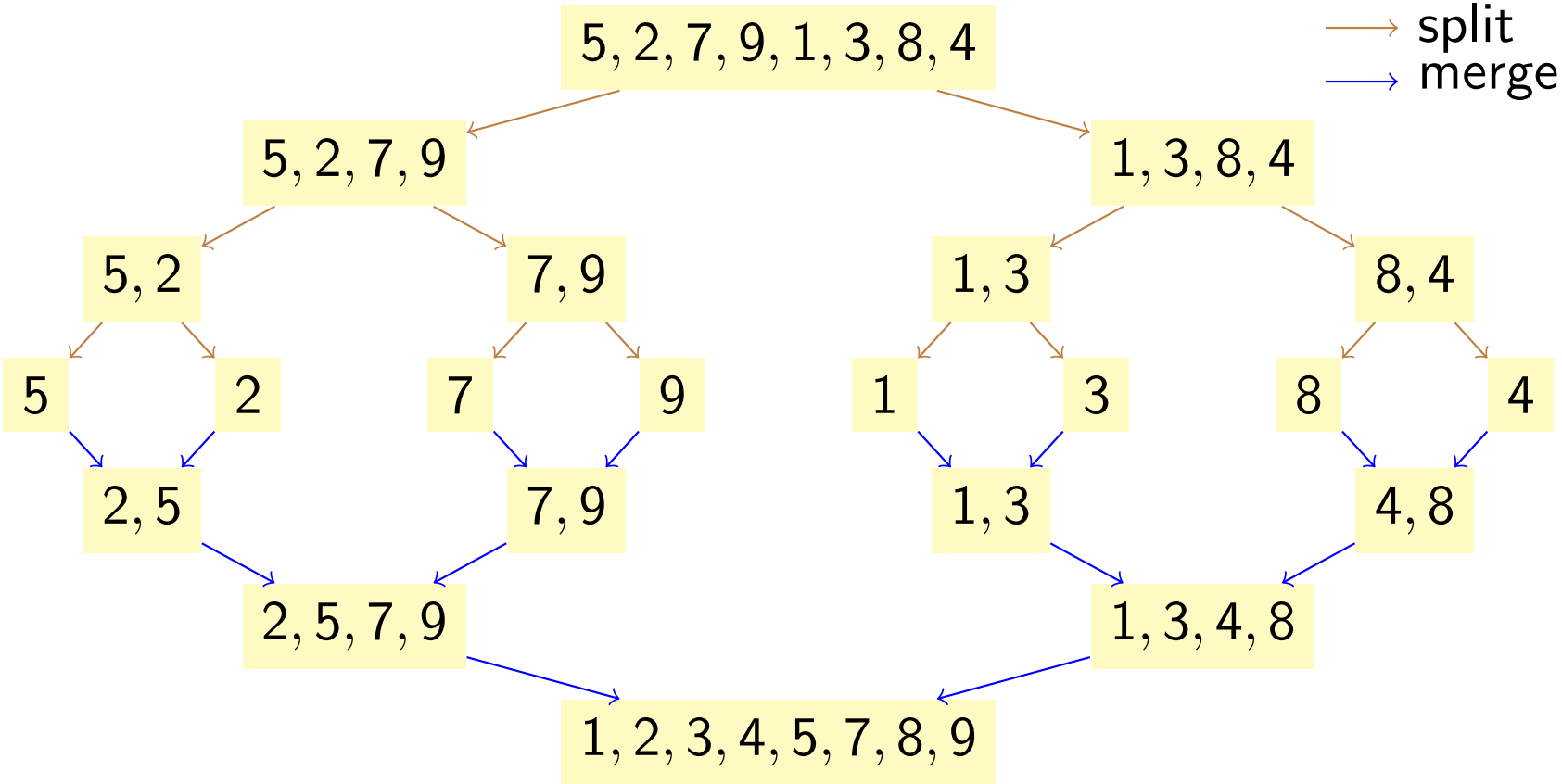
José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

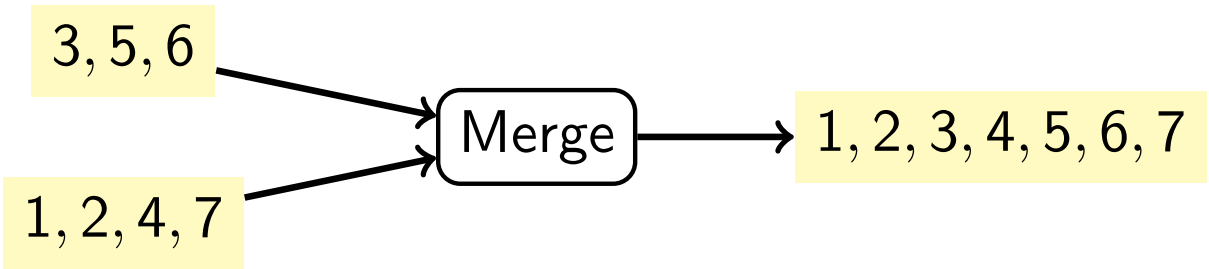
Algoritmo `Mergesort`: idea

- Ordenar de forma ascendente un vector V de n elementos.
- Solución usando el esquema "divide y vencerás":



Algoritmo `Mergesort`: función `merge`

- El algoritmo `mergeSort` utiliza la función `merge` que obtiene un vector ordenado como fusión de dos vectores también ordenados



Algoritmo *Merge*

Mergesort

```
1  void merge( vector<int> &v, size_t first, size_t middel, size_t last ) {
2      vector<int> v_merged;
3      v_merged.reserve( last - first); // to make it faster
4
5      size_t l = first, r = middel;
6      while( l != middel && r != last ) {
7          if( v[l] < v[r] ) {
8              v_merged.push_back(v[l]); ++l;
9          } else {
10             v_merged.push_back(v[r]); ++r;
11         }
12     }
13     for( ; l != middel; ++l ) v_merged.push_back(v[l]);
14     for( ; r != last; ++r) v_merged.push_back(v[r]);
15     //    for( size_t i = first; i < last; ++i ) v[i] = v_merged[i-first];
16     copy( begin(v_merged), end(v_merged), &v[first] );  // faster
17 }
```

¿Complejidad?

Algoritmo *Merge*

Mergesort

```
1  void merge( vector<int> &v, size_t first, size_t middel, size_t last ) {
2      vector<int> v_merged;
3      v_merged.reserve( last - first); // to make it faster
4
5      size_t l = first, r = middel;
6      while( l != middel && r != last ) {
7          if( v[l] < v[r] ) {
8              v_merged.push_back(v[l]); ++l;
9          } else {
10             v_merged.push_back(v[r]); ++r;
11         }
12     }
13     for( ; l != middel; ++l ) v_merged.push_back(v[l]);
14     for( ; r != last; ++r) v_merged.push_back(v[r]);
15     //    for( size_t i = first; i < last; ++i ) v[i] = v_merged[i-first];
16     copy( begin(v_merged), end(v_merged), &v[first] );  // faster
17 }
```

¿Complejidad? $\Theta(n)$ donde $n =$`last - first`

Algoritmo *Mergesort*

- Si la longitud de la lista es 0 ó 1, terminar. En otro caso
- Dividir la lista en dos sublistas de aproximadamente igual tamaño
- Ordenar cada sublista recursivamente aplicando `mergesort`
- Mezclar las dos sublistas ordenadas en una sola lista ordenada

Algoritmo *Mergesort*

Mergesort

```
1  void mergesort(
2      vector<int> &v,
3      size_t first,
4      size_t last    // past-the-end
5  ) {
6
7      if( last - first < 2 )
8          return;
9
10     size_t middel = first + ( last - first ) / 2;
11
12     mergesort( v, first, middel);
13     mergesort( v, middel, last);
14
15     merge( v, first, middel, last );
16 }
```

Algoritmo *Mergesort*: Complejidad

- Talla: n $(n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2\,T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

Algoritmo *Mergesort*: Complejidad

- Talla: n $(n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2\,T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

Algoritmo *Mergesort*: Complejidad

- Talla: n $(n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

- Hay que resolver:

$$M(n) = \begin{cases} 1 & n \leq 1 \\ n + M(\frac{n}{2}) & n > 1 \end{cases}$$

Algoritmo *Mergesort*: Complejidad

- Talla: n $(n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

- Hay que resolver:

$$M(n) = \begin{cases} 1 & n \leq 1 \\ n + M(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad espacial: $O(n)$

Técnica de divide y vencerás

- Técnica de diseño de algoritmos que consiste en:
  - Descomponer el problema en subproblemas de menor tamaño que el original
  - Resolver cada subproblema de forma individual e independiente
  - Combinar las soluciones de los subproblemas para obtener la solución del problema original
- Consideraciones:
  - No siempre un problema de talla menor es más fácil de resolver
  - La solución de los subproblemas no implica necesariamente que la solución del problema original se pueda obtener fácilmente
- Aplicable si encontramos:
  - Forma de descomponer un problema en subproblemas de talla menor
  - Forma directa de resolver problemas menores a un tamaño determinado
  - Forma de combinar las soluciones de los subproblemas que permita obtener la solución del problema original

Esquema de divide y vencerás

Esquema divide y vencerás (DC)

```
1  Solution DC( Problem p ) {
2    if( is_simple(p) )
3       return trivial(p);
4
5    list<Solution> s;
6    for( Problem q : divide(p) )
7      s.push_back( DC(q) );
8
9    return combine(s);
10 }
```

Mergesort como divide y vencerás

Particularización (instanciación) del esquema general para el caso de Mergesort:

- `is_simple`: $(\texttt{last} - \texttt{first} < 2)$
- `trivial`: retorno sin hacer nada
- `divide`: $\texttt{middle} = \texttt{first} + (\texttt{last} - \texttt{first})/2$
- `combine`: merge(...)

Quicksort

```
1  void quicksort(
2    vector<int> &v,
3    size_t first,
4    size_t last      // past-the-end
5  ) {
6
7    if( last - first < 2 ) return;
8
9    size_t p = first, l = last;
10   while(p+1 < l) {
11     if (v[p+1] < v[p]) {
12       swap( v[p+1], v[p] );
13       p++;
14     } else {
15       l--;
16       swap( v[p+1], v[l] );
17     }
18   }
19
20   quicksort(v, first, p);
21   quicksort(v, p+1, last);
22 }
```

Quicksort como divide y vencerás

Particularización (instanciación) del esquema general para el caso de quickSort:

- `is_simple`: (`last - first < 2`)
- `trivial`: retorno sin hacer nada
- `divide`: cálculo de la posición del pivote y reparto del resto
- `combine`: no es necesario

Análisis de eficiencia

- Eficiencia: costes de logarítmicos a exponenciales.
  Depende de:
  - Nº de subproblemas (h)
  - Tamaño de los subproblemas
  - Grado de intersección entre los subproblemas
- Ecuación de recurrencia:
  - $g(n)$ = tiempo del esquema para tamaño n. (sin llamadas recursivas)
  - b = Cte. de división de tamaño de problema

$$T(n) = hT\left(\frac{n}{b}\right) + g(n)$$

  - Solución general (master theorem): suponiendo la existencia de un entero k tal que: $g(n) \in \Theta(n^k)$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } h < b^k \quad \text{manda el segundo término} \\ \Theta(n^k \log_b n) & \text{si } h = b^k \\ \Theta(n^{\log_b h}) & \text{si } h > b^k \quad \text{manda el primer término} \end{cases}$$

Análisis de eficiencia (caso especial)

- **Teorema de reducción**: los mejores resultados en cuanto a coste se consiguen cuando los subproblemas son aproximadamente del mismo tamaño (y no contienen subproblemas comunes).
- **Caso especial:**
  Si se cumple la condición del teorema de reducción ($b = h = a$)

$$T(n) = aT\left(\frac{n}{a}\right) + g(n) \qquad\qquad g(n) \in \Theta(n^k)$$

$$T(n) = \begin{cases} \Theta(n^k) & k > 1 \\ \Theta(n \log n) & k = 1 \\ \Theta(n) & k < 1 \end{cases}$$

Demostración

La ecuación de recurrencia a calcular es:

$$T(n) = \begin{cases} 1 & n = 1 \\ a\,T(\frac{n}{a}) + n^k & n > 1 \end{cases} \tag{1}$$

Supongo que:

- $a \in \mathbb{N}, a > 1$ (el problema debe reducirse)
- $k \in \mathbb{R}$,
- $k \geq 0$

$$
\begin{aligned}
T(n) &\overset{1}{=} a\,T\left(\frac{n}{a}\right) + n^k \\
&\overset{2}{=} a\left[a\,T\left(\frac{n}{a^2}\right) + \left(\frac{n}{a}\right)^k\right] + n^k \\
&\overset{2}{=} a^2\,T\left(\frac{n}{a^2}\right) + a\left(\frac{n}{a}\right)^k + n^k \\
&\overset{3}{=} a^2\left[a\,T\left(\frac{n}{a^3}\right) + \left(\frac{n}{a^2}\right)^k\right] + a\left(\frac{n}{a}\right)^k + n^k \\
&\overset{3}{=} a^3\,T\left(\frac{n}{a^3}\right) + a^2\left(\frac{n}{a^2}\right)^k + a\left(\frac{n}{a}\right)^k + n^k \\
&\dots \\
&\overset{j}{=} a^j\,T\left(\frac{n}{a^j}\right) + n^k \sum_{i=0}^{j-1}\left(\frac{1}{a^{k-1}}\right)^i
\end{aligned}
$$

Pararemos cuando $\frac{n}{a^j} = 1$, o sea cuando $j = \log_a(n)$.

$$a^j\ T\left(\frac{n}{a^j}\right) + n^k \sum_{i=0}^{j-1} \left(\frac{1}{a^{k-1}}\right)^i$$

Sustituyendo $j = \log_a(n)$:

$$T(n) = n + n^k \sum_{i=0}^{\log_a(n)-1} \left(\frac{1}{a^{k-1}}\right)^i$$

Sumamos ahora la serie geométrica de razón $r = \frac{1}{a^{k-1}}$

$$\sum_{i=0}^{j-1} r^i = \frac{1-r^j}{1-r} = \frac{r^j-1}{r-1}$$

Tres casos

- $r = 1$ $(k = 1)$
  La fórmula general no vale, pero la suma vale j:

$$T(n) = n + n\log_a(n) \in \Theta(n\log(n))$$

- $r < 1$ ($k > 1$)
  Para valores grandes de j, $r^j \lll 1$, el sumatorio será $\sim \frac{1}{1-r}$

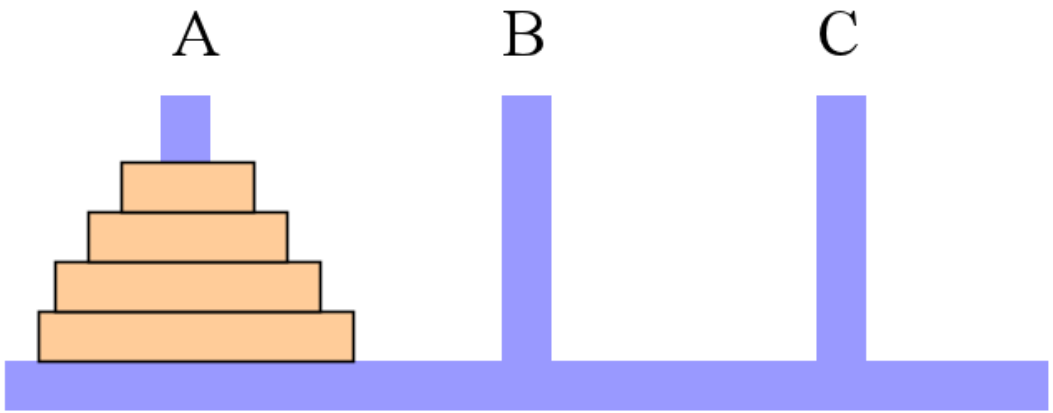
$$T(n) = n + n^k \frac{1}{1-r} \in \Theta(n^k)$$

- $r > 1$ $(k < 1)$
  Para valores grandes de j, $r^j \ggg 1$, el sumatorio será $\sim \frac{r^j}{r-1}$,
  Y como: $r^j = \left(\frac{1}{a^{k-1}}\right)^{\log_a(n)} = n^{\log_a\left(\frac{1}{a^{k-1}}\right)} = n^{\log_a a^{-(k-1)}} = n^{1-k}$

$$T(n) = n + n^k \frac{r^j}{r-1} = n + n^k \frac{n^{1-k}}{r-1} \in \Theta(n)$$

Las torres de Hanoi

- Colocar los discos de la torre A en la C empleando como ayuda la torre B
- Los discos han de moverse uno a uno y sin colocar nunca un disco sobre otro más pequeño



- ¿Cómo se afrontaría el problema?
- ¿Cuál sería la complejidad del algoritmo resultante?

Las torres de Hanoi: solución

- hanoi$(n, A \xrightarrow{B} C)$ es la solución del problema: mover los n discos superiores del pivote A al pivote C.
- Supongamos que sabemos mover $n-1$ discos: sabemos cómo resolver hanoi$(n-1, X \xrightarrow{Y} Z)$.
- También sabemos como mover 1 disco del pivote X al Z: hanoi$(1, X \xrightarrow{Y} Z)$, que es el caso trivial. Lo llamaremos mover$(X \to Z)$.
- Resolver hanoi$(n, A \xrightarrow{B} C)$ equivale a ejecutar:
  - hanoi$(n-1, A \xrightarrow{C} B)$
  - mover$(A \to C)$
  - hanoi$(n-1, B \xrightarrow{A} C)$

Las torres de Hanoi: Complejidad (1)

Nótese que aquí la talla de los subproblemas no es $\frac{n}{a}$ sino $n-1$:

- No se pueden aplicar las fórmulas generales de las transparencias anteriores
- El problema tiene una complejidad intrínseca peor que las descritas en las transparencias anteriores

Las torres de Hanoi: Complejidad (2)

- Ecuación de recurrencia para el coste exacto (asumiendo coste 1 para todas las operaciones de 1 disco):

$$T(n) = \begin{cases} 1 & n = 1 \\ 1 + 2T(n-1) & n > 1 \end{cases}$$

- Solución:

$$\begin{aligned} T(n) &\overset{1}{=} 1 + 2T(n-1) \\ &\overset{2}{=} 1 + 2(1 + 2T(n-2)) = 1 + 2 + 2^2 T(n-2) \\ &\overset{3}{=} 1 + 2 + 2^2(1 + 2T(n-3)) = 2^0 + 2^1 + 2^2 + 2^3 T(n-3) = \ldots \\ &\overset{k}{=} \sum_{i=0}^{k-1} 2^i + 2^k T(n-k) = 2^k - 1 + 2^k T(n-k) \end{aligned}$$

- Paramos cuando $n - k = 1$, o sea, en el paso $k = n - 1$,

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

Ejercicios

1. Selección del k-ésimo mínimo
   - Dado un vector A de n números enteros diferentes, diseñar un algoritmo que encuentre el k-ésimo valor mínimo.
2. Búsqueda binaria o dicotómica
   - Dado un vector X de n elementos ordenado de forma ascendente y un elemento e, diseñar un algoritmo que devuelva la posición del elemento e en el vector X.
3. Calculo recursivo de la potencia enésima.

Selección del k-ésimo mínimo

- Recoloca los elementos de un vector de forma que el elemento en la `nth` posición sea el elemento que estaría en la `nth` posición si ordenásemos el vector
- Dados los iteradores al principio, nth elemento y final de un rango, se puede calcular con la función de las STL:

```
1 void nth_element (
2   RandomIt first,
3   RandomIt nth,
4   RandomIt last
5 );
```

```
1 void nth_element (
2   RandomIt first,
3   RandomIt nth,
4   RandomIt last,
5   Compare comp
6 );
```

- El algoritmo `quickselect` permite hacer lo mismo y está basado en el `quicksort`

Quicksort algoritm

```
1  void quicksort(
2    vector<int> &v,
3
4    size_t first,
5    size_t last  // past-the-end index
6  ) {
7    if( last - first < 2 ) return;
8
9    size_t p = first, l = last;
10
11   while(p+1 < l) {
12     if (v[p+1] < v[p]) {
13       swap( v[p+1], v[p] ); p++;
14     } else {
15       l--; swap( v[p+1], v[l] );
16     }
17   }
18
19
20
21   quicksort(v, first, p);
22
23   quicksort(v, p+1, last);
24 }
```

Algoritmo Quickselect

```
1  void quickselect(
2    vector<int> &v,
3    size_t first,
4    size_t nth,
5    size_t last  // past-the-end index
6  ) {
7    if( last - first < 2 ) return;
8
9    size_t p = first, l = last;
10
11   while( p+1 < l ) {
12     if( v[p + 1] < v[p] ) {
13       swap( v[p+1], v[p] ); p++;
14     } else {
15       l--; swap( v[p+1], v[l] );
16     }
17   }
18
19   if( nth == p ) return;
20   if( nth < p )
21     quickselect( v, first, nth, p );
22   else
23     quickselect( v, p+1, nth, last);
24 }
```

Complejidad Quickselect

- Tamaño del problema: $n = \texttt{last} - \texttt{first}$
- Mejor caso
  - cuando $\texttt{nth} = \texttt{p}$
  - complejidad: $O(n)$
- Peor caso
  - cuando $\texttt{nth} \neq \texttt{p}$ y el pivote siempre queda al principio o el final
  - Complejidad

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + T(n-1) & n > 1 \end{cases} \in O(n^2)$$

- Otro caso interesante
  - cuando $\texttt{nth} \neq \texttt{p}$ y el pivote siempre queda en el centro
  - Complejidad

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + T(\frac{n}{2}) & n > 1 \end{cases} \in O(n)$$

Búsqueda binaria o dicotómica

Búsqueda binaria

```
1  size_t lower_bound( const auto &v, auto val, size_t first, size_t count) {
2    if( count == 0 ) return first;
3    size_t step = count/2;
4    size_t med = first + step;
5    if( v[med] < val )
6      return lower_bound( v, val, med+1, count - (step + 1));
7    else
8      return lower_bound( v, val, first, step );
9  }
10
11 size_t binary_search( const auto &v, auto val ) {
12   size_t first = lower_bound( v, val, 0, v.size() );
13   if( first < v.size() && v[first] == val )
14     return first;
15   else
16     return NOT_FOUND;
17 }
```

- ¿Reconocéis en el algoritmo los componentes del *divide y vencerás*?

Búsqueda binaria

- Esta solución se puede ver como un *divide y vencerás* en el que
  - La operación `divide` viene representada por `med = first + step;`
  - El problema `is_simple` corresponde a cuando `count == 0`
  - Sólo se resuelve uno de los dos subproblemas
    - *divide y vencerás* $\rightarrow$ *reduce y vencerás*
  - No es necesaria `combine`

Búsqueda binaria: complejidad

- Ecuación de recurrencia para el caso peor:

$$T(n) = \begin{cases} 1 & n = 1 \\ 1 + T(\frac{n}{2}) & n > 1 \end{cases}$$

(agrupamos $n = 0$ en $n = 1$ porque no se produce división).

- Solución:

$$\begin{aligned} T(n) &\overset{1}{=} 1 + T(\frac{n}{2}) \\ &\overset{2}{=} 1 + 1 + T(\frac{n}{2^2}) = 2 + T(\frac{n}{2^2}) \\ &\dots \\ &\overset{k}{=} k + T(\frac{n}{2^k}) \end{aligned}$$

- Paramos cuando $\frac{n}{2^k} = 1$, o sea en el paso $k = \log(n)$,

$$T(n) \in O(\log(n))$$

Cálculo de la potencia enésima

Si asumimos que multiplicar dos elementos de un determinado tipo tiene un coste constante, es posible calcular la enésima potencia x^n de un elemento x de ese tipo en tiempo sublineal usando la siguiente recursión:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{\frac{n}{2}} x^{\frac{n}{2}} & n \text{ es par} \\ x^{\frac{n-1}{2}} x^{\frac{n-1}{2}} x & n \text{ es impar} \end{cases}$$

Escribid un algoritmo para calcular eficientemente x^n.

- ¿Se puede evitar repetir operaciones?
- ¿Cuál es el coste asintótico del algoritmo resultante?

Análisis y diseño de algoritmos

4. Programación Dinámica

José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

11 de marzo de 2024

Índice

Índice

El problema de la mochila (*Knapsack problem*)

- Sean n objetos con valores ($v_i \in \mathbb{R}$) y pesos ($w_i \in \mathbb{R}^{>0}$) conocidos
- Sea una mochila con capacidad máxima de carga W
- ¿Cuál es el valor máximo que puede transportar la mochila sin sobrepasar su capacidad?

- Un caso particular: **La mochila 0/1 con pesos discretos**
  - Los objetos no se pueden fraccionar (mochila 0/1 o mochila discreta)
    - La variante más difícil
  - **Simplificación:** Los pesos son cantidades discretas o discretizables
    - Se utilizarán para indexar una tabla
    - Una versión menos general que suaviza su dificultad

Formalización

- Es un problema de optimización:
  - Secuencia de decisiones: $(x_1, x_2 \dots x_n) : \ x_i \in \{0, 1\}, \ 1 \leq i \leq n$
    - En x_i se almacena la decisión sobre el objeto i
    - Si x_i es escogido $x_i = 1$, en caso contrario $x_i = 0$
  - Una secuencia óptima de decisiones es la que maximiza $\sum_{i=1}^{n} x_i v_i$

    sujeto a las restricciones:
    - $\sum_{i=1}^{n} x_i w_i \leq W$
    - $\forall i : 1 \leq i \leq n, \ x_i \in \{0, 1\}$
- Representamos mediante `knapsack(`k`,`C`)` al problema de la mochila con los objetos 1 hasta k y capacidad C
  - El problema inicial es, por tanto, `knapsack(n,W)`

Subestructura óptima o Pricipio de Optimalidad

- Sea $(x_1, x_2 \ldots x_n)$ una secuencia óptima de decisiones para el problema $\texttt{knapsack}(n, W)$
  - Si $x_n = 0$ entonces $(x_1 \ldots x_{n-1})$ es una secuencia óptima para el subproblema $\texttt{knapsack}(n-1, W)$
  - Si $x_n = 1$ entonces $(x_1 \ldots x_{n-1})$ es una secuencia óptima para el subproblema $\texttt{knapsack}(n-1, W - w_n)$

Demostración:

Supongamos que $(x'_1 \ldots x'_{n-1})$ es una solución mejor para el subproblema útil.
Entonces la secuencia $(x'_1, x'_2 \ldots x'_{n-1}, x_n)$ será mejor que $(x_1, x_2 \ldots x_n)$ para el problema original, y esto contradice la suposición inicial de que era la óptima.

$\Rightarrow$ La solución al problema presenta una subestructura óptima

Aproximación matemática

- Se toman decisiones en orden descendente: x_n, x_{n-1}, ... x_1
- Ante la decisión x_i hay dos alternativas:
  - Rechazar el objeto i: $x_i = 0$
    - No hay ganancia adicional pero la capacidad de la mochila no se reduce
  - Seleccionar el objeto i: $x_i = 1$
    - La ganancia adicional es v_i, a costa de reducir la capacidad en w_i
- Se selecciona la alternativa que mayor ganancia global resulte

Solución

$\{ W \geq 0,\ n > 0 \}$

$$\text{knapsack}(0, W) = 0$$

$$\text{knapsack}(n, W) = \max \begin{cases} \text{knapsack}(n-1, W) \\ \text{knapsack}(n-1, W - w_n) + v_n \quad \text{if } W \geq w_n \end{cases}$$

Una solución recursiva

Solución recursiva (ineficiente)

```
1  #include <limits>
2
3  double knapsack(
4      const vector<double> &v,     // values
5      const vector<unsigned> &w,   // weights
6      int n,                       // number of objects
7      unsigned W                  // knapsack weight limit
8  ) {
9      if( n == 0 )                                          // base case
10         return 0;
11
12     double S1 = knapsack( v, w, n-1, W );       // try not to put it on
13
14     double S2 = numeric_limits<double>::lowest();
15     if( w[n-1] <= W )                        // does it fits in the knapsack?
16         S2 = v[n-1] + knapsack( v, w, n-1, W-w[n-1] ); // try to put it on
17
18     return max( S1, S2 );                               // choose the best
19 }
```

Versión recursiva: Complejidad temporal

- En el mejor de los casos:
  ningún objeto cabe en la mochila, se tiene $T(n) \in \Omega(n)$
- En el peor de lo casos:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{en otro caso} \end{cases}$$

El témino general queda como:

$$T(n) \stackrel{k}{=} 2^k - 1 + 2^k T(n-k)$$

Que terminará cuando $n - k = 0$, o sea:

$$T(n) = 2^n - 1 + 2^n \in O(2^n)$$

Versión recursiva: Complejidad temporal

- En el mejor de los casos:
  ningún objeto cabe en la mochila, se tiene $T(n) \in \Omega(n)$
- En el peor de lo casos:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{en otro caso} \end{cases}$$

El témino general queda como:

$$T(n) \overset{k}{=} 2^k - 1 + 2^k T(n-k)$$

Que terminará cuando $n - k = 0$, o sea:

$$T(n) = 2^n - 1 + 2^n \in O(2^n)$$

Pero ...

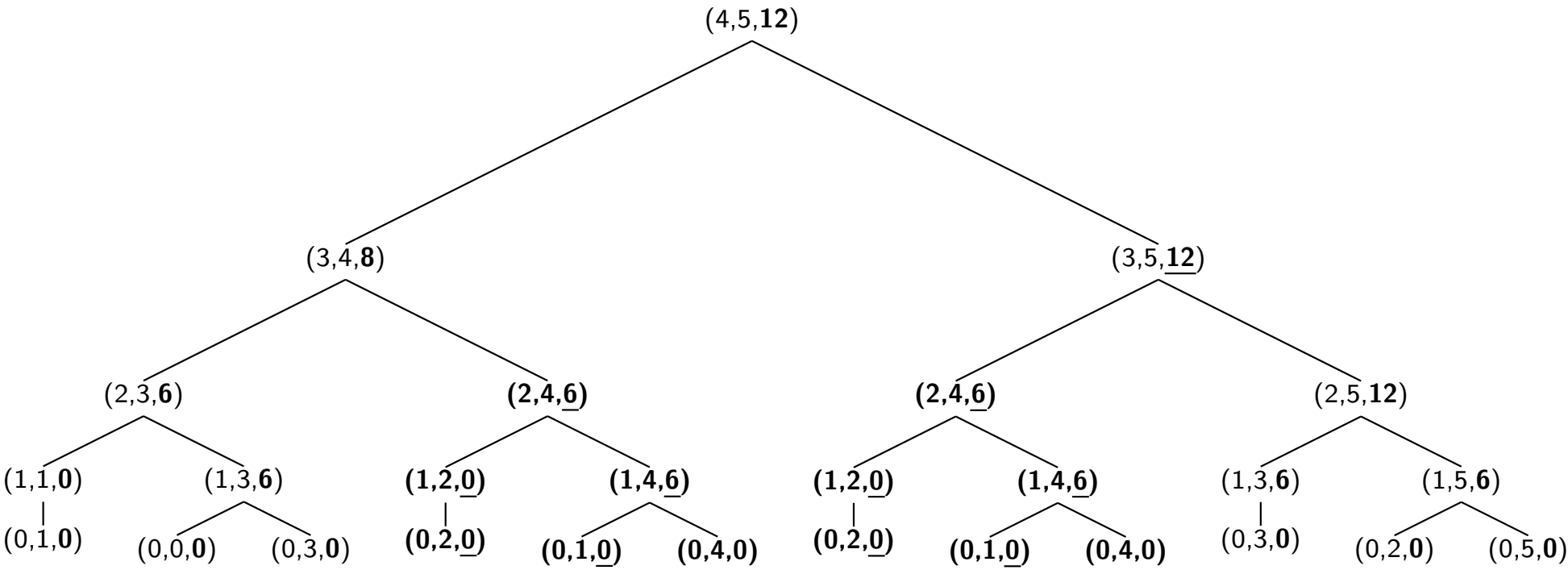
¡solo pueden haber nW problemas distintos!

Version recursiva: Subproblemas repetidos

- Ejemplo:

$$\begin{cases} n = 4,\ W = 5 \\ w = (3, 2, 1, 1) \\ v = (6, 6, 2, 1) \end{cases}$$

Nodos: $(i, W, \texttt{knapsack}(i, W))$; izquierda, $x_i = 1$; derecha, $x_i = 0$.



¿Cómo evitar repeticiones?

- Antes de resolver un nuevo problema
  - Si no ha sido resuelto $\Rightarrow$ resolverlo y anotarlo
  - Si sí ha sido resuelto $\Rightarrow$ usar la solución anotada

Necesitamos:

- Encontrar una forma de indexar los subproblemas $[(n, W)]$
- Identificar un tipo de dato aceptable para la solución [double]
- Definir una estructura donde almacenar los resultados
  - `vector<vector<double>> M`
- Encontrar un centinela. Indica cuando el problema no ha sido resuelto
  - suele ser del mismo tipo que las soluciones, pero imposible $[-1.0]$
- Modificar el programa para:
  - inicializar la estructura al centinela
    - `vector<vector<double>> M( n+1, vector<double>(W+1,-1.0) )`
  - al comenzar el programa comprobar si antes se ha resuelto el problema
    - si ya se ha resulto previamente devolver la solución almacenada
  - al terminar el programa almacenar la solución

Una solución recursiva con almacén (Memoización)

```
1  const double SENTINEL = -1.0;
2  double knapsack(
3      vector< vector< double >> &M,                                    // Storage
4      const vector<double> &v, const vector<unsigned> &w,     // values & weights
5      int n, unsigned W                          // num. of objects & Knapsack limit
6  ) {
7      if( M[n][W] != SENTINEL ) return M[n][W];              // if it is known ...
8      if( n == 0 ) return M[n][W] = 0.0;                     // base case
9
10     double S1 = knapsack( M, v, w, n-1, W );
11     double S2 = numeric_limits<double>::lowest();
12     if (w[n-1] <= W) S2 = v[n-1] + knapsack( M, v, w, n-1, W - w[n-1]);
13     return M[n][W] = max( S1, S2 );            // store and return the solution
14 }
15 //-------------------------------------------------------------
16 double knapsack(
17     const vector<double> &v, const vector<unsigned> &w, int n, unsigned W
18 ) {
19     vector< vector< double >> M( n+1, vector<double>( W+1, SENTINEL)); // init.
20     return knapsack( M, v, w, n, W );
21 }
```

Programación dinámica recursiva con almacén

- Ejemplo: Sean $n = 5$ objetos con pesos (w_i) y valores (v_i) indicados en la tabla. Sea $W = 11$ el peso máximo de la mochila.

M[6][12]	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0		0		0	
$w_1 = 2, v_1 = 1$	0		1	1	1	1	1			1		1
$w_2 = 2, v_2 = 7$	0				8	8	8					8
$w_3 = 5, v_3 = 18$					8	18						26
$w_4 = 6, v_4 = 22$					8							40
$w_5 = 7, v_5 = 28$												40

- El 60 % de las celdas no se han utilizado por lo tanto:

  El subproblema asociado no ha sido resuelto
  ¡Ahorro computacional!

40 Solución al problema
Contorno o perfil
Celdas sin uso

$$M[5][11] = \max\left(M[4][11], M[4][11 - w_5] + v_5\right) = \max(40, 36)$$ 5 no se toma

$$M[4][11] = \max\left(M[3][11], M[3][11 - w_4] + v_4\right) = \max(26, 40)$$ 4 sí se toma

$$M[3][5] = \max\left(M[2][5], M[2][5 - w_3] + v_3\right) = \max(8, 18)$$ 3 sí se toma

$$M[2][0] = M[1][0] = 0$$ 1 y 2 no se toman

Paso a iterativa

- Encontrar una forma de recorrer los problemas de forma que siempre que se haga referencia a subproblemas estos ya hayan sido resueltos
  - En nuestro caso rellenar por filas
- Cambiar las llamadas recursivas por consultas al almacén
- Cambiar los `return` por `continue`
- No hace falta ni el centinela ni iniciaizar el almacén
- No olvidar actualizar el almacén cada vez que se tienen una solución

Una solución iterativa

```
1  double knapsack(
2      const vector<double> &v,       // values
3      const vector<unsigned> &w,     // weights
4      int last_n, unsigned last_W    // number of objects & knapsack limit weight
5  ) {
6      vector< vector< double >> M( last_n+1, vector<double>(last_W+1));
7
8      for( int n = 0; n <= last_n; n++ )
9          for( unsigned W = 0; W <= last_W; W++ ) {
10
11             if( n == 0 ) {                                  // no objects
12               M[n][W] = 0;
13               continue;
14             }
15
16             double S1 = M[n-1][W];
17             double S2 = numeric_limits<double>::lowest();
18             if( W >= w[n-1] )                              // if it fits ...
19                 S2 = v[n-1] + M[n-1][W-w[n-1]];             // try to put it
20             M[n][W] = max( S1, S2 );                     // store the best
21         }
22
23     return M[last_n][last_W];
24 }
```

Solución iterativa

Otra solución iterativa (mejor)

```
1  double knapsack(
2       const vector<double> &v,  // values
3      const vector<unsigned> &w, // weights
4      int last_n,                // assessed object
5      unsigned last_W            // Knapsack limit weight
6  ) {
7      vector< vector< double >> M( last_n+1, vector<double>(last_W+1));
8
9      for( unsigned W = 0; W <= last_W; W++ ) M[0][W] = 0;        // no objects
10
11     for( int n = 1; n <= last_n; n++ )
12         for( unsigned W = 1; W <= last_W; W++ ) {
13             double S1 = M[n-1][W];
14             double S2 = numeric_limits<double>::lowest();
15             if( W >= w[n-1] )                    // if it fits ...
16                 S2 = v[n-1] + M[n-1][W-w[n-1]]; // try to put it
17             M[n][W] = max( S1, S2 );       // store the best
18         }
19
20     return M[last_n][last_W];
21 }
```

Almacén de resultados parciales

- Ejemplo: Sean $n = 5$ objetos con pesos (w_i) y valores (v_i) indicados en la tabla. Sea $W = 11$ el peso máximo de la mochila.

M[6][12]	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 1$	0	0	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 7$	0	0	7	7	8	8	8	8	8	8	8	8
$w_3 = 5, v_3 = 18$	0	0	7	7	8	18	18	25	25	26	26	26
$w_4 = 6, v_4 = 22$	0	0	7	7	8	18	22	25	29	29	30	40
$w_5 = 7, v_5 = 28$	0	0	7	7	8	18	22	28	29	35	35	40

$M[i][j] \equiv$ Ganancia máxima con los i primeros objetos y con una carga máxima j. Por tanto, solución en $M[5][11]$

40 Solución al problema

Contorno o perfil

$$M[i][j] = \max(\underbrace{M[i-1][j]}_{\text{rechazar } i}, \underbrace{M[i-1][j-w_i] + v_i}_{\text{seleccionar } i})$$

$$M[5][11] = \max\left(M[4][11], M[4][11-w_5] + v_5\right) = \max(40, 36).$$ 5 no se toma

$$M[4][11] = \max\left(M[3][11], M[3][11-w_4] + v_4\right) = \max(26, 40).$$ 4 sí se toma

$$M[3][5] = \max\left(M[2][5], M[2][5-w_3] + v_3\right) = \max(8, 18).$$ 3 sí se toma

$$M[2][0] = M[1][0] = 0.$$ 1 y 2 no se toman

Iterativo: Complejidad temporal y espacial

- Complejidad temporal

$$T(n, W) = 1 + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} \sum_{j=1}^{W} 1 = 1 + n + nW$$

  Por tanto,

$$T(n, W) \in \Theta(nW)$$

- Complejidad espacial

$$T_S(n, W) \in \Theta(nW)$$

  - la complejidad espacial es mejorable ...

Solución iterativa

Solución iterativa economizando memoria

```
1  double knapsack(
2      const vector<double> &v, const vector<unsigned> &w, // data vectors
3      int last_n, unsigned last_W     // num. objects & Knapsack limit weight
4  ) {
5      vector<double> v0(last_W+1);    // M[0] & M[n-1] & M[last_n]
6      vector<double> v1(last_W+1);    // M[n]
7
8      for( unsigned W = 0; W <= last_W; W++ ) v0[W] = 0;        // no objects
9
10     for( int n = 1; n <= last_n; n++ ) {
11         for( unsigned W = 1; W <= last_W; W++ ) {
12             double S1 = v0[W];
13             double S2 = numeric_limits<double>::lowest();
14             if( W >= w[n-1] )                  // if it fits ...
15                 S2 = v[n-1] + v0[W-w[n-1]]; // try to put it
16             v1[W] = max( S1, S2 );       // store the best
17         }
18         swap(v0,v1);
19     }
20     return v0[last_W];
21 }
```

¿Cómo obtener las decisiones tomadas

Dos formas:

- Marcando los problemas a partir de los cuales hemos obtenido el máximo
  - posteriormente se siguen esas marcas para obtener las decisiones óptimas
- Directamente del almacén
  - se vuelve a resolver el problema pero al conocer la solución de los subproblemas solo vamos por el camino de las decisiones óptimas

Una solución iterativa con extracción de la selección

```
1  double knapsack(                          // in trace we store the taken decision
2      const vector<double> &v, const vector<unsigned> &w,   // values & weights
3      int last_n, unsigned last_W,          // assessed object  & Knapsack limit
4      vector<vector<bool>> &trace           // trace (true->store, false->don't)
5  ) {
6      vector< vector< double >> M( last_n+1, vector<double>(last_W+1));
7      trace = vector<vector<bool>>( last_n+1, vector<bool>(last_W+1));
8
9      for( unsigned W = 0; W <= last_W; W++ ) {
10         M[0][W] = 0;                  // no objects
11         trace[0][W] = false;          // I don't take it
12     }
13
14     for( int n = 1; n <= last_n; n++ )
15         for( unsigned W = 1; W <= last_W; W++ ) {
16             double S1 = M[n-1][W];
17             double S2 = numeric_limits<double>::lowest();
18             if( W >= w[n-1] )                      // if it fits ...
19                 S2 = v[n-1] + M[n-1][W-w[n-1]];      // try to put it
20             M[n][W] = max( S1, S2 );              // store the best
21             trace[n][W] = S2 > S1;                // if true I take it
22         }
23     return M[last_n][last_W];
24 }
```

Extracción de la selección

Extracción de la selección

```
1  void parse(
2      const vector<unsigned> &w,              // weights
3      const vector<vector<bool>> &trace,      // true -> take it, false -> don't
4      vector<bool> &sol                       // true -> take it, false -> don't
5  ) {
6      unsigned last_n = trace.size()-1;
7      int W = trace[0].size()-1;
8
9      for( int n = last_n; n > 0; n-- ) {
10         sol[n-1] = trace[n][W];
11         if( sol[n-1] )
12             W -= w[n-1];
13     }
14 }
```

Extracción de la selección

Extracción de la selección (directamente del almacén)

```
1  void parse(
2      const vector<vector<double>> &M,
3      const vector<double> &v, const vector<unsigned> &w,    // values & weights
4      unsigned W,                                 // num. of objects & Knapsack limit
5      vector<bool> &sol                           // true -> take it, false -> don't
6  ){
7      int n = v.size();
8
9      for( int o = n-1; o >= 0; o-- ) {
10
11         double S1 = M[o][W];
12
13         double S2 = numeric_limits<double>::lowest();
14         if (W >= w[o] )
15             S2 = v[o] + M[o][W-w[o]];
16
17         sol[o] = S2 > S1;
18         if( sol[o] )
19             W -= w[o];
20     }
21 }
```

Conclusiones

- La complejidad temporal de la solución obtenida mediante programación dinámica está en $\Theta(nW)$
  - Un recorrido descendente a través de la tabla permite obtener también, en tiempo $\Theta(n)$, la secuencia óptima de decisiones tomadas.
- Si W es muy grande entonces las solución obtenida mediante programación dinámica no es buena
- Si los pesos w_i o la capacidad W pertenecen a dominios continuos (p.e. los reales) entonces esta solución no sirve
- La complejidad espacial de la solución obtenida se puede reducir hasta $\Theta(W)$
- En este problema, la solución PD-recursiva puede ser más eficiente que la iterativa
  - Al menos, la versión que no realiza cálculos innecesarios es más fácil de obtener en recursivo

Índice

Corte de tubos

- Una empresa compra tubos de longitud n y los corta en tubos más cortos, que luego vende
- El corte le sale gratis
- El precio de venta de un tubo de longitud i ($i = 1, 2, \ldots, n$) es p_i
  Por ejemplo:

longitud i	1	2	3	4	5	6	7	8	9	10
precio p_i	1	5	8	9	10	17	17	20	24	30

- ¿Cual es la forma óptima de cortar un tubo de longitud n para maximizar el precio total?
- Probar todas las formas de cortar es prohibitivo (¡hay 2^{n-1}!)

Corte de tubos

- Buscamos una descomposición

$$n = i_1 + i_2 + \ldots + i_k$$

  por la que se obtenga el precio máximo
- El precio es

$$r_n = p_{i_1} + p_{i_2} + \ldots + p_{i_k}$$

- Una forma de resolver el problema recursivamente es:
  - Cortar el tubo de longitud n de las n formas posibles,
  - y buscar el corte que maximiza la suma del precio del trozo cortado p_i y del resto r_{n-i},
  - suponiendo que el resto del tubo se ha cortado de forma óptima:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}); \qquad r_0 = 0$$

Corte de tubos

Solución recursiva (ganancia máxima)

```
1  int tube_cut(
2      const vector<int> &p,        // tube length prices
3      const int l                  // assessed length
4  ) {
5
6      if( l == 0 )
7          return 0;
8
9      int q = numeric_limits<int>::lowest();  // q ~ -∞
10     for( int i = 1; i <= l; i++ )
11         q = max( q, p[i] + tube_cut( p, l-i ));
12
13     return q;
14 }
```

- Es ineficiente

Complejidad de la solución recursiva

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + \sum_{j=0}^{n-1} T(j) & \text{en otro caso} \end{cases}$$

Como:

$$T(n-1) = n - 1 + \sum_{j=0}^{n-2} T(j) \Rightarrow 2T(n-1) = n - 1 + \sum_{j=0}^{n-1} T(j)$$

y teniendo en cuenta que $T(n) = n + \sum_{j=0}^{n-1} T(j)$ llegamos a

$$T(n) = 1 + 2T(n-1)$$

Por tanto:

$$T(n) = 2^n - 1 + 2^n \in O(2^n)$$

Corte de tubos

Solución recursiva (corte óptimo)

```
1  int trace_tube_cut( const vector<int> &p, const int n, vector<int> &trace ) {
2      if( n == 0 ) {    // trace stores for each length which is the optimal cut
3          trace[0] = 0;
4          return 0;
5      }
6      int q = numeric_limits<int>::lowest();
7      for( int i = 1; i <= n; i++ ) {
8          int aux = p[i] + trace_tube_cut( p, n-i, trace);
9          if( aux > q ) {    // Maximum gain
10             q = aux;
11             trace[n] = i;
12         }
13     }
14     return q;
15 }
16 //-------------------------------------------------------------------
17 vector<int> trace_tube_cut( const vector<int> &p, const int n ) {
18     vector<int> trace(n+1);
19     trace_tube_cut(p, n, trace);
20     return trace;
21 }
```

Corte de tubos

Extrayendo los cortes óptimos

```
1  vector<int> parse(
2      const vector<int> &trace
3  ) {
4      vector<int> sol(trace.size(),0);     // How many cuts for each size
5
6      int l = trace.size()-1;
7      while( l != 0 ) {
8          sol[trace[l]]++;                 //where to cut
9          l -= trace[l];                 // the rest
10     }
11     return sol;
12 }
13 //-------------------------------------------------------------
14 ...
15             vector<int> trace = trace_tube_cut( price, n );
16             vector<int> sol = parse(trace);
17             for( unsigned i = 0; i < sol.size(); i++)
18                 if( sol[i] != 0 )
19                   cout << sol[i] << " cuts of length " << i << endl;
20 ...
```

Solución recursiva con almacén (Memoización)

Corte de tubos. Ganancia máxima

```
1  const int SENTINEL = -1;
2
3  int tube_cut(
4      vector<int> &M,              // Sub-problem Storage
5      const vector<int> &p, int l
6  ) {
7      if( M[l] != SENTINEL ) return M[l];// is known?
8
9      if( l == 0 ) return M[0] = 0;
10
11     int q = numeric_limits<int>::lowest();
12     for( int i = 1; i <= l; i++ )
13         q = max( q, p[i] + tube_cut( M, p, l-i));
14
15     return M[l] = q;      // store solution & return
16 }
17
18 int tube_cut( const vector<int> &p, int l ) {
19     vector<int> M(l+1,SENTINEL); //  initialization
20     return tube_cut( M, p, l );
21 }
```

- Complejidad espacial: $O(n)$
- Complejidad temporal: $O(n^2)$

Solución Recursiva con almacén (Memoización)

Corte de tubos. Corte óptimo

```
1  vector<int> parse( const vector<int> &p, const vector<int> &M ) {
2
3      vector<int> sol(M.size(), 0);
4
5      int l = M.size() - 1;
6      while( l != 0 ) {
7          int max_i;
8          int q = numeric_limits<int>::lowest();
9          for( int i = 1; i <= l; i++ ) {
10             int v = p[i] + M[l-i];
11             if( v > q ) {
12                 q = v;
13                 max_i = i;
14             }
15         }
16         sol[max_i]++;
17         l -= max_i;
18     }
19     return sol;
20 }
```

Solución iterativa (directa)

Corte de tubos

```
1  int tube_cut(
2      const vector<int> &p,  // tube length prices
3      int n                  // assessed length
4  ) {
5      vector<int> M(n+1);    // Sub-problem storage
6
7      for( int l = 0; l <= n; l++ ) {
8
9          if( l == 0 ) {             // base case
10             M[0] = 0;
11             continue;
12         }
13
14         int q = numeric_limits<int>::lowest();
15         for( int i = 1; i <= l; i++ )
16             q = max( q, p[i] + M[l-i]);
17         M[l] = q;                  // Store solution
18     }
19     return M[n];
20 }
```

- Complejidad espacial: $O(n)$
- Complejidad temporal: $O(n^2)$

Solución iterativa (mejor).

Corte de tubos

```
1  int tube_cut(
2      const vector<int> &p,  // tube length prices
3      int n                  // assessed length
4  ) {
5      vector<int> M(n+1);    // Sub-problem storage
6
7      M[0] = 0;                          // Base case
8      for( int l = 1; l <= n; l++ ) {
9          int q = numeric_limits<int>::lowest();
10         for( int i = 1; i <= l; i++ )
11             q = max( q, p[i] + M[l-i]);
12         M[l] = q;                  // Store solution
13     }
14
15     return M[n];
16 }
```

- Complejidad espacial: $O(n)$
- Complejidad temporal: $O(n^2)$

Índice

¿Qué hemos aprendido de estos ejemplos?

Hay problemas . . .

- . . . con soluciones recursivas elegantes, compactas e intuitivas
- pero prohibitivamente lentas debido a que resuelven repetidamente los mismos problemas.

Hemos aprendido a:

- **Evitar repeticiones guardando resultados de subproblemas** (*memoización*) . . .
- . . . a expensas de aumentar la complejidad espacial.

Esto se llama Programación Dinámica.

Subestructura óptima

Definición:

Un problema tiene una subestructura óptima si una solución óptima puede construirse eficientemente a partir de las soluciones óptimas de sus subproblemas

- Esto también se conoce como principio de optimalidad
- Esta es una condición necesaria para que se puede aplicar Programación Dinámica
- Ejemplos:
  - Problema de la mochila
  - Corte de tubos
  - Quicksort
  - Mergesort

Paso de Divide y Vencerás a Programación Dinámica

Esquema Divide y Vencerás (DC)

```
1 Solution DC( Problem p ) {
2   if( is_simple(p) ) return trivial(p);
3
4   list<Solution> s;
5   for( Problem q : divide(p) )
6       s.push_back( DC(q) );
7   return combine(s);
8 }
```

Esquema Programación dinámica (DP, recursiva)

```
1 Solution DP( Problem p ) {
2   if( is_solved(p) ) return M[p];
3   if( is_simple(p) ) return M[p] = trivial(p); // or simply: return trivial(p)
4
5   list<Solution> s;
6   for( Problem q : divide(p) )
7       s.push_back( DP(q) );
8   return M[p] = combine(s);
9 }
```

Paso a Programación Dinámica iterativa

Esquema Programación dinámica (iterativa)

```
1  Solution DP( Problem P) {
2      vector<Solution> M;
3      list<Problem> e = enumeration(P);
4
5      while( !e.empty() ) {
6          Problem p = e.pop_front();
7          if( is_simple(p) )
8              M[p] = trivial(p);
9          else {
10             list<Solution> s;
11             for( Problem q : divide(p) ) s.push_back( M[q] );
12             M[p] = combine(s);
13         }
14     }
15     return M[P];
16 }
```

Le enumeración ha de cumplir:

- todo problema en `divide(p)` aparece antes que `p`
- el problema `P` es el último de la enumeración.

Ejemplos de aplicación

- Problemas clásicos para los que resulta eficaz la programación dinámica
  - El problema de la mochilla 0-1
  - Cálculo de los números de Fibonacci
  - Problemas con cadenas:
    - La subsecuencia común máxima (*longest common subsequence*) de dos cadenas.
    - La distancia de edición (*edit distance*) entre dos cadenas.
  - Problemas sobre grafos:
    - El viajante de comercio (*travelling salesman problem*)
    - Caminos más cortos en un grafo entre un vértice y todos los restantes (alg. de Dijkstra)
    - Existencia de camino entre cualquier par de vértices (alg. de Warshall)
    - Caminos más cortos en un grafo entre cualquier par de vértices (alg. de Floyd)

Índice

Obtener el valor del coeficiente binomial

- Identidad de Pascal:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r} \text{ con } n \geq r; \quad \binom{n}{0} = \binom{n}{n} = 1$$

Coeficiente binomial precondición: $\{ n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N} \}$

```
1 unsigned binomial( unsigned n, unsigned r){
2
3     if ( r == 0 || r == n )
4         return 1;
5
6     return binomial( n-1, r-1 ) + binomial( n-1, r );
7 }
```

- Complejidad temporal (relación de recurrencia múltiple)

$$T(n,r) = \begin{cases} 1 & r = 0 \vee r = n \\ 1 + T(n-1, r-1) + T(n-1, r) & \text{en otro caso} \end{cases}$$

La solución recursiva es ineficiente

- Aproximando a una relación de recurrencia lineal:
- si suponemos que:

$$T(n-1,r) \leq T(n-1,r-1)$$

$$T(n,r) \geq g(n,r) = \begin{cases} 1 & n = r \\ 1 + 2g(n-1,r) & \text{en otro caso} \end{cases}$$

$$g(n,r) = 2^k - 1 + 2^k g(n-k,r) \quad \forall k = 1 \ldots (n-r)$$

- Por tanto:

$$T(n,r) \sim g(n,r) \in O(2^{n-r})$$

La solución recursiva es ineficiente

- Si suponemos

$$T(n-1,r) \geq T(n-1,r-1)$$

$$T(n,r) \geq g(n,r) = \begin{cases} 1 & r = 0 \\ 1 + 2g(n-1,r-1) & \text{en otro caso} \end{cases}$$

$$g(n,r) = 2^k - 1 + 2^k g(n-k,r-k) \quad \forall k = 1 \ldots r$$

- Por tanto:

$$T(n,r) \sim g(n,r) \in O(2^r)$$

O combinando los dos:

$$T(n,r) \sim g(n,r) \in O(2^{\min(r,n-r)})$$

¡Esta solución recursiva no es aceptable!

Algunos números

$(n, r) = \binom{n}{r}$	Pasos
(40, 0)	1
(40, 1)	79
(40, 2)	1559
(40, 3)	19759
(40, 4)	182779
(40, 5)	1.3×10^{06}
(40, 7)	3.7×10^{07}
(40, 9)	5.4×10^{08}
(40, 11)	4.6×10^{09}
(40, 15)	8.0×10^{10}
(40, 17)	1.8×10^{11}
(40, 20)	2.8×10^{11}

$(n, r) = \binom{n}{r}$	Pasos
(2, 1)	3
(4, 2)	11
(6, 3)	39
(8, 4)	139
(10, 5)	503
(12, 6)	1847
(14, 7)	6863
(16, 8)	25739
(18, 9)	97239
(20, 10)	369511
(22, 11)	1410863
(24, 12)	5408311

- Caso más costoso: $n = 2r$; crecimiento aprox. 2^n.
- los resultados son claramente prohibitivos

La innecesaria repetición de cálculos

- ¿Por qué es ineficiente?
  - Los problemas se reducen en subproblemas de tamaño similar $(n-1)$.
  - Un problema se divide en dos subproblemas, y así sucesivamente.

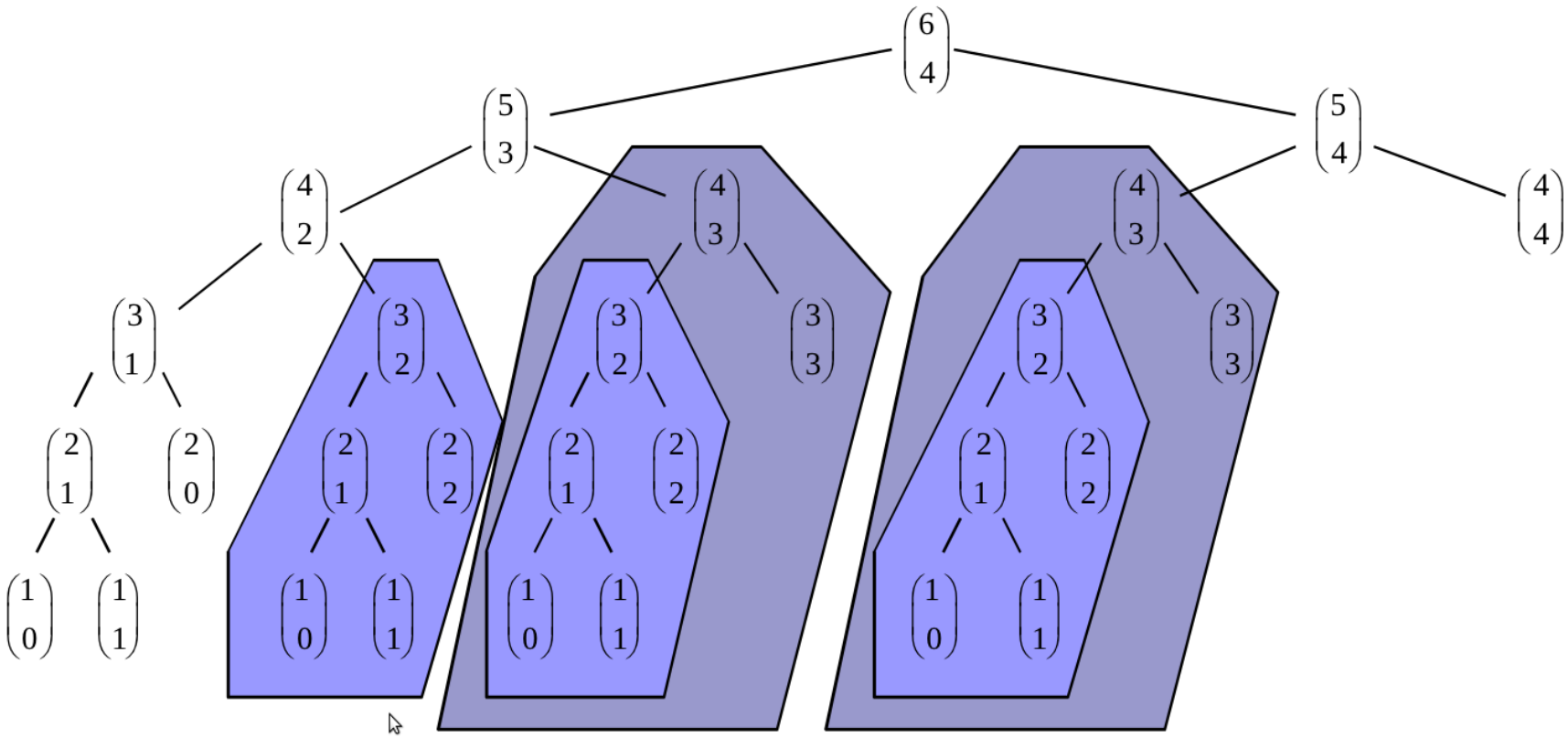
  $\Rightarrow$ Esto lleva a complejidades prohibitivas (p.e. exponenciales)
- Pero, ¡el total de subproblemas diferentes no es tan grande!
  - sólo hay nr posibilidades distintas

¡La solución recursiva está generando y resolviendo el mismo problema muchas veces!

- ¡Cuidado! la ineficiencia no es debida a la recursividad

La innecesaria repetición de cálculos

- Solución recursiva: ejemplo para $n = 6$ y $r = 4$



- INCONVENIENTE: subproblemas repetidos.
  - Pero sólo hay nr subproblemas diferentes: $\Rightarrow$ uso de almacenes

¿Cómo evitar la repetición de cálculos?

$\Rightarrow$ almacenar los valores ya calculados para no recalcularlos:

Solución recursiva mejorada $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1  const unsigned SENTINEL = 0;
2
3  unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
4
5      if( M[n][r] != SENTINEL )
6          return M[n][r];
7
8      if( r == 0 || r == n )
9          return M[n][r] = 1;
10
11     M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
12
13     return M[n][r];
14 }
15
16 unsigned binomial( unsigned n, unsigned r) {
17     vector<vector<unsigned>> M( n+1, vector<unsigned>(r+1, SENTINEL));
18     return binomial( M, n, r);
19 }
```

Memoización (para varios problemas)

Memoización $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1  unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
2      if( M[n][r] != 0 ) return M[n][r];
3      if( r == 0 || r == n ) return M[n][r] = 1;
4      M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
5      return M[n][r];
6  }
7
8  const unsigned MAX_N = 100;
9
10 unsigned binomial( unsigned n, unsigned r) {
11     static vector<vector<unsigned>> M;
12     static bool initialized = false;
13
14     if( !initialized ) {
15         M = vector<vector<unsigned>>(MAX_N, vector<unsigned>(MAX_N, SENTINEL));
16         initialized = true;
17     }
18
19     return binomial( M, n, r);
20 }
```

Memoización (para varios problemas)

Memoización (functores) $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1  const unsigned SENTINEL = 0;
2  const unsigned MAX_N = 100;
3
4  class Binomial {
5  public:
6      Binomial( unsigned max_n = MAX_N ) : M(
7          vector<vector<unsigned>>(max_n+1, vector<unsigned>(max_n+1, SENTINEL))
8      ){};
9
10     unsigned operator()( unsigned n, unsigned r ) {
11         if( M[n][r] != SENTINEL ) return M[n][r];
12         if( r == 0 || r == n ) return M[n][r] = 1;
13         M[n][r] = operator()(n-1, r-1) + operator()( n-1, r);
14         return M[n][r];
15     }
16
17 private:
18     vector<vector<unsigned>> M;
19 };
20
21 Binomial binomial(40);  // use: a = binomial(30,20);
```

Algunos números

$(n, r) \equiv \binom{n}{r}$	Ingenuo	Mem.
(40, 0)	1	1
(40, 1)	79	79
(40, 2)	1559	116
(40, 3)	19759	151
(40, 4)	182779	184
(40, 5)	1.3×10^{06}	215
(40, 7)	3.7×10^{07}	271
(40, 9)	5.4×10^{08}	319
(40, 11)	4.6×10^{09}	359
(40, 15)	8.0×10^{10}	415
(40, 17)	1.8×10^{11}	432
(40, 20)	2.8×10^{11}	440

$(n, r) \equiv \binom{n}{r}$	Ingenuo	Mem.
(2, 1)	3	3
(4, 2)	11	8
(6, 3)	20	15
(8, 4)	139	24
(10, 5)	503	35
(12, 6)	1847	48
(14, 7)	6863	64
(16, 8)	25739	80
(18, 9)	97239	99
(20, 10)	369511	120
(22, 11)	1410863	143
(24, 12)	5408311	168

- En el caso $n = 2r$, el crecimiento es del tipo $(n/2)^2 + n \in \Theta(n^2)$.
- Los resultados mejoran muchísimo cuando se añade un almacén

¿Cómo evitar la recursividad?

- ¿Se puede evitar la recursividad? En este caso sí
  - Resolver los subproblemas de menor a mayor
  - Almacenar sus soluciones en una tabla $M[n][r]$ donde

$$M[i][j] = \binom{i}{j}$$

    - El almacén de resultados parciales permite evitar repeticiones
    - La tabla se inicializa con la solución a los subproblemas triviales:

$$M[i][0] = 1 \qquad \forall i = 1, \cdots, (n-r)$$
$$M[i][i] = 1 \qquad \forall i = 1, \cdots, r$$

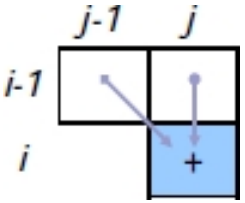
Puesto que

$$\binom{m}{0} = \binom{m}{m} = 1, \quad \forall m \in \mathbb{N}$$

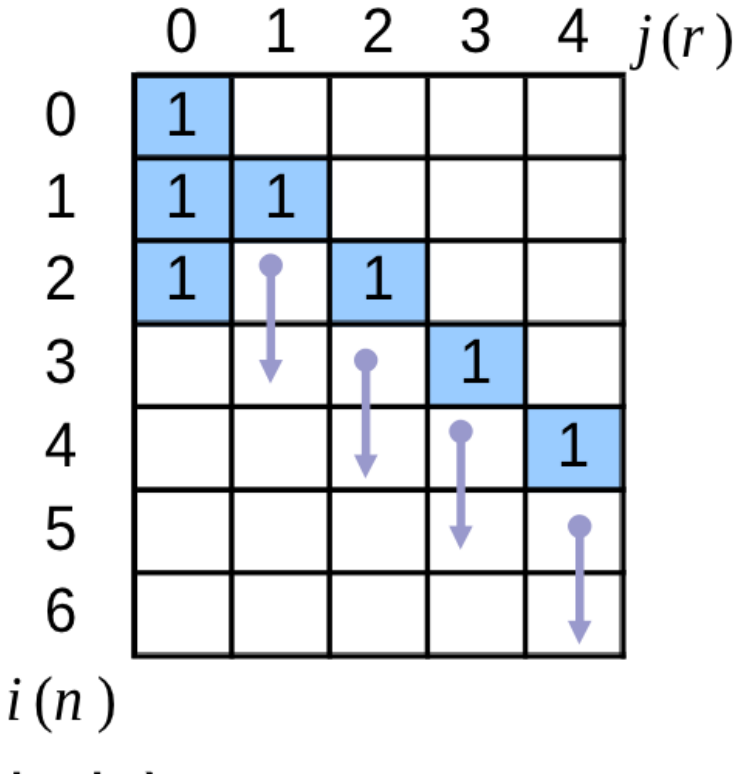
Recorrido de los subproblemas

- Resolviendo los subproblemas en sentido ascendente y almacenando sus soluciones:

$$M[i][j] = M[i-1][j-1] + M[i-1][j]$$

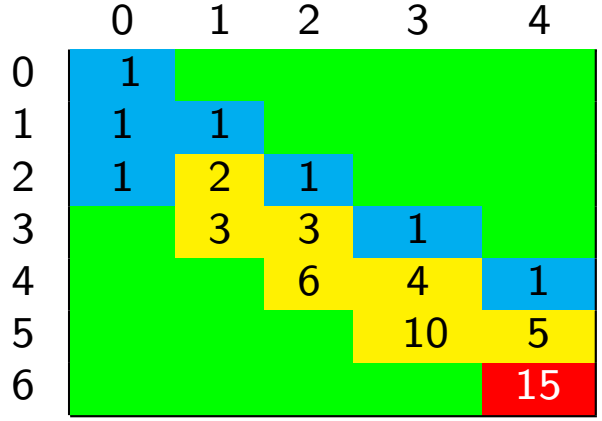


$$\forall (i,j) : (1 \leq j \leq r,\ j+1 \leq i \leq n-r+j)$$



Una solución polinómica (mejorable)

- Ejemplo: Sea $n = 6$ y $r = 4$



Celdas sin utilizar ¡desperdicio de memoria!
Instancias del caso base: perfil o contorno de la matriz
Soluciones de los subproblemas. Obtenidos, en este caso, de arriba hacia abajo y de izquierda a derecha
Solución del problema inicial. $M[6][4] = \binom{6}{4}$

Solución trivial de programación dinámica $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1  unsigned binomial(unsigned n,unsigned r){
2      unsigned M[n+1][r+1];
3
4      for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
5      for (unsigned i=1; i <= r;   i++) M[i][i]= 1;
6
7      for (unsigned j=1; j<=r; j++)
8          for (unsigned i=j+1; i<=n-r+j; i++)
9              M[i][j]= M[i-1][j-1] + M[i-1][j];
10     return M[n][r];
11 }
```

Una solución polinómica (mejorable)

Solución trivial de programación dinámica $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1  unsigned binomial(unsigned n, unsigned r){
2      unsigned M[n+1][r+1];
3
4      for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
5      for (unsigned i=1; i <= r;   i++) M[i][i]= 1;
6
7      for (unsigned j=1; j<=r; j++)
8          for (unsigned i=j+1; i<=n-r+j; i++)
9              M[i][j]= M[i-1][j-1] + M[i-1][j];
10     return M[n][r];
11 }
```

- **Coste temporal exacto:**

$$T(n,r) = 1 + \sum_{i=0}^{n-r} 1 + \sum_{i=1}^{r} 1 + \sum_{j=1}^{r} \sum_{i=j+1}^{n-r+j} 1 = 1 + (n-r+1) + r + r(n-r) \in \Theta(rn)$$

- Idéntico al descendente con memoización (almacén)

Complejidad espacial

Solución trivial de programación dinámica $\{n \geq r,\ n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1  unsigned binomial(unsigned n, unsigned r){
2      unsigned M[n+1][r+1];
3
4      for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
5      for (unsigned i=1; i <= r;   i++) M[i][i]= 1;
6
7      for (unsigned j=1; j<=r; j++)
8          for (unsigned i=j+1; i<=n-r+j; i++)
9              M[i][j]= M[i-1][j-1] + M[i-1][j];
10     return M[n][r];
11 }
```

- **Coste espacial:** $\Theta(rn)$ ¿Se puede mejorar?

Mejorando la complejidad espacial

- Ejercicios propuestos: Reducción de la complejidad espacial:
  - Modificar la función anterior de manera que el almacén no sea más que dos vectores de tamaño $1 + \min(r, n - r)$
  - Modificar la función anterior de manera que el almacén sea un único vector de tamaño $1 + \min(r, n - r)$
  - Con estas modificaciones, ¿queda afectada de alguna manera la complejidad temporal?

Análisis y diseño de algoritmos

5. Algoritmos voraces

José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

Índice

Índice

Problema de la Mochila continuo

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad W (peso)
- Seleccionar un conjunto de objetos de forma que:
  - no se sobrepase el peso W (restricción)
  - el valor transportado sea máximo (función objetivo)
  - se permite fraccionar los objetos
- El problema se reduce a:
  - Seleccionar un subconjunto de (fracciones de) los objetos disponibles,
  - que cumpla las restricciones, y
  - que maximice la función objetivo.
- ¿Cómo resolverlo?
  - Se necesita un criterio que decida qué objeto seleccionar en cada momento (criterio de selección)

Posibles criterios

- Supongamos el siguiente ejemplo:

$$W = 12 \quad w = (6, 5, 2) \quad v = (48, 35, 20) \quad v/w = (8, 7, 10)$$

Criterios	**Solución**	**Peso** W	**Valor** v
valor decreciente	$(1, 1, \frac{1}{2})$	12	93
peso creciente	$(\frac{5}{6}, 1, 1)$	12	95
valor especifico decreciente $(\frac{v_i}{w_i})$	$(1, \frac{4}{5}, 1)$	12	**96**

Formalización

- Solución: $X = (x_1, x_2, \ldots, x_n)$, $x_i \in [0, 1]$
  - $x_i = 0$: no se selecciona el objeto i
  - $0 < x_i < 1$: fracción seleccionada del objeto i
  - $x_i = 1$: se selecciona el objeto i completo
- Función objetivo:

$$\max \left(\sum_{i=1}^{n} x_i v_i \right) \qquad \text{valor transportado}$$

- Restricción:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Correción del algoritmo

Teorema: El algoritmo encuentra la solución óptima

Sea $X = (x_1, x_2, \ldots, x_n)$ la solución del algoritmo $(\sum_{i=1}^{n} x_i w_i = W)$
— supongamos que está ordenada por v_i/w_i
Sea $Y = (y_1, y_2, \ldots, y_n)$ otra solución factible $(\sum_{i=1}^{n} y_i w_i = Q \leq W)$
Tenemos que:

$$V(X) - V(Y) = \sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{n} (x_i - y_i) \frac{v_i}{w_i} w_i$$

Como $(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$ donde $j : x_i = 1$ si $i < j$ y $x_i = 0$ si $i > j$
concluimos que:

$$V(X) - V(Y) \geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i = \frac{v_j}{w_j}(W - Q) \geq 0$$

algoritmo voraz (valor óptimo)

```
1  double knapsack(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<size_t> idx(w.size());  // objects sorted by value density
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( begin(idx), end(idx),
10         [&v,&w]( size_t x, size_t y ){      // function "bigger than"
11             return v[x]/w[x] > v[y]/w[y];    // sorts form bigger to lower
12         }
13     );
14     double acc_v = 0.0;
15     for( auto i : idx ) {
16         if( w[i] >= W ) {
17             acc_v += W/w[i] * v[i];
18             break;
19         }
20         acc_v += v[i];
21         W -= w[i];
22     }
23     return acc_v;
24 }
```

algoritmo voraz (vector óptimo)

```
1  vector<double> knapsack_W(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<size_t> idx(w.size());
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8      sort( begin(idx), end(idx), [&v,&w]( size_t x, size_t y ){
9          return v[x]/w[x] > v[y]/w[y]; } );
10
11     vector<double> x(w.size(),0);
12     double acc_v = 0.0;
13     for( auto i : idx ) {
14         if( w[i] >= W ) {
15             acc_v += W/w[i] * v[i];
16             x[i] = W/w[i];
17             break;
18         }
19         acc_v += v[i];
20         W -= w[i];
21         x[i] = 1.0;
22     }
23     return x;
24 }
```

Índice

Algoritmos voraces (*Greedy*)

Definición:

Un algoritmo voraz es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción local óptima en cada paso con la esperanza de llegar a una solución general óptima

Dicho de otra forma:

- descompone el problema en un conjunto de decisiones . . .
- . . . y elige la mas prometedora
- nunca reconsidera las decisiones ya tomadas

Características

- Son algoritmos eficientes y fáciles de implementar
- Es necesario un buen criterio de selección para tener garantías
- A veces se usan para obtener soluciones aproximadas
  - Puede que no se encuentre la solución óptima
  - Incluso pude que no se encuentre ninguna solución
- Se aplican mucho:
  - Cuando es suficiente una solución aproximada
  - Como cota en problemas con muy alta complejidad

Índice

Índice

Problema de la mochila discreta (sin fraccionamiento)

- Sean n objetos de valores v_i y pesos w_i y una mochila con capacidad W. Seleccionar un conjunto de objetos de forma que:
  - no sobrepase el peso W
  - el valor transportado sea máximo
- Formulación del problema:
  - Expresaremos la solución mediante un vector $(x_1, x_2, \ldots, x_n)$ donde x_i representa la decisión tomada con respecto al elemento i.
  - Función objetivo:

$$\max\left(\sum_{i=1}^{n} x_i v_i\right) \qquad \text{valor transportado}$$

  - Restricciones

$$\sum_{i=1}^{n} x_i w_i \leq W \qquad x_i \in \{0,1\} \begin{cases} x_i = 0 & \text{no se selecciona el objeto } i \\ x_i = 1 & \text{sí se selecciona el objeto } i \end{cases}$$

problema de la mochila discreta (sin fraccionamiento)

- En este caso el método voraz no resuelve el problema
- Ejemplo:

$$W = 120 \quad w = (60, 60, 20) \quad v = (300, 300, 200) \quad v/w = (5, 5, 10)$$

  - solución voraz: $(0, 1, 1) \rightarrow$ valor total $= 500$
  - solución óptima: $(1, 1, 0) \rightarrow$ valor total $= 600$

$\Rightarrow$ El ordenar por valor específico no conduce al óptimo

Algoritmo de la mochila (discreto)

```
1  double knapsack_d(
2      const vector<double> &v,
3      const vector<double> &w,
4      double W
5  ) {
6      vector<size_t> idx( w.size() );
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( begin(idx), end(idx), [&w,&v]( size_t x, size_t y ){
10         return v[x]/w[x] > v[y]/w[y];
11     } );
12
13     double acc_v = 0.0;
14
15      for( auto i : idx ) {
16
17         if( w[i] <= W ) {
18             acc_v += v[i];
19             W -= w[i];
20         }
21     }
22
23     return acc_v;
24 }
```

Índice

El problema del cambio

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C:
  - Una solución es una secuencia de decisiones

$$S = (s_1, s_2, \ldots, s_n)$$

  - La función objetivo es

$$\min |S|$$

  - La restricción es

$$\sum_{i=1}^{n} \text{valor}(s_i) = M$$

- La solución voraz es tomar en cada momento la moneda de mayor valor posible

Ejemplo

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C:
- Sea $M = 65$

C	S	n	**Solución**
$\{1, 5, 25, 50\}$	$(50, 5, 5, 5)$	4	óptima y voraz
$\{1, 5, 7, 25, 50\}$	$(50, 7, 7, 1)$	4	óptima y voraz
	$(50, 5, 5, 5)$	4	óptima, pero no voraz
$\{1, 5, 11, 25, 50\}$	$(50, 11, 1, 1, 1, 1)$	6	voraz, pero no óptima
	$(50, 5, 5, 5)$	4	óptima, pero no voraz
$\{5, 11, 25, 50\}$	$(50, 11, ?)$	???	no encuentra solución
	$(50, 5, 5, 5)$	4	óptima, pero no voraz

Índice

Árbol de recubrimiento de coste mínimo

- Partimos de un grafo $g = (V, A)$:
  - conexo
  - no dirigido
  - arcos ponderados
  - con arcos positivos
- Queremos el árbol de recubrimiento de g de coste mínimo:
  - subgrafo de g
  - que contenga todos los vértices (recubrimiento)
  - conexo (árbol)
  - sin ciclos (árbol)
  - coste mínimo

Algorítmos de Prim y Kruskal

- Existen al menos dos algoritmos voraces que resuelven este problema,
  - algortimo de Prim (hacer crecer un árbol)
  - algortimo de Kruskal (añadir aristas evitando ciclos)
- En ambos se van añadiendo arcos de uno en uno a la solución
- La diferencia está en la forma de elegir los arcos a añadir

Índice

Algoritmo de Prim

- Se mantiene un conjunto de vértices explorados
- Se coge un vértice al azar y se añade al conjunto de explorados
- en cada paso:
  - buscar el arco de mínimo peso que va de un vértice explorado a uno que no lo está
  - añadir el arco a la solución y el vértice a los explorados

Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1  list<edge> prim( const Graph &g ){
2      unsigned n = g.size();
3      vector<bool> visited(n,false);
4      list<edge> r;
5
6      edge e{0,0};
7      for( unsigned i=0; i<n-1; i++) {
8          visited[e.d] = true;
9
10         e = min_edge(g, visited);
11
12         r.push_back(e);
13
14     }
15     return r;
16 }
```

Estructuras de datos

```
1  using Graph = vector<vector<unsigned>>;
2
3  struct edge {
4      unsigned s;
5      unsigned d;
6  };
7
8  // Graph instantiation example
9  const unsigned INF
10     = numeric_limits<unsigned>::max();
11
12 Graph g{
13    { INF,   3,   1,   6, INF, INF },
14    {   3, INF,   5, INF,   3, INF },
15    {   1,   5, INF,   5,   6,   4 },
16    {   6, INF,   5, INF, INF,   2 },
17    { INF,   3,   6, INF, INF,   6 },
18    { INF, INF,   4,   2,   6, INF }
19 };
```

Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1  edge min_edge(
2      const Graph &g,
3      const vector<bool> &visited
4  ) {
5
6      unsigned n = g.size();
7      unsigned min = numeric_limits<unsigned>::max();
8      edge e;
9      for( unsigned i = 0; i < n; i++ )
10         for( unsigned j = 0; j < n; j++ )
11             if( visited[i] && !visited[j] )
12                 if( g[i][j] < min ) {
13                     min = g[i][j];
14                     e = {i, j};
15                 }
16
17     return e;
18 }
```

Complejidades:

- `min_edge`: $O(V^2)$
- `prim`: $O(V^3)$

¿Se puede mejorar?

Algoritmo de Prim

Mejora:

- No hace falta recorrer todos los arcos cada vez
- Si cambia el mínimo es a causa del último vértice añadido

$\Rightarrow$ Hay que guardarse, para cada vértice no visitado:
el arco de peso minimo procedente de un vértice visitado

Algoritmo de Prim

Algoritmo de Prim (con indices)

```
1  list<edge> prim( const Graph &g ) {
2      unsigned n = g.size();
3      vector<bool> visited( n, false);                    // visited vertex
4      vector<unsigned> lcv(n,0);          // source of the lowest cost vertex
5      list<edge> r;
6
7      edge e{0,0};
8      for( unsigned i = 0; i < n-1; i++ ) {
9
10         visited[e.d] = true;
11         update_idx( g, lcv, e.d );                  // update the index
12
13         e = min_edge( g, lcv, visited );
14
15         r.push_back(e);
16     }
17     return r;
18 };
```

Algoritmo de Prim

Actualizar índices

```
1  void update_idx(
2    const  Graph      &g,
3    vector<unsigned> &lcv,
4    unsigned          nv
5  ) {
6
7    unsigned n = g.size();
8    for( unsigned j = 0; j < n; j++ )
9      if( g[lcv[j]][j] > g[nv][j] )
10       lcv[j] = nv;
11
12 }
```

Buscar mejor arco

```
1  edge min_edge(
2    const Graph              &g,
3    const vector<unsigned> &lcv,
4    const vector<bool>     &visited
5  ) {
6    unsigned n = g.size();
7
8    edge e;
9    for( unsigned j = 0; j < n; j++ ) {
10     if( !visited[j]
11         && g[lcv[j]][j]<min ) {
12       min = g[lcv[j]][j];
13       e = { lcv[j], j};
14     }
15   }
16   return e;
17 }
```

Índice

Conjuntos disjuntos (Disjoint-set)

Tenemos una partición de un conjunto de datos y queremos:

- Inicializar la partición: cada elemento en un bloque distinto
- Saber a qué bloque pertenece un elemento (`find`)
- Poder unir dos bloques de la partición (`union`)

Ejemplo:

- Objetos: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Inicialización: $\{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\}$
- Ejemplo de conjunto disjunto: $\{\{0, 1, 2\}, \{3, 4, 7, 8, 9\}, \{5\}, \{6\}\}$
- Cada bloque tiene un representante, p.e. el menor número $\{\{0, 1, 2\}, \{3, 4, 7, 8, 9\}, \{5\}, \{6\}\}$
- `find(·)` devuelve el nombre del representante. `find(8)` = 3
- `union(8,2)` une los conjuntos representados por 3 y 0.

$$\{\{0, 1, 2, 3, 4, 7, 8, 9\}, \{5\}, \{6\}\}$$

Conjuntos disjuntos (Quick-find)

Una forma de abordarlo:

- Asignamos una etiqueta a cada partición
- `find`: consultar la etiqueta
- `union`: reetiquetamos los elementos de uno de los bloques

Ejemplos:

`init`

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

`union(1,3)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	7	8	9

`union(5,7)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	5	8	9

`union(7,3)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	1	6	1	8	9

¿Compejidades?

Conjuntos disjuntos (Quick-find)

Una forma de abordarlo:

- Asignamos una etiqueta a cada partición
- `find`: consultar la etiqueta
- `union`: reetiquetamos los elementos de uno de los bloques

Ejemplos:

`init`

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

`union(1,3)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	7	8	9

`union(5,7)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	5	8	9

`union(7,3)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	1	6	1	8	9

¿Compejidades?

- `find`: $O(1)$

Conjuntos disjuntos (Quick-find)

Una forma de abordarlo:

- Asignamos una etiqueta a cada partición
- `find`: consultar la etiqueta
- `union`: reetiquetamos los elementos de uno de los bloques

Ejemplos:

`init`

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

`union(1,3)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	7	8	9

`union(5,7)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	5	8	9

`union(7,3)`

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	1	6	1	8	9

¿Compejidades?

- `find`: $O(1)$
- `union`: $O(n)$

Conjuntos disjuntos (Quick-union))

Otra forma de abordarlo:

- Representamos cada partición como un árbol
  - cada elemento almacena la etiqueta de otro elemento que está en el mismo bloque
  - el elemento raíz del árbol almacena su propia etiqueta
- `find`: seguir la cadena de etiquetas hasta llegar a la raíz
  - Complejidad: $O(n)$
- `union`: cambiamos la etiqueta de la raíz de uno de los árboles al elemento raíz del otro
  - Complejidad $O(n)$ (hay que buscar la raíz)

Ejemplos Quick-union

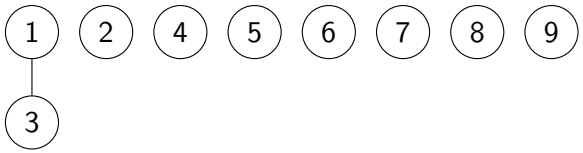
Como vector:

- los elementos del vector se interpretan como punteros al padre
- si un elemento apunta a si mismo se interpreta como un nodo raíz
- (juntamos sobre el representante del primer argumento)

`init`

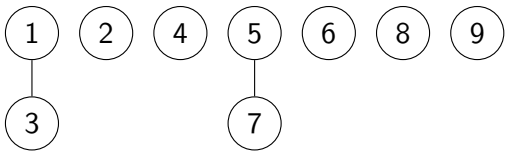
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

`union(1,3)`



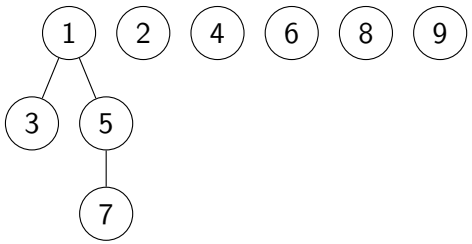
0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	7	8	9

`union(5,7)`



0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	5	6	5	8	9

`union(3,7)`



0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	1	6	5	8	9

Conjuntos disjuntos (ineficiente)

```
1  class disjoint_set {
2  public:
3      disjoint_set( unsigned n ) : s(n) {
4          iota( s.begin(), s.end(), 0 );   // the label is the index
5      }
6
7      unsigned find( unsigned x ) {        // recursivelly find the root
8          if( s[x] == x )
9              return x;
10         return find(s[x]);
11     }
12
13     void merge( unsigned x, unsigned y ) { // "union" is a reserved word :-(
14         unsigned x_root = find(x);
15         unsigned y_root = find(y);
16
17         s[x_root] = y_root;
18     }
19
20 private:
21     vector<unsigned> s;
22 };
```

Conjuntos disjuntos (Disjoint-set)

Mejoras:

- Al unir dos árboles: juntar el menos profundo sobre el más profundo
  - hay que llevar un registro de la profundidad del árbol
  - con esto la complejidad del `find` cae a $O(\log(n))$
- Al hacer un `find`: cambiar la etiqueta para que apunte directamente a la raíz (*compresion de camino*)

Complejidad:

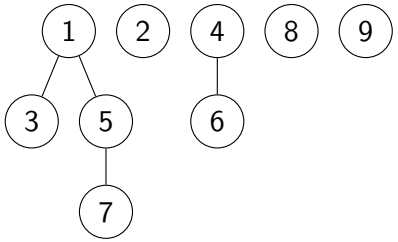
- Para cualquier sucesión de m operaciones `find` y `union`:

$$O(m\,\alpha(n)) \qquad \text{(a esto se llama complejidad amortizada)}$$

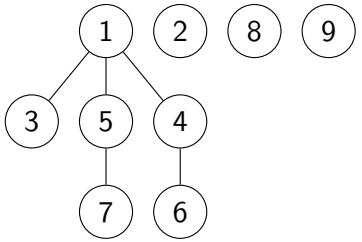
donde $\alpha(n)$ es la función inversa a la función de Akermann ($A^{-1}(n,n)$)

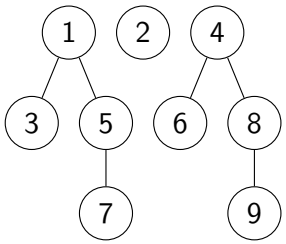
- $A(4,4) \approx 2^{2^{10^{19729}}} \Rightarrow \alpha\left(2^{2^{10^{19729}}}\right) \approx 4$

Ejemplos

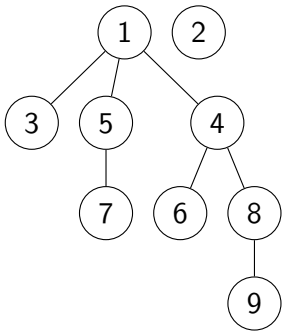


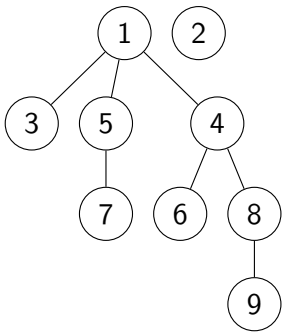
`union(6,5)`



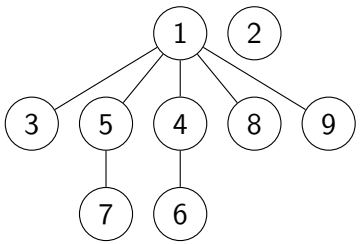


`union(7,8)`
(sin compresión de camino)





`find(9)`



Conjuntos disjuntos (eficiente)

```
1  class disjoint_set {
2  private:
3
4      struct node {
5          unsigned parent;
6          unsigned rank;
7      };
8
9      vector<node> s;
10
11 public:
12     disjoint_set( unsigned n ) : s(n) {
13         for( unsigned i = 0; i < s.size(); i++ )
14             s[i] = node{i,0};
15     }
16
17     unsigned find( unsigned x ) {
18         if( s[x].parent != x )
19             s[x].parent = find(s[x].parent);
20         return s[x].parent;
21     }
22
23 // ....
```

Conjuntos disjuntos (eficiente)

```
1  // ...
2
3      void merge( unsigned x, unsigned y ) {
4          unsigned x_root = find(x);
5          unsigned y_root = find(y);
6
7          if( s[x_root].rank < s[y_root].rank )
8              s[x_root].parent = y_root;
9          else if( s[x_root].rank > s[y_root].rank )
10             s[y_root].parent = x_root;
11         else {
12             s[y_root].parent = x_root;
13             s[x_root].rank++;
14         }
15     }
16 };
```

Índice

Algoritmo de Kruskal

- mantener una partición de los vértices
- mientras queda más de un bloque
  - buscar el arco de menor peso que una dos bloques distintos
    - (Esto asegura que no habrán ciclos)
  - añadir el arco a la solución
  - unir los dos bloques
- Complejidad: $O(E \log E)$

Algoritmo de Kruskal

```
1  list<edge> kruskal( const Graph &g ) {
2      struct node { unsigned w; edge e; };
3      int n = g.size();
4
5      vector<node> v;  // make a list of edges
6      for( unsigned i = 1; i < n; i++ )
7          for( unsigned j = 0; j < i; j++ )
8              v.push_back({ g[i][j], {i, j} });
9
10     sort( begin(v), end(v), []( const node &n1, const node &n2) {
11         return n1.w < n2.w;
12     });
13
14     list<edge> r;
15     disjoint_set s(n);
16     for( auto i: v ) {
17         if( s.find(i.e.s) != s.find(i.e.d) ) {
18             r.push_back(i.e);
19             s.merge( i.e.s, i.e.d );
20         }
21     }
22     return r;
23 }
```

Comparación

- El algoritmo de Prim tiene una complejidad de $O(V^2)$
- El algoritmo del Kruskal tiene una complejidad de $O(E \log E)$

¿Cuándo usar cada algoritmo?

- En el peor caso el número de arcos de un grafo es $E \in O(V^2)$
- En el peor caso, la complejidad del algoritmo de Kruskal será:

$$O(E \log E) = O(V^2 \log V^2) = O(V^2 \log V)$$

- El algoritmo de Kruskal es mejor cuando el grafo es disperso

Índice

El fontanero diligente

- Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.
- En otras palabras, si llamamos E_i a lo que espera el cliente i-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E = \sum_{i=1}^{n} E_i \quad \text{con} \quad E_i = \sum_{j=1}^{i} t_j$$

Índice

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i.
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí.
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} b_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.

Análisis y diseño de algoritmos

6. Vuelta atrás

José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

Índice

Índice

Ejemplo Introductorio

El problema de la mochila (general)

Dados:

- n objetos con valores v_i y pesos w_i
- una mochila que solo aguanta un peso máximo W

Seleccionar un conjunto de objetos de forma que:

- no se sobrepase el peso límite W (restricción)
- el valor transportado sea máximo (función objetivo)

- **¿Cómo obtener la solución óptima?**
  - Programación dinámica: objetos no fragmentables y pesos discretos
  - Algoritmos voraces: objetos fragmentables
- **– No podemos fragmentar los objetos y los pesos son valores reales –**

Formalización del problema

- Solución: $X = (x_1, x_2, \ldots, x_n) \quad x_i \in \{0, 1\}$
- Restricciones:
  - Implícitas:

$$x_i \in \begin{cases} 0 & \text{no se selecciona el objeto } i \\ 1 & \text{se selecciona el objeto } i \end{cases}$$

  - Explícitas:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

- Función objetivo:

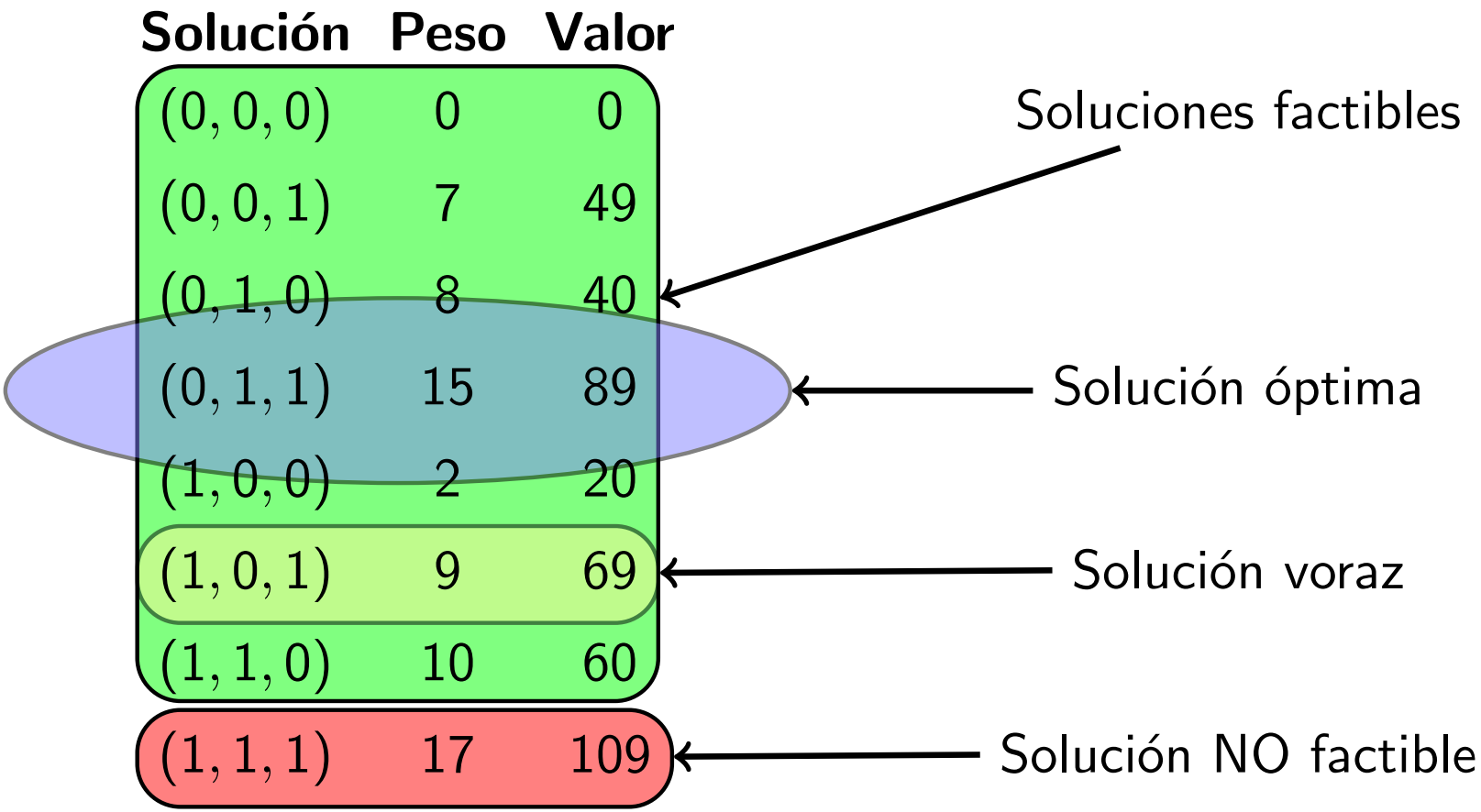
$$\max \sum_{i=1}^{n} x_i v_i$$

Tipos de soluciones

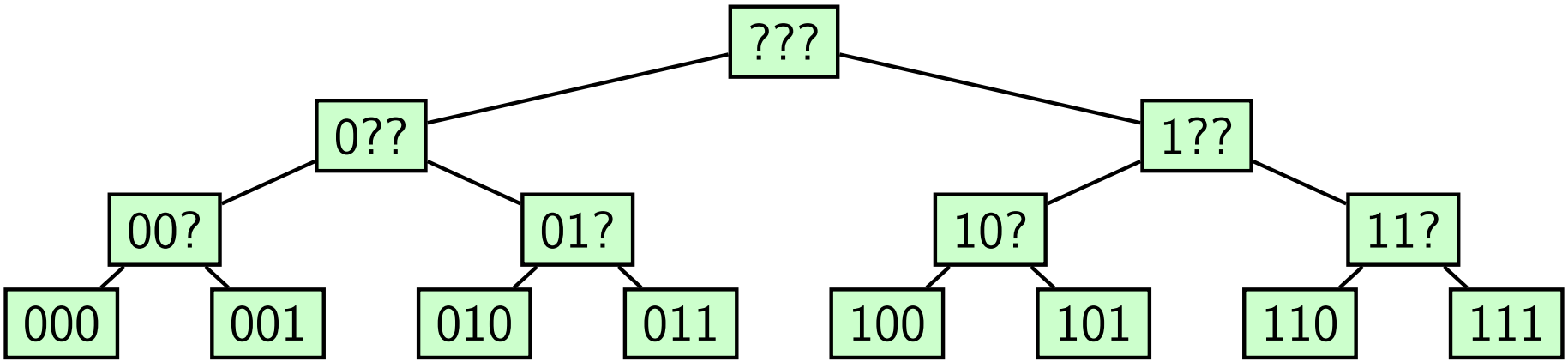
- Supongamos el siguiente ejemplo:

$$W = 16 \qquad w = (2, 8, 7) \qquad v = (20, 40, 49)$$

- Combinaciones posibles (espacio de soluciones):



Generación de todas las combinaciones



Recorrer todas las combinaciones

```
1  void comb(size_t k, vector <unsigned> &x){
2      if( k == x.size() ) {  // It is a leaf
3          cout << x << endl;  // to overload
4          return;
5      }                      // It is not a leaf
6      for ( unsigned j = 0; j < 2; j++) {
7          x[k]=j;             // New alternative
8          comb(k+1, x);                 // expand
9      }
10 }
```

Programa principal

```
1 void comb(size_t n) {
2     vector<unsigned> x(n);
3     comb( 0, x );
4 }
```

Complejidad temporal:
$\Theta(n2^n)$

¿Cómo generar solo soluciones factibles?

- Solo imprimimos las soluciones (hojas) que cumplen:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Generación de todas las soluciones factibles

Cálculo del peso de la solución x

```
1 double weight( const vector<double> &w, const vector<unsigned> &x){
2     double acc_w = 0.0;
3     for (size_t i = 0; i < w.size(); i++ ) acc_w += x[i] * w[i];
4     return acc_w;
5 }
```

Imprime todas las soluciones factibles

```
1  void feasible(
2      const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x
4  ){
5      if( k == x.size() ) {   // It is a leaf
6          if( weight( w, x ) <= W )
7              cout << x << endl;
8          return;
9      }                       // It is not a leaf
10     for (unsigned j=0; j<2; j++) {
11         x[k]=j;
12         feasible(w,W,k+1,x);       // expand
13     }
14 }
```

Llamada principal

```
1 void feasible(
2     const vector<double> &w,
3     double W
4 ){
5     vector<unsigned> x(w.size());
6     feasible( w, W, 0, x );
7     return;
8 }
```

Complejidad temporal: $\Theta(n2^n)$

Búsqueda del óptimo

- Para encontrar el óptimo hay que:
  - recorrer todas las soluciones factibles
  - calcular su valor y . . .
  - . . . quedarse con el mayor de todos

Búsqueda del valor óptimo

```
1  double value( const vector<double> &v, const vector<unsigned> &x ) {
2      double r = 0.0;
3      for( size_t i = 0; i < v.size(); i++ ) r += v[i] * x[i];
4      return r;
5  }
6  void knapsack( const vector<double> &v, const vector<double> &w, double W,
7      size_t k, vector<unsigned> &x, double &best_v
8  ){
9      if( k == x.size() ) {  // It is a leaf
10         if( weight( w, x ) <= W )
11             best_v = max( best_v, value(v,x) );
12         return;
13     }
14     for (unsigned j=0; j<2; j++) {
15         x[k]=j;
16         knapsack( v, w, W, k+1, x, best_v );      // expand
17     }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W){
20     vector<unsigned> x(w.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, best_v );
23     return best_v;
24 }
```

Búsqueda de la asignación óptima

```
1  void knapsack( const vector<double> &v, const vector<double> &w, double W,
2      size_t k, vector<unsigned> &x, double &best_v, vector<unsigned> &sol
3  ){
4      if( k == x.size() ) { // It is a leaf
5          if( weight( w, x ) <= W ){
6              double actual_v = value(v,x);
7              if( actual_v > best_v ) {
8                  best_v = actual_v;
9                  sol = x;
10             }
11         }
12         return;
13     }
14     for (unsigned j=0; j<2; j++) {
15         x[k]=j;
16         knapsack( v, w, W, k+1, x, best_v );      // expand
17     }
18 }
19 auto knapsack( const vector<double> &v, const vector<double> &w, double W){
20     vector<unsigned> x(v.size()), sol(v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, best_v, sol );
23     return make_tuple( best_v, sol );
24 }
```

¿Podemos acelerar el programa?

- Sí, evitando explorar ramas que no pueden dar soluciones factibles
- Por ejemplo:
  - hemos construido una solución hasta el elemento k:

$$(x_1, x_2, \ldots, x_k, ?, \ldots, ?)$$

  - si tenemos que:

$$\sum_{i=1}^{k} x_i w_i \geq W$$

$\Rightarrow$ Ninguna expansión de esta solución puede ser factible

Eliminando ramas por peso

```
1  double weight( const vector<double> &w, size_t k, const vector<double> &x ) {
2     double r = 0.0;
3     for( size_t i = 0; i < k; i++ ) r = w[i] * x[i];
4     return r;
5  }
6  void knapsack( const vector<double> &v, const vector<double> &w, double W,
7      size_t k, vector<unsigned> &x, double &best_v
8  ){
9      if( k == x.size() ) {                                  // if it is a leaf
10         best_v = max( best_v, value(v,x));
11         return;
12     }                                                    // if it's not a leaf
13     for (unsigned j = 0; j < 2; j++ ) {
14         x[k]=j;
15         if ( weight(w, k+1, x ) <= W )                    // if it is feasible
16             knapsack( v, w, W, k+1, x, best_v );                  // expand
17   }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
20     vector<unsigned> x( v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, size_t k, const vector<unsigned> &x, best_v );
23     return best_v;
24 }
```

¿se puede mejorar?

- Nótese que el peso se puede ir calculando a medida que se rellena la solución

Calculando el peso de forma incremental

```
1  void knapsack( const vector<double> &v, const vector<double> &w, double W,
2      size_t k, vector<unsigned> &x, double acc_w, double &best_v
3  ){
4      if( k == x.size() ) {                                    // if it is a leaf
5          best_v = max( best_v, value(v,x));
6          return;
7      }
8                                                              // if it is not a leaf
9      for (unsigned j = 0; j < 2; j++ ) {
10         x[k]=j;
11         double present_w = acc_w + x[k] * w[k];               // update weight
12         if ( present_w <= W )                                // if it is feasible
13             knapsack(v, w, W, k+1, x, present_w, best_v);          // expand
14     }
15 }
16 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
17     vector<unsigned> x;
18     double best_v = numeric_limits<double>::lowest();
19     knapsack( v, w, W, 0, x, 0, best_v);
20     return best_v;
21 }
```

- **Peor caso** Todos los objetos caben: $O(n\,2^n)$
- **Mejor caso** Ningún objeto cabe: $\Omega(n)$

¿Se puede mejorar?

- El valor se puede ir calculando a medida que se rellena la solución

Calculando el valor de forma incremental

```
1  void knapsack(
2      const vector<double> &v, const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x, double acc_w, double acc_v, double &best_v
4  ){
5      if( k == x.size() ) {                                    // if it is a leaf
6          best_v = max( best_v, acc_v);
7          return;
8      }
9                                                              // it is not a leaf
10     for (unsigned j = 0; j < 2; j++ ) {
11         x[k]=j;
12         double present_w = acc_w + x[k] * w[k];               // update weight
13         double present_v = acc_v + x[k] * v[k];                // update value
14         if ( present_w <= W )                                // if it is feasible
15             knapsack( v, w, W, k+1, x, present_w, present_v, best_v);
16     }
17 }
18 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
19     vector<unsigned> x(v.size());
20     double best_v = numeric_limits<double>::lowest();
21     knapsack( v, w, W, 0, x, 0, 0, best_v );
22     return best_v;
23 }
```

¿Podemos acelerar el programa?

- Sí, evitando explorar ramas que no pueden dar soluciones mejores que la que ya tenemos.
- Por ejemplo:
  - hemos construido una solución hasta el elemento k:

$$(x_1, x_2, \ldots, x_k, ?, \ldots, ?)$$

  - ya hemos encontrado una solución factible de valor v_b
  - si tenemos que:

$$\sum_{i=1}^{k} x_i v_i + \sum_{i=k+1}^{n} v_i \leq v_b$$

$\Rightarrow$ Ninguna expansión de esta solución puede dar un valor mayor que v_b

función para añadir el resto

```
1 double add_rest( const vector<double> &v, size_t k ){
2     double v = 0.0;
3     for( size_t i = k; i < v.size(); i++) r += v[i];
4     return r;
5 }
```

poda: cota optimista ingenua

```
1  void knapsack(
2      const vector<double> &v, const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x, double acc_w, double acc_v, double &best_v
4  ){
5      if( k == x.size() ) {                                  // if it is a leaf
6          best_v = max(best_v,acc_v);
7          return;
8      }
9      for (unsigned j = 0; j < 2; j++ ) {                    // it is not a leaf
10         x[k]=j;
11         double present_w = acc_w + x[k] * w[k];              // update weight
12         double present_v = acc_v + x[k] * v[k];               // update value
13         if( present_w <= W &&                               // if is feasible ...
14             present_v + add_rest(v, k+1) > best_v      // ... and is promising
15         )
16             knapsack(v, w, W, k+1, x, present_w, present_v, best_v);
17     }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
20     vector<unsigned> x(v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, 0, 0, best_v );
23     return best_v;
24 }
```

Podas mas ajustadas

- Interesa que los mecanismos de poda "actúen" lo antes posible
- Una poda mas ajustada se puede obtener usando la solución voraz al problema de la mochila continua
- la solución al problema de la mochila continua es siempre mayor que la solución al problema de la mochila discreto

$\Rightarrow$ El mejor valor que se puede obtener para una solución incompleta:

$$(x_1, x_2, \ldots, x_k, ?, \ldots, ?)$$

será menor que:

$$\sum_{i=1}^{k} x_i v_i + \texttt{knapsack}_c \left(\{x_{k+1}, \ldots, x_n\}, W - \sum_{i=1}^{k} x_i w_i \right)$$

donde $\texttt{knapsack}_c(X, P)$ es la solución de la mochila continua

Poda optimista basada en la mochila continua

```
1  void knapsack(
2      const vector<double> &v, const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x, double acc_w, double acc_v, double &best_v
4  ){
5      if( k == x.size() ) {                                  // if it is a leaf
6          best_v = max( acc_v, best_v);
7          return;
8      }
9      for (unsigned j = 0; j < 2; j++ ) {                   // it is not a leaf
10         x[k]=j;
11         double present_w = acc_w + x[k] * w[k];             // update weight
12         double present_v = acc_v + x[k] * v[k];              // update value
13         if( present_w <= W &&                              // if it is promising
14             present_v + knapsack_c( v, w, k+1, W - present_w) > best_v
15         )
16             knapsack( v, w, W, k+1, x, present_w, present_v, best_v);
17     }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
20     vector<unsigned> x(v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, 0, 0, best_v );
23     return best_v;
24 }
```

Partiendo de una solución cuasi–óptima

- La efectividad de la poda también puede aumentarse partiendo de una solución factible muy "buena"
- Una posibilidad es usar la solución voraz para la mochila discreta (`knapsack_d`) de la siguiente forma:

Solución óptima partiendo de un subóptimo.

```
1 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
2     vector<unsigned> x(v.size());
3     double best_v = knapsack_d(v,w,W);
4     knapsack(v, w, W, 0, x, best_v );
5     return best_v;
6 }
```

Análisis empírico

- 25 muestras aleatorias de tamaño $n = 30$

Tipo de poda	**Partiendo de un subóptimo voraz**	**Llamadas recursivas realizadas (promedio)**	**Tiempo medio (segundos)**
Ninguna	–	1054.8×10^6	875.65
Completando con todos los objetos restantes	No	925.5×10^3	0.112
	Si	389.0×10^3	0.072
Completando según la sol. voraz mochila continua	No	2.3×10^3	0.034
	Si	18	0.002

Otras mejoras

También puede ser relevante:

- El orden en el que se explora los objetos
  - i.e.: ordenándolos por valor especifico
- La forma en la que se "despliega el árbol":
  - i.e.: completar la tupla primero con los unos y después con los ceros

Cambiando el orden de exploración de los objetos

```
1  double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2      vector<size_t> idx( v.size() );                          // index vector
3      iota( begin(idx), end(idx), 0 );
4
5      sort( begin(idx), end(idx),
6          [&v,&w]( size_t i, size_t j) {
7              return v[i]/w[i] > v[j]/w[j];
8          }
9      );
10
11     vector<double> s_v( v.size() ), s_w( w.size() );
12
13     for( size_t i = 0; i < v.size(); i++ ) {
14         s_v[i] = v[ idx[i] ];                                 // sorted values
15         s_w[i] = w[ idx[i] ];                                // sorted weights
16     }
17
18   vector<short> x(v.size());
19   double best_val = knapsack_d( s_v, s_w, 0, W );  // simplified version
20             // we can use a simplified version of knapsack_c in knapsack
21   knapsack( s_v, s_w, W, 0, x, 0, 0, best_val);
22
23   return best_val;
24 }
```

Cambiando el orden de explorar las decisiones (expansión)

```
1  void knapsack( const vector<double> &v, const vector<double> &w,
2                 double W, unsigned k, vector<short> &x,
3                 double weight, double value, double &best_val ){
4
5    if( k == x.size() ) {  // base case
6      best_val = value;
7      return;
8    }
9
10   for (int j = 1; j >= 0; j-- ) {  // <== Reversing the order
11
12     x[k]=j;
13     double new_weight = weight + x[k] * w[k]; // updating weight
14     double new_value  = value  + x[k] * v[k]; // updating value
15
16     if( new_weight <= W &&                          // is promising
17       new_value + knapsack_c( v, w, k+1, W - new_weight ) // simplified version
18         > best_val
19     )
20       knapsack( v, w, W, k+1, x, new_weight, new_value, best_val);
21   }
22 }
```

Índice

Vuelta atrás: definición y ámbito de aplicación

- Algunos problemas sólo se pueden resolver mediante el estudio exhaustivo del conjunto de posibles soluciones al problema
- De entre todas ellas, se podrá seleccionar un subconjunto o bien, aquella que consideremos la mejor (la solución óptima)
- *Vuelta atrás* proporciona una forma sistemática de generar todas las posibles soluciones a un problema
- Generalmente se emplea en la resolución de problemas de selección u optimización en los que el conjunto de soluciones posibles es finito
- En los que se pretende encontrar una o varias soluciones que sean:
  - Factibles: que satisfagan unas restricciones y/o
  - Óptimas: optimicen una cierta función objetivo

Vuelta atrás: características

- Se trata de un recorrido sobre una estructura arbórea imaginaria
- La solución debe poder expresarse mediante una tupla de decisiones: $(x_1, x_2, \ldots, x_n) \quad x_i \in D_i$
  - Las decisiones pueden pertenecer a dominios diferentes entre sí pero estos dominios siempre serán discretos o discretizables
  - El número de soluciones posibles ha de ser finito.
- La estrategia puede proporcionar:
  - una solución factible
  - todas las soluciones factibles
  - la solución óptima al problema
  - las n mejores soluciones factibles al problema
- En la mayoría de los casos las complejidades son prohibitivas

Vuelta atrás: Eficiencia

- Cálculos incrementales
- Podas para evitar la exploración completa del espacio de soluciones:
  - Podar ramas que llevan soluciones no factibles
  - Podar ramas que solo llevan soluciones malas (optimización)
    - uso de cotas optimistas
    - inicialización con cotas pesimistas
- Cambios en el orden de exploración/expansión para conseguir rápidamente soluciones casi óptimas

Podas

- Cota optimista:
  - estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo
  - puede que no haya ninguna solución factible que alcance ese valor
  - normalmente se obtienen relajando las restricciones del problema
  - si la cota optimista de un nodo es peor que la solución en curso, se puede podar el nodo
- Cota pesimista:
  - estima, a peor, el mejor valor que podría alcanzarse al expandir el nodo
  - ha de asegurar que existe una solución factible con un valor mejor que la cota
  - normalmente se obtienen mediante soluciones voraces del problema
  - se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista
  - permite la poda aún antes de haber encontrado una solución factible
- Cuanto mas ajustadas sean las cotas, mas podas se producirán

Ejemplo de cotas optimista

Problema de la mochila discreta:

- Restricciones:
  - hay un peso límite W
  - los objetos no se pueden partir
- Relajaciones:
  - no hay límite $\Rightarrow$ añadir todos los objetos que quedan
  - podemos partir los objetos $\Rightarrow$ problema de la mochila continua

Vuelta atrás: Esquema general recursivo

Esquema recursivo de *backtracking* (optimización)

```
1  void backtracking( node n, solution& present_best ){
2
3      if ( is_leaf(n) ) {
4          if( is_best( solution(n), present_best ) )
5              present_best = solution(n);
6          return;
7      }
8
9      for( node a : expand(n) )
10         if( is_feasible(a) && is_promising(a) )
11             backtracking( a, present_best );
12
13     return;
14 }
15
16 solution backtracking( problem P ){
17     solution present_best = feasible_solution(n);
18     backtracking( initial_node(P), present_best);
19     return present_best;
20 }
```

Vuelta atrás: Estadísticas

```
1  void backtracking( node n, solution& present_best ){
2
3      if ( is_leaf(n) ) {
4          visited_leaf_nodes++;
5          if( is_best( solution(n), present_best ) )
6              present_best = solution(n);
7          return;
8      }
9
10     for( node a : expand(n) )
11         visited_nodes++;
12         if( is_feasible(a) ) {
13             if( is_promising(a) ) {
14                 explored_nodes++;
15                 backtracking( a, present_best );
16             } else
17                 no_promising_discarded_nodes++;
18         } else
19             no_feasible_discarded_nodes++;
20
21     return;
22 }
```

Índice

Índice

Permutaciones

Dado un entero positivo n, escribir un algoritmo que muestre todas las permutaciones de la secuencia $(0, \ldots, n-1)$

- Solución:
  - sea $X = (x_0, x_1, \ldots, x_{n-1}) \quad x_i \in \{0, 1, \ldots, n-1\}$
  - cada permutación será cada una de las reordenaciones de X
  - restricción: X no puede tener elementos repetidos
  - no hay función objetivo: se buscan todas las combinaciones factibles

Ejemplo:
Generar todas las permutaciones de $(0, 1, 2, 3)$:

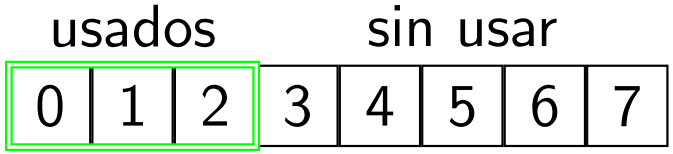
$$(0,1,2,3),(0,1,3,2),(0,2,1,3),(0,2,3,1),(0,3,1,2),(0,3,2,1),$$
$$(1,0,2,3),(1,0,3,2),(1,2,0,3),(1,2,3,0),(1,3,0,2),(1,3,2,1),$$
$$(2,0,1,3),(2,0,3,1),(2,1,0,3),(2,1,3,0),(2,3,0,1),(2,3,1,0),$$
$$(3,0,1,2),(3,0,2,1),(3,1,0,2),(3,1,2,0),(3,2,0,1),(3,2,1,0)$$

```
1  bool is_used( vector<size_t> &x, size_t k, size_t e ) {
2      for( size_t i = 0; i < k; i++ )
3          if( x[i] == e )
4              return true;
5      return false;
6  }
7
8  void permutations( size_t k, vector<size_t> &x ) {
9      if( k == x.size() ) {
10         cout << x << endl;   // Should be overloaded
11         return;
12     }
13
14     for( size_t c = 0; c < x.size(); c++ )
15         if( !is_used( x, k, c ) ) {
16             x[k] = c;
17             permutations( k+1, x );
18         }
19 }
20
21 void permutations( size_t k) {
22     vector<size_t> x(k);
23     permutations( 0, x );
24 }
```

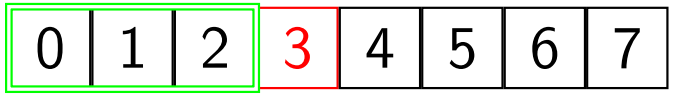
```
1  void permutations( size_t k, vector<size_t> &x, vector<bool>& is_used ){
2      if( k == x.size() ) {
3          cout << x << endl;  // Should be overloaded
4          return;
5      }
6
7      for( size_t c = 0; c < x.size(); c++ ) {
8
9          if( !is_used[c] ) {
10             x[k] = c;
11             is_used[c] = true;
12             permutations( k+1, x, is_used );
13             is_used[c] = false;
14         }
15
16     }
17 }
18
19 void permutations( size_t k ) {
20     vector<bool> is_used(k, false);
21     vector<size_t> x(k);
22     permutations( 0, x, is_used );
23 }
```

Permutaciones usando `swap`

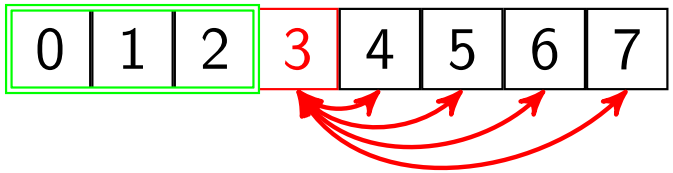
- se separa los elementos ya usados (izquierda) de los sin usar (derecha)



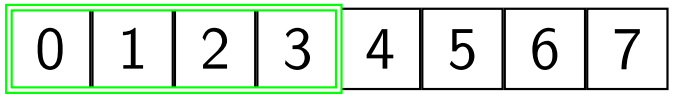
- el pivote es el elemento mas a la izquierda de los no usados

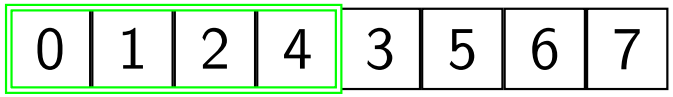


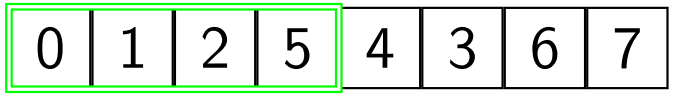
- se va intercambiando el pivote con cada uno de los no usados

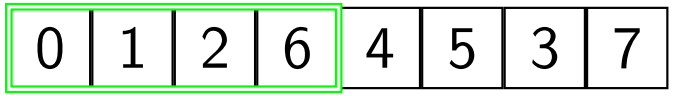


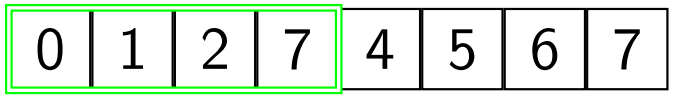
- para cada uno de los intercambios se integra el pivote en los usados











Restricción (con swap)

```
1  void permutations( size_t k, vector<size_t> &x ) {
2
3      if( k == x.size() ) {
4          cout << x << endl;  // Should be overloaded
5          return;
6      }
7
8      for( size_t c = k; c < x.size(); c++ ) {
9          swap(x[k],x[c]);
10         permutations( k+1, x );
11         swap(x[k],x[c]);
12     }
13 }
14
15 void permutations( size_t k ) {
16     vector<size_t> x(k);
17
18     for( size_t i = 0; i < k; i++ )
19         x[i] = i;
20
21     permutations( 0, x );
22 }
```

Índice

El viajante de comercio

Dado un grafo ponderado $g = (V, E)$ con pesos no negativos, el problema consiste en encontrar un *ciclo hamiltoniano* de mínimo coste

- Un *ciclo hamiltoniano* es un recorrido en el grafo que recorre todos los vértices sólo una vez y regresa al de partida
- El coste de un ciclo viene dado por la suma de los pesos de las aristas que lo componen

El viajante de comercio

- Expresamos la solución mediante una tupla $X = (x_1, x_2, \ldots, x_n)$ donde $x_i \in \{1, 2, \ldots, n\}$ es el vértice visitado en i-ésimo lugar
  - Asumimos que los vértices están numerados, $V = \{1, 2, \ldots, n\}, \quad n = |V|$
  - Fijamos el vértice de partida (para evitar rotaciones):
    - $x_1 = 1;\ x_i \in \{2, 3, \ldots, n\} \quad \forall i : 2 \leq i \leq n$
- Restricciones
  - No se puede visitar dos veces el mismo vértice: $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
  - Existencia de arista: $\forall i : 1 \leq i < n$, $\texttt{weight}(g, x_i, x_{i+1}) \neq \infty$
  - Existencia de arista que cierra el camino: $\texttt{weight}(g, x_n, x_1) \neq \infty$
- Función objetivo:

$$\min \sum_{i=1}^{n-1} \texttt{weight}(g, x_i, x_{i+1}) + \texttt{weight}(g, x_1, x_n)$$

El viajante de comercio

```
1  unsigned round( const graph &g, const vector<size_t> &x ) {
2      size_t d = 0;
3      for( size_t i = 0; i < x.size() - 1; i++ )
4          d += g.dist( x[i], x[i+1]);
5      d += g.dist( x[x.size()-1], x[0] );
6      return d;
7  }
8
9  void solve(
10     const graph &g, size_t k, vector<size_t> &x,
11     unsigned &shortest
12 ) {
13     if( k == x.size() ) {
14         shortest = min( shortest, round(g,x));
15         return;
16     }
17
18     for( size_t c = k; c < x.size(); c++ ) {
19         swap( x[k], x[c] );
20         solve( g, k+1, x, shortest );
21         swap( x[k], x[c] );
22     }
23 }
```

El viajante de comercio (con swap)

```
1  void solve(
2      const graph &g, size_t k, vector<size_t> &x,
3      unsigned &shortest
4  ) {
5      if( k == x.size() ) {
6          shortest = min( shortest, round(g,x));
7          return;
8      }
9
10     for( size_t c = k; c < x.size(); c++ ) {
11         swap( x[k], x[c] );
12         solve( g, k+1, x, shortest );
13         swap( x[k], x[c] );
14     }
15 }
16
17 size_t solve( const graph &g ) {
18     size_t shortest = numeric_limits<unsigned>::max();
19     vector<size_t> x(g.num_cities());
20     for( size_t i = 0; i < x.size(); i++ )
21         x[i] = i;
22     solve( g, 1, x, shortest ); // fix the first city
23     return shortest;
24 }
```

El viajante de comercio

- La propuesta es inviable por su prohibitiva complejidad: $O(n^n)$

Mejoras:

- Cálculo incremental de la vuelta
- Cota optimista:
  - Restricciones:
    - el camino ha de pasar por todas las ciudades
    - el camino ha de ser continuo
  - Relajaciones:
    - no pasar por todas las ciudades $\Rightarrow$ saltar de la última a la primera
    - ir saltando de una ciudad a otra $\Rightarrow$ árbol de recubrimiento mínimo
- Poda basada en la mejor solución hasta el momento (cota pesimista)
  - buscar una solución subóptima
    - algoritmo voraz

Las n mejores soluciones

```
1  void solve_nbest(
2      const graph &g,
3      size_t k,
4      vector<size_t> &x,
5      priority_queue<size_t> &pq,
6      size_t n
7  ){
8      if( k == x.size() ) {
9          size_t len = round(g,x);
10         if( pq.top() > len ) {   // access to the higher element
11             if( pq.size() == n )
12                 pq.pop();        // extract the higher element
13             pq.push(len);
14         }
15         return;
16     }
17
18     for( size_t c = k; c < x.size(); c++ ) {
19         swap(x[k],x[c]);
20         solve_nbest( g, k+1, x, pq, n );
21         swap(x[k],x[c]);  // not needed
22     }
23 }
```

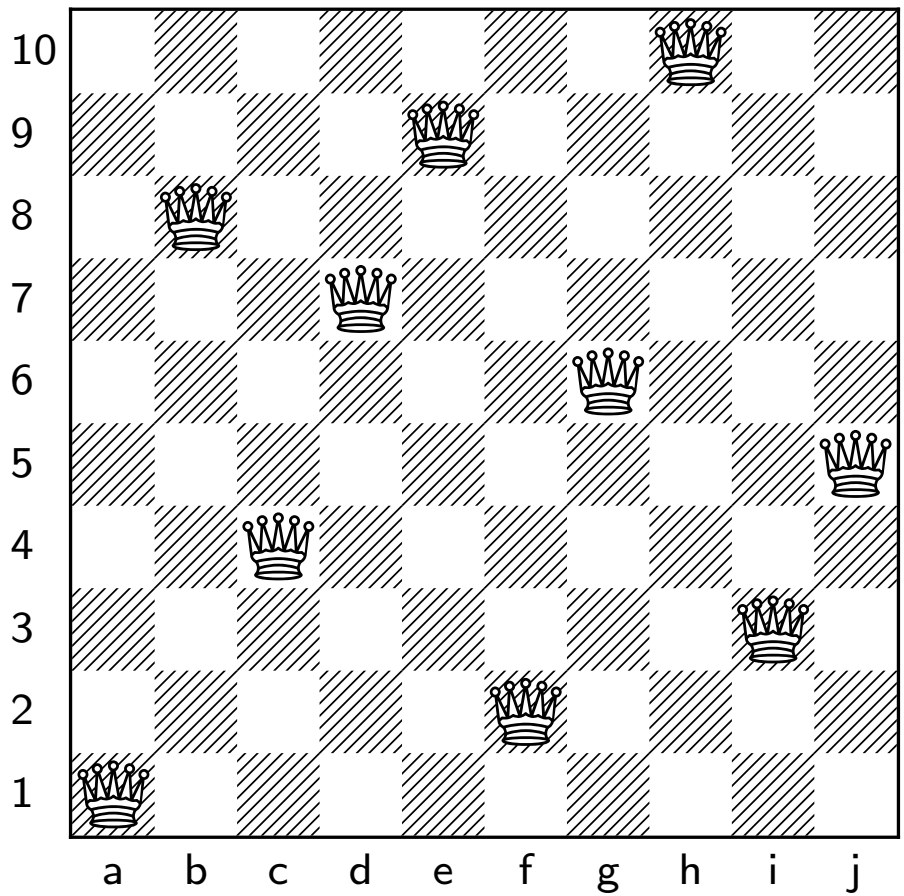
Las *n* mejores soluciones

```
1  priority_queue<unsigned> solve_nbest(
2      const graph &g,
3      size_t n
4  ){
5      vector<size_t> x(g.num_cities());
6      for( size_t i = 0; i < x.size(); i++ )
7          x[i] = i;
8
9      priority_queue<size_t> pq;
10     pq.push(numeric_limits<size_t>::max());
11
12     solve_nbest( g, 1, x, pq, n );
13
14     return pq;
15 }
```

Índice

El problema de las n reinas

- En un tablero de "ajedrez" de $n \times n$ obtener todas las formas de colocar n reinas de forma que no se ataquen mutuamente (no estén ni en la misma fila, ni columna, ni diagonal).



El problema de las n reinas

Solución:

- No puede haber dos reinas en la misma fila,
  - la reina i se colocará en la fila i.
  - El problema es determinar en qué columna se colocará.
- Sea $X = (x_1, x_2, \ldots, x_n)$,
  - $x_i \in \{1, 2, \ldots, n\}$: columna en la que se coloca la reina de la fila i
- Restricciones:
  1. No puede haber dos reinas en la misma fila:
     - implícito en la forma de representar la solución.
  2. No puede haber dos reinas en la misma columna
     - X no puede tener elementos repetidos.
  3. No puede haber dos reinas en la misma diagonal:

     $\Rightarrow \; |i - j| \neq |x_i - x_j|$

El problema de las *n* reinas (escribir todas las soluciones)

```
1  bool feasible( vector<size_t> &x, size_t k ) {
2      for( size_t i = 0; i < k; i++ ) {
3          if( x[i] == x[k] ) return false;
4          if( x[i] < x[k] && x[k] - x[i] == k - i ) return false;
5          if( x[i] > x[k] && x[i] - x[k] == k - i ) return false;
6      }
7      return true;
8  }
9
10 void n_queens( size_t k, vector<size_t> &x ) {
11     if( k == x.size() ) {
12         cout << x << endl;  // Should be overloaded
13         return;
14     }
15     for( size_t i = 0; i < x.size(); i++ ) {
16         x[k] = i;
17         if( feasible(x, k) ) n_queens( k+1, x );
18     }
19 }
20
21 void n_queens( size_t n ) {
22     vector<size_t> x(n);
23     n_queens(0,x);
24 }
```

El problema de las *n* reinas (escribir sólo una solución)

```
1  void n_queens( size_t k, vector<size_t> &x, bool &found ) {
2
3      if( k == x.size() ) {
4          cout << x << endl;  // Should be overloaded
5          found = true;
6          return;
7      }
8
9      for( size_t i = 0; i < x.size(); i++ ) {
10         x[k] = i;
11         if( factible(x, k) ) {
12             n_queens( k+1, x, found );
13             if( found ) return;
14         }
15     }
16
17 }
18
19 void n_queens( size_t n ) {
20     vector<size_t> x(n);
21     bool found = false;
22
23     n_queens( 0, x, found );
24 }
```

El problema de las *n* reinas (devolver sólo una solución)

```
1  void n_queens(
2      size_t k, vector<size_t> &x, vector<size_t> &sol, bool &found
3  ) {
4      if( k == x.size() ) {
5          sol = x;
6          found = true;
7          return;
8      }
9      for( size_t i = 0; i < x.size(); i++ ) {
10         x[k] = i;
11         if( factible(x, k) ) {
12             n_queens( k+1, x, sol, found );
13             if( found ) return;
14         }
15     }
16 }
17
18 vector<size_t> n_queens( size_t n ) {
19     vector<size_t> x(n);
20     vector<size_t> sol(n);
21     bool found = false;
22     n_queens( 0, x, sol, found);
23     return sol;
24 };
```

Índice

La función compuesta mínima

Dadas dos funciones $f(x)$ y $g(x)$ y dados dos números cualesquiera x e y, encontrar la función compuesta mínima que obtiene el valor y a partir de x tras aplicaciones sucesivas e indistintas de $f(x)$ y $g(x)$

- Ejemplo: Sean $f(x) = 3x$, $g(x) = \lfloor x/2 \rfloor$, y sean $x = 3$, $y = 6$
  - Una transformación de 3 en 6 con operaciones f y g es:

$$(g \circ f \circ g \circ f \circ f \circ g)(3) = 6 \qquad \text{(5 composiciones)}$$

  - La mínima (aunque no única) es:

$$(f \circ g \circ g \circ f)(3) = 6 \qquad \text{(3 composiciones)}$$

La función compuesta mínima

Solución:

- $X = (x_1, x_2, \ldots, x_?) \quad x_i \in \{0, 1\} \begin{cases} 0 \equiv & \text{se aplica } f(x) \\ 1 \equiv & \text{se aplica } g(x) \end{cases}$
  - $(x_1, x_2, \ldots, x_k) \equiv (x_k \circ \cdots \circ x_2 \circ x_1)$
  - El tamaño de la tupla no se conoce a priori
    - se pretende minimizar el tamaño de la tupla solución (función objetivo)
    - asumiremos un máximo de M composiciones (evitar ramas infinitas)
- Llamamos $F(X, k, x)$ al resultado de aplicar al valor x la composición representada en la tupla X hasta su posición k

$$F(X, k, x) = (x_k \circ \cdots \circ x_2 \circ x_1)(x) = x_k(\ldots x_2(x_1(x))\ldots)$$

- Restricciones:
  - $F(X, k, x) \neq F(X, i, x)\ \forall i < k$, para evitar recálculos
  - $k < M$, para evitar búsquedas infinitas
  - siempre se puede calcular $F(X, k, x)$
  - $k < v_b$, (v_b es la mejor solución actual) tupla "prometedora"

La función compuesta mínima (sin poda)

```
1  const size_t NOT_FOUND = numeric_limits<size_t>::max();
2
3  int F( const vector<short> &x, unsigned k, int init) {
4      for( unsigned i = 0; i < k; i++ )
5          if( x[i] == 0 ) init = f(init);
6          else            init = g(init);
7      return init;
8  }
9  void composition( size_t k, vector<short> &x, size_t M,
10     int first, int last, size_t &best
11 ) {
12     if( F(x, k, first) == last && k < best ) best = k; // update the best
13     if( k == M ) return;                               // limit bound
14
15     for( short i = 0; i < 2; i++ ) {
16         x[k] = i;
17         composition(k+1, x, M, first, last, best);
18     }
19 }
20 size_t composition(size_t M, int first, int last) {
21     vector<short> x(M);
22     size_t best = NOT_FOUND;
23     composition( 0, x, M, first, last, best);
24     return best;
25 }
```

La función compuesta mínima (incremental)

```
1  int F1( unsigned i, int init) {
2      if( i == 0 ) return f(init);
3      else         return g(init);
4  }
5
6  void composition( unsigned k, unsigned M,
7      int present, int last, size_t &best
8  ) {
9      if( present == last && k < best ) best = k; // update the best
10
11     if( k == M ) return;                        // limit bound
12
13     for( short i = 0; i < 2; i++ ) {
14         int next = F1(i,present);
15         composition(k+1, M, next, last, best);
16     }
17 }
18
19 size_t composition(unsigned M, int first, int last) {
20     size_t best = NOT_FOUND;
21     composition( 0, M, first, last, best);
22
23     return best;
24 }
```

La función compuesta mínima (incremental con poda)

```
1  void composition( unsigned k, unsigned M,
2      int present, int last, size_t &best
3  ) {
4      if( present == last ){    // update the best
5          best = k;
6          return;
7      }
8
9      if( k >= best ) return;  // best solution in progess bound
10
11     if( k == M ) return;     // limit bound
12
13     for( short i = 0; i < 2; i++ ) {
14         int next = F1(i,present);
15         composition(k+1, M, next, last, best);
16     }
17 }
18
19 size_t composition(unsigned M, int first, int last) {
20     size_t best = NOT_FOUND;
21     composition( 0, M, first, last, best);
22
23     return best;
24 }
```

La función compuesta mínima (con poda y memoria)

```
1  void composition( unsigned k, unsigned M,
2      int present, int last,
3      unordered_map<int, Default<size_t, NOT_FOUND>> &best // steps to the index
4  ) {
5      if( best[present] <= k ) return; // update the best
6      best[present] = k;
7
8      if( k >= best[last] ) return;   // best solution in progress bound
9
10     if( k == M ) return;            // limit bound
11
12     for( short i = 0; i < 2; i++ ) {
13         int next = F1(i,present);
14         composition(k+1, M, next, last, best);
15     }
16 }
17
18 size_t composition(unsigned M, int first, int last) {
19     unordered_map<int, Default<size_t, NOT_FOUND>> best;
20     composition( 0, M, first, last, best);
21     return best[last];
22 }
```

Utilidad:

Cambio del valor por defecto del operador []

```
1  template<typename T, T X>
2  class Default {
3  public:
4      Default () : val(T(X)) {}
5      Default (T const & _val) : val(_val) {}
6      operator T & () { return val; }
7      operator T const & () const { return val; }
8  private:
9      T val;
10 };
```

Potencia de las podas

Número de llamadas recursivas para encontrar el mínimo número de composiciones (12) para llegar de 1 a 11

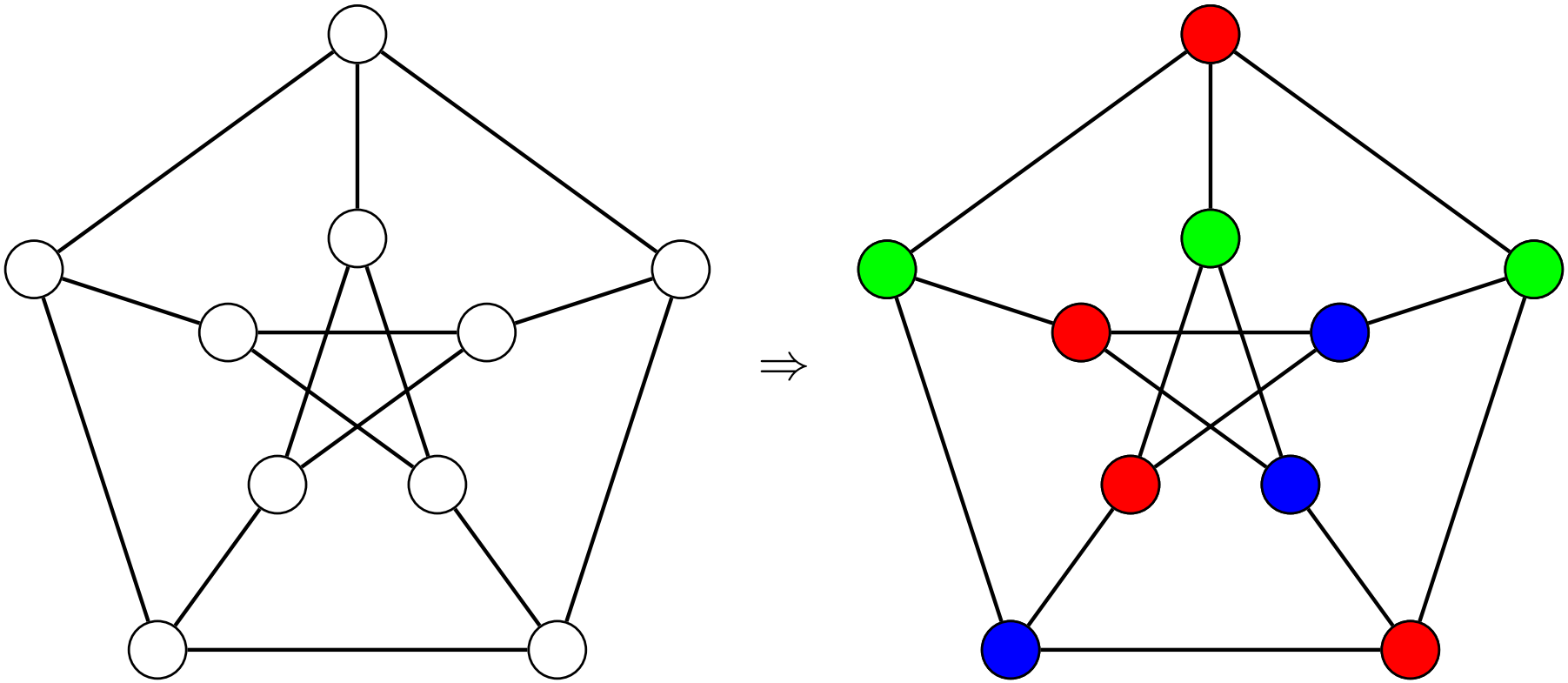
Tipo poda	$M = 15$	$M = 20$	$M = 25$
Básico	65 535	2 097 151	67 108 863
mejor en curso	9 075	40 323	1 040 259
memorización	1 055	7 243	50 669
las dos	565	2 541	16 469

Sería mejor hacer una búsqueda por niveles...

Índice

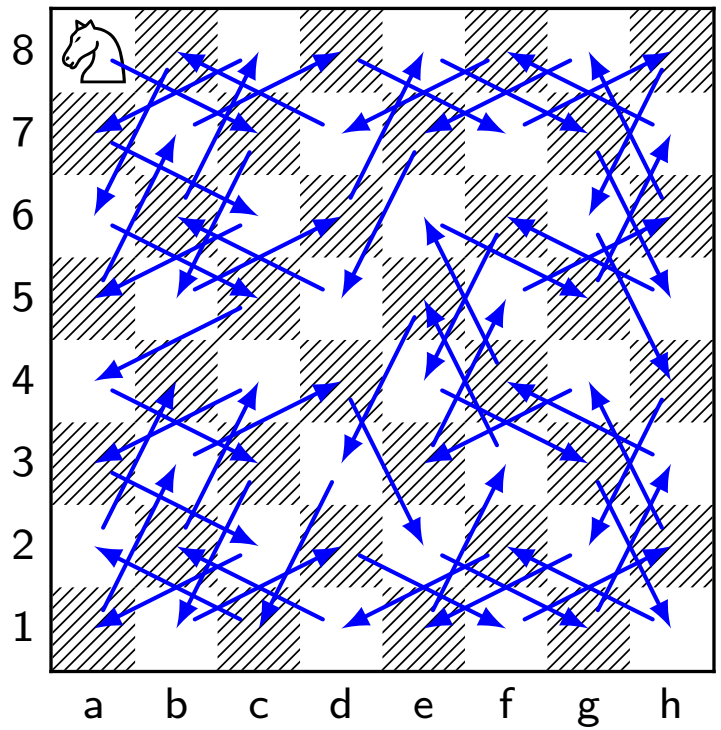
El coloreado de grafos

- Dado un grafo G, encontrar el menor número de colores con el que se pueden colorear sus vértices de forma que no haya dos vértices adyacentes con el mismo color



El recorrido del caballo de ajedrez

- Encontrar una secuencia de movimientos "legales" de un caballo de ajedrez de forma que éste pueda visitar las 64 casillas de un tablero sin repetir ninguna



El laberinto con cuatro movimientos

- Se dispone de una cuadrícula $n \times m$ de valores $\{0, 1\}$ que representa un laberinto. Un valor 0 en una casilla cualquiera de la cuadrícula indica una posición inaccesible; por el contrario, con el valor 1 se simbolizan las casillas accesibles.
- Encontrar un camino que permita ir de la posición $(1, 1)$ a la posición $(n \times m)$ con cuatro tipos de movimiento (arriba, abajo, derecha, izquierda)

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i.
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} b_{ij}$$

- Encontrar una asignación óptima, es decir, de mínimo coste

La empresa naviera

- Supongamos una empresa naviera que dispone de una flota de N buques cada uno de los cuales transporta mercancías de un valor v_i que tardan en descargarse un tiempo t_i. Solo hay un muelle de descarga y su máximo tiempo de utilización es T
- Diseñar un algoritmo que determine el orden de descarga de los buques de forma que el valor descargado sea máximo sin sobrepasar el tiempo de descarga T. (Si se elige un buque para descargarlo, es necesario que se descargue en su totalidad)

La asignación de turnos

- Estamos al comienzo del curso y los alumnos deben distribuirse en turnos de prácticas
- Para solucionar este problema se propone que valoren los turnos de práctica disponibles a los que desean ir en función de sus preferencias
- El número de alumnos es N y el de turnos disponibles es T
- Se dispone una matriz de preferencias P, $N \times T$, en la que cada alumno escribe, en su fila correspondiente, un número entero (entre 0 y M) que indica la preferencia del alumno por cada turno (0 indica la imposibilidad de asistir a ese turno; M indica máxima preferencia)
- Se dispone también de un vector C con T elementos que contiene la capacidad máxima de alumnos en cada turno
- Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia sin exceder la capacidad de los turnos

Sudoku

- El famoso juego del Sudoku consiste en rellenar una rejilla de 9×9 celdas dispuestas en 9 subgrupos de 3×3 celdas, con números del 1 al 9, atendiendo a la restricción de que no se debe repetir el mismo número en la misma fila, columna o subgrupo 3×3
- Además, varias celdas disponen de un valor inicial

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku sin resolver

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Sudoku resuelto

Análisis y diseño de algoritmos

7. Ramificación y Poda

José Luis Verdú Mas, Jose Oncina, Víctor Sánchez

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de marzo de 2024

Índice

Índice

Ejemplo Introductorio

El problema de la mochila (general)

Dados:

- n objetos con valores v_i y pesos w_i
- una mochila que solo aguanta un peso máximo W

Seleccionar un conjunto de objetos de forma que:

- no se sobrepase el peso límite W (restricción)
- el valor transportado sea máximo (función objetivo)

Formalización

- Solución: $X = (x_1, x_2, \ldots, x_n) \quad x_i \in \{0, 1\}$
- Restricciones:
  - Implícitas:

$$x_i \in \begin{cases} 0 & \text{no se selecciona el objeto } i \\ 1 & \text{se selecciona el objeto } i \end{cases}$$

  - Explícitas:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

- Función objetivo:

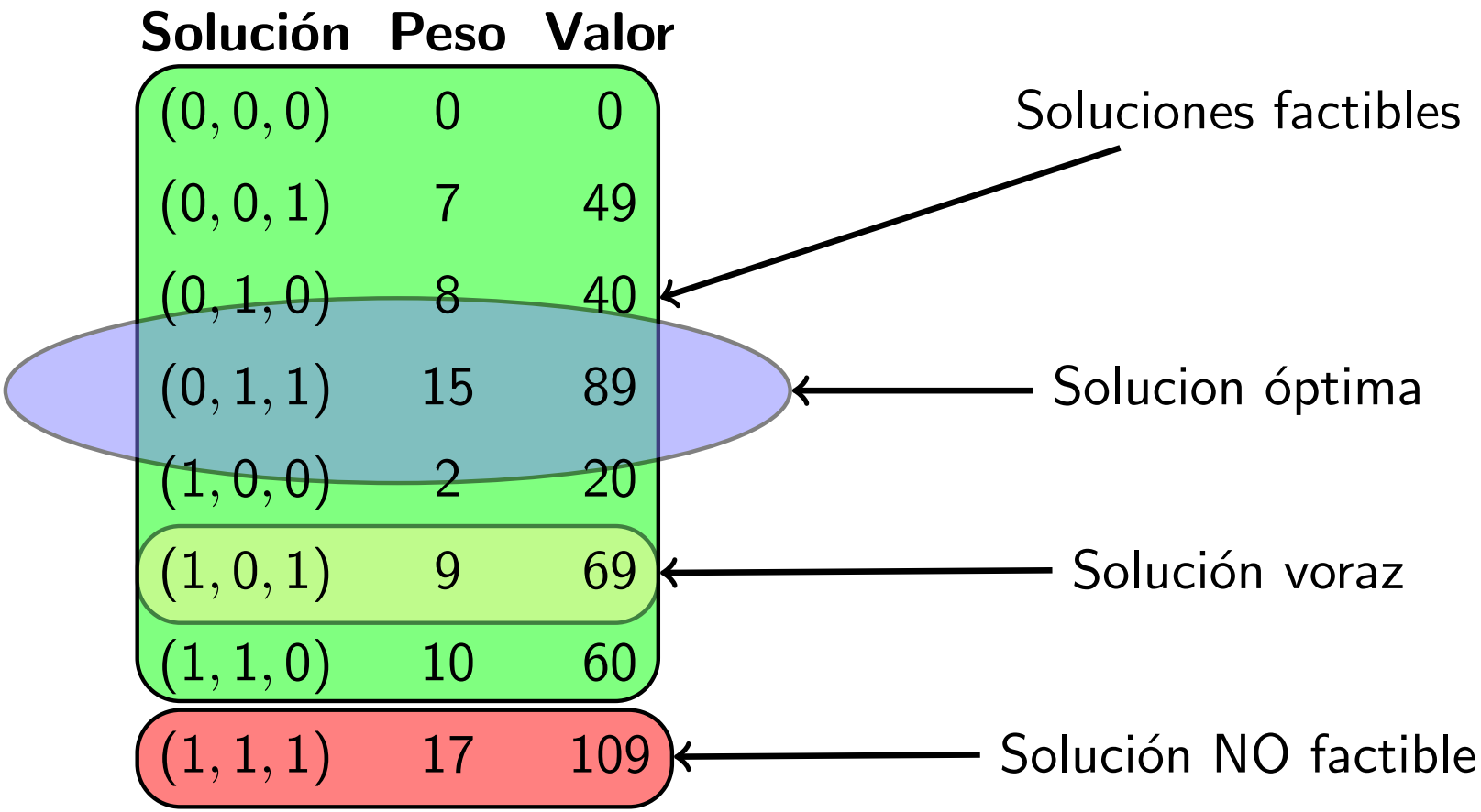
$$\max \sum_{i=1}^{n} x_i v_i$$

Tipos de soluciones

- Supongamos el siguiente ejemplo:

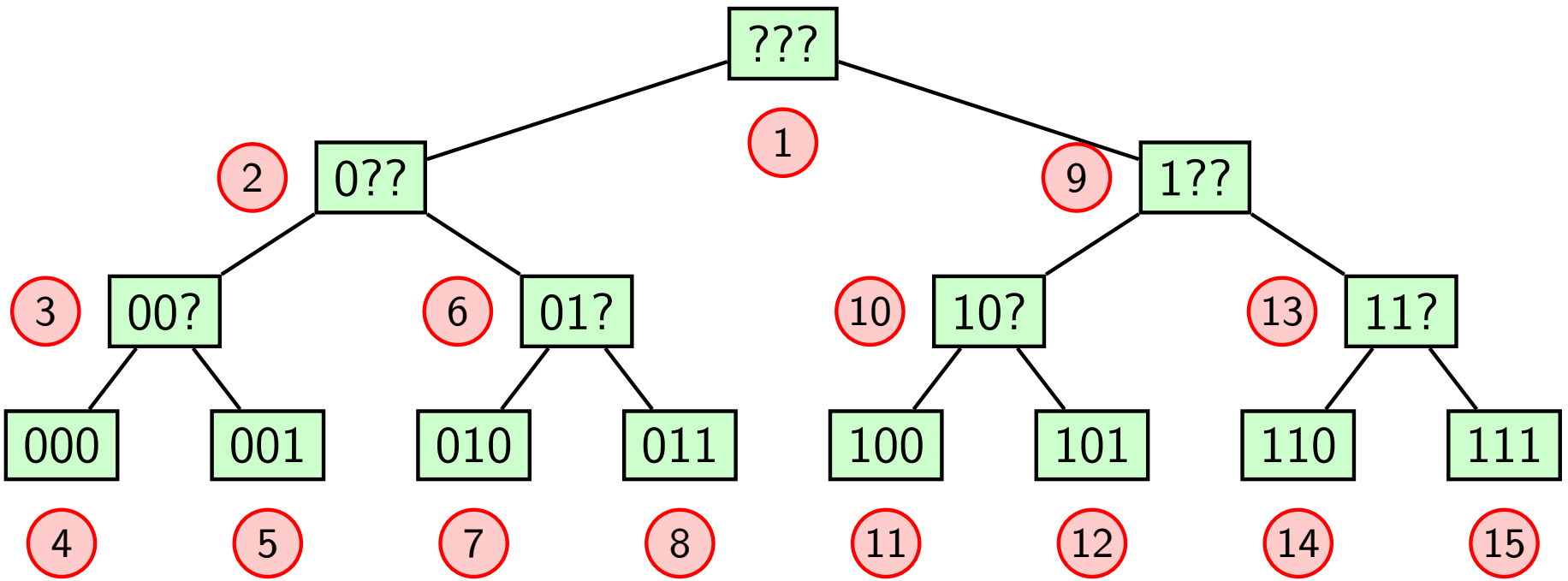
$$W = 16 \qquad w = (2, 8, 7) \qquad v = (20, 40, 49)$$

- Combinaciones posibles (espacio de soluciones):



Solución usando vuelta atrás

- Combinaciones posibles:
  - Supongamos el ejemplo: $W = 16 \quad w = (7, 8, 2) \quad v = (49, 40, 20)$
  - Espacio de soluciones
    - Generación ordenada mediante vuelta atrás
    - Nodos generados: 15
    - Nodos expandidos: 7



¿Es el recorrido importante?

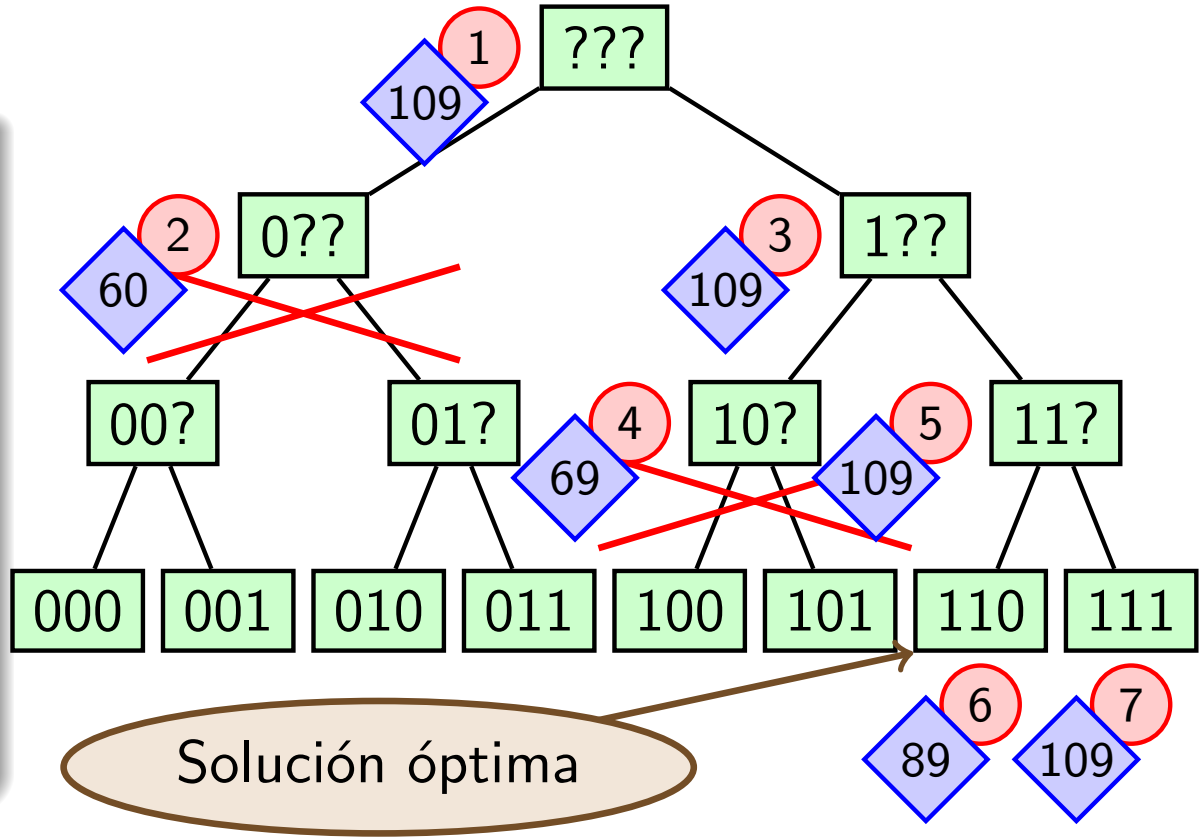
- ¿Podríamos llegar antes a la solución óptima con otro recorrido?
- ¿Cómo?
  - Adecuando el orden de exploración del árbol de soluciones según nuestros intereses. Se priorizará para su exploración aquellos nodo mas prometedores
- ... sin olvidar las podas

Ejemplo Introductorio

- Combinaciones posibles:
  - Supongamos el ejemplo: $W = 16 \quad w = (7, 8, 2) \quad v = (49, 40, 20)$

Cota optimista

El valor que resultaría de incluir en la solución aquellos objetos pendientes de tratar (sustituir en cada nodo los '?' por '1'), independientemente de que quepan o no.

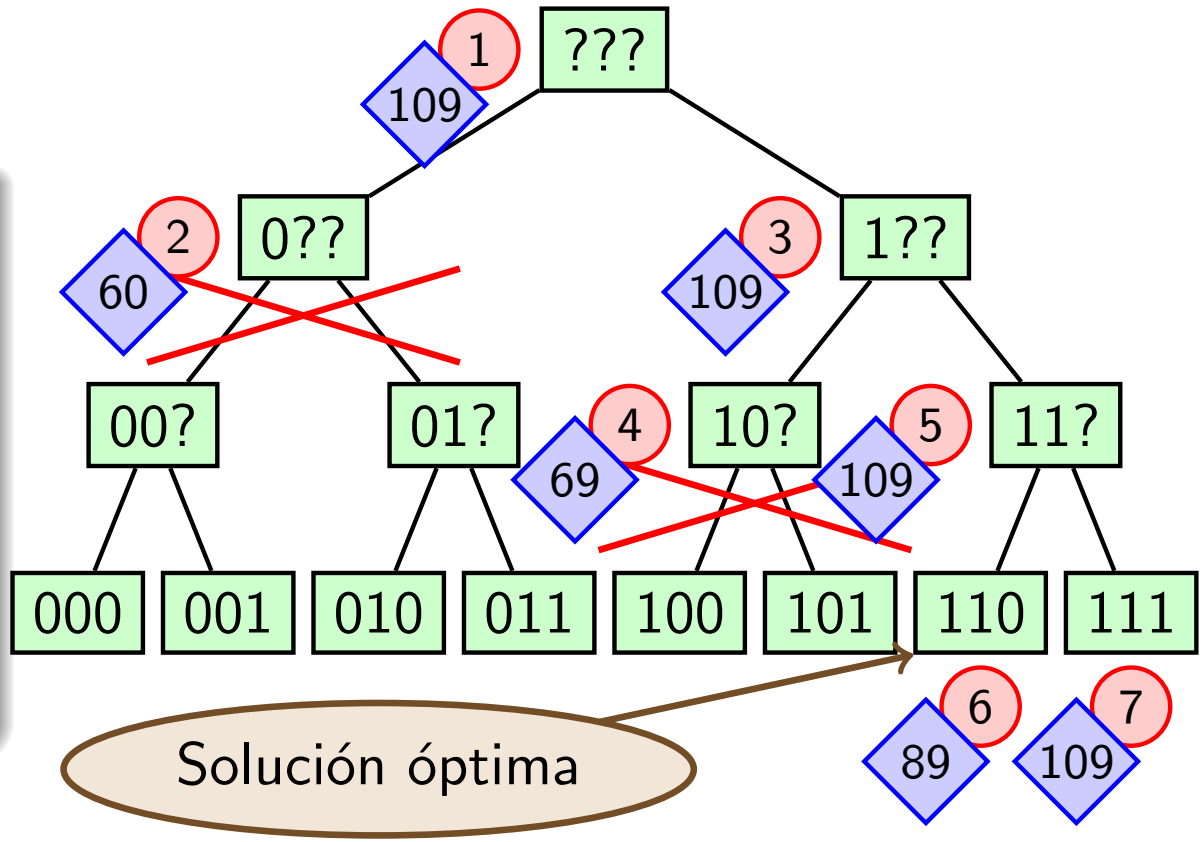


Ejemplo Introductorio

- Combinaciones posibles:
  - Supongamos el ejemplo: $W = 16 \quad w = (7, 8, 2) \quad v = (49, 40, 20)$

Espacio de soluciones

- Orden de expansión priorizando los nodos con mayor cota
- Generados: 7
- Expandidos: 3
- Reducción $\geqslant 50\,\%$



Implementación

```
1  float knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2
3  //    typedef vector<short> Sol;
4  //    typedef tuple<double, double, Sol, int> Node;
5      using Sol = vector<short>;
6      using Node = tuple<double, double, Sol, int>;  // acc_v, acc_w, vector x, k
7      priority_queue< Node > pq;                     // A priority_queue is a max-heap
8
9      double best_v = knapsack_d( v, w, 0, W); // updating best current solution
10     pq.emplace( 0.0, 0.0, Sol(v.size()), 0 );   // insert initial node (c++17)
11
12     while( !pq.empty() ) {
13
14         auto [acc_v, acc_w, x, k] = pq.top();       // structured auto (c++17)
15         pq.pop();
16
17         /* ... next slide ...*/
18
19     }
20
21     return best_v;
22 }
```

Dentro del bucle ...

```
1  /* ... */
2      if( k == v.size() ) {                              // base case
3          best_v = max( acc_v, best_v ) ;
4          continue;
5      }
6
7      for (unsigned j = 0; j < 2; j++ ) {              // expanding
8          x[k] = j;
9
10         double present_w = acc_w + x[k] * w[k]; // updating weight
11         double present_v = acc_v + x[k] * v[k];    // updating value
12
13         if( present_w <= W &&                        // is feasible & is promising
14             present_v + knapsack_c( v, w, k + 1, W - present_w ) > best_v
15         )
16             pq.emplace( present_v, present_w, x, k + 1 );
17     }
18 /* ... */
```

Puede mejorarse?

Sí, añadiendo cotas pesimistas al expandir el nodo

- La cota pesimista permite mejorar la mejor solución en curso
- El mejorar la mejor solución en curso permite hacer mas podas
- Esto no era útil en vuelta atrás

Usando podas pesimistas

```
1  /* ... */
2      if( k == v.size() ) {                                  // base case
3          best_v = max( acc_v, best_v ) ;
4          continue;
5      }
6
7      for (unsigned j = 0; j < 2; j++ ) {
8          x[k] = j;
9
10         double present_w = acc_w + x[k] * w[k];            // updating weight
11         double present_v = acc_v + x[k] * v[k];            // updating value
12
13         if( present_w <= W ) {                             // is feasible
14             // pessimistic bound
15             double pes_bound = present_v + knapsack_d( v, w, k+1, W-present_w);
16             best_v = max( best_v, pes_bound);
17
18             double opt_bound = present_v + knapsack_c(v, w, k+1, W-present_w);
19             if( opt_bound > best_v )                          // is promising
20                 pq.emplace( present_v, present_w, x, k+1 );
21         }
22     }
23 /* ... */
```

Ordenando por cota optimista

```
1  double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2    using Sol = vector<short>;
3    //                opt_bound,  acc_v, acc_w,   x,   k
4    using Node = tuple< double, double, double, Sol, unsigned >;
5    priority_queue< node > pq;              // Remind: priority queue is a max-heap
6
7    double best_v = knapsack_d( v, w, 0, W);
8    double opt_bound = knapsack_c(v, w, 0, W);
9
10   pq.emplace( opt_bound, 0.0, 0.0, sol(v.size()), 0 );
11
12   while( !pq.empty() ) {
13
14     auto [ignore, acc_v, acc_w, x, k] = pq.top();
15     pq.pop();
16
17     /* ... Next slide ... */
18
19   }
20   return best_v;
21 }
```

Dentro del bucle

```
1  /* ... */
2      if( k == v.size() ) {  //  base case
3          best_v = max( best_v, acc_v ) ;
4          continue;
5      }
6
7      for (unsigned j = 0; j < 2; j++ ) {
8          x[k] = j;
9
10         double present_w = acc_w + x[k] * w[k]; // updating weight
11         double present_v = acc_v + x[k] * v[k]; // updating value
12
13         if( present_w <= W ) {
14
15             double pes_bound = present_v + knapsack_d( v, w, k+1, W-present_w);
16             best_v = max( best_v, pes_bound);
17
18             double opt_bound = present_v + knapsack_c( v, w, k+1, W-present_w);
19             if( opt_bound > best_v)  // is promising
20                 pq.emplace( opt_bound, present_v, present_w, x, k+1 );
21         }
22     }
23 /* ... */
```

Comparación

Promedio del número de iteraciones para 100 instancias aleatorias del problema de la mochila con 100 objetos.

	optimista	inicializando	pesimista
Vuelta Atrás	4 491	277	253
RyP (por valor)	2 406	229	197
RyP (por cota optimista)	206	122	112

- En cuanto a RyP, en este caso se han comparado dos estrategias de búsqueda:
  1. priorizar los nodos con mayor valor;
  2. priorizar los nodos con mayor cota optimista.

Índice

Definición y ámbito de aplicación

- Variante del diseño de Vuelta Atrás
  - Realiza una enumeración parcial del espacio de soluciones mediante la generación de un árbol de expansión
  - Uso de cotas para podar ramas que no son de interés
- Nodo vivo: aquel con posibilidades de ser ramificado (visitado pero no completamente expandido)
- Los nodos vivos se almacenan en estructuras que faciliten su recorrido y eficiencia de la búsqueda:

  - En profundidad (estrategia LIFO) $\Rightarrow$ pila
  - En anchura (estrategia FIFO) $\Rightarrow$ cola
  - Dirigida (primero el mas prometedor) $\Rightarrow$ cola de prioridad

Definición y ámbito de aplicación

- Funcionamiento de un algoritmo de ramificación y poda
- Etapas
  - Partimos del nodo inicial del árbol
  - Se asigna una solución pesimista (subóptima, soluciones voraces)
  - Selección
    - Extracción del nodo a expandir del conjunto de nodos vivos
    - La elección depende de la estrategia empleada
    - Se actualiza la mejor solución con las nuevas soluciones encontradas
  - Ramificación
    - Se expande el nodo seleccionado en la etapa anterior dando lugar al conjunto de sus nodos hijos
  - Poda
    - Se eliminan (podan) nodos que no contribuyen a la solución
    - El resto de nodos se añaden al conjunto de nodos vivos
  - El algoritmo finaliza cuando se agota el conjunto de nodos vivos

Proceso de poda

- Cota optimista:
  - estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo
  - puede que no haya ninguna solución factible que alcance ese valor
  - normalmente se obtienen relajando las restricciones del problema
  - si la cota optimista de un nodo es peor que la solución en curso, se puede podar el nodo
  - suele ser una solución no factible
- Cota pesimista:
  - estima, a peor, el mejor valor que podría alcanzarse al expandir el nodo
  - ha de asegurar que existe una solución factible con un valor mejor que la cota
  - normalmente se obtienen mediante soluciones voraces del problema
  - se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista
  - permite la poda aún antes de haber encontrado una solución factible
  - suele ser una solución factible
- Cuanto mas ajustadas sean las cotas, mas podas se producirán

Esquema de Ramificación y Poda

```
1  solution branch_and_bound( problem p ) {
2
3      node initial = initial_node(p);                    // supossed feasible
4      solution current_best = pessimistic_solution(initial);   // pessimistic
5      priority_queue<Node> q.push(initial);
6
7      while( ! q.empty() ) {
8          node n = q.top();
9          q.pop();
10
11         if( is_leaf(n) ) {
12             if( is_better( solution(n), current_best) )
13                 current_best = solution(n);
14             continue;
15         }
16
17         for( node a : expand(n) )
18             if( is_feasible(a) && is_promising( a, current_best))
19                 q.push(a);
20     }
21
22     return current_best;
23 }
```

Esquema de Ramificación y Poda

- Funciones:
  - `initial_node(p)`: obtiene el nodo inicial para la expansión
  - `pesimistic_solution(n)`: devuelve una solución aproximada (factible pero no la óptima)
  - `is_leaf(n)`: mira si `n` es una posible solución
  - `solution(n)` devuelve el valor del nodo `n`
  - `expand(n)`: devuelve la expansión de `n`
  - `is_feasible(n)`: comprueba si `n` cumple las restricciones
  - `is_promising( n, current_best )`: mira si a partir del nodo `n` se pueden obtener soluciones mejores que `current_best` (normalmente se obtiene mediante una solución optimista)

Podando con cotas pesimistas

```
1  solution branch_and_bound( problem p ) {
2      node initial = initial_node(p);                    // supossed feasible
3      solution current_best = pessimistic_solution(initial);  // pessimistic
4      priority_queue<Node> q.push(initial);
5
6      while( ! q.empty() ) {
7          node n = q.top();
8          q.pop();
9
10         if( is_leaf(n) ) {
11             if( is_better( solution(n), current_best) )
12                 current_best = solution(n);
13             continue;
14         }
15         for( node a : expand(n) )
16             if( is_feasible(a) ) {
17                 if( is_better( pessimistic_solution(n), current_best ) )
18                     current_best = pessimistic_solution(n);
19                 if( is_promising (a, current_best ))
20                     q.push(a);
21             }
22     }
23     return current_best;
24   }
```

Estadísticas (I)

```
1  solution branch_and_bound( problem p ) {
2      node initial = initial_node(p);
3      solution current_best = pessimistic_solution(initial);
4      priority_queue<Node> q.push(initial);
5
6      while( ! q.empty() ) {
7          node n = q.top();
8          q.pop();
9
10         if( !is_promissing(n,current_best) ){
11             discarded_promissing_nodes++;
12             continue;
13         }
14
15 /* ... next slide ... */
16
17     return current_best;
18   }
```

Estadísticas (II)

```
1  ...
2          if( is_leaf(n) ) {
3              completed_nodes++;
4              if( is_better( solution(n), current_best) ) {
5                  current_best_updates_from_completed_nodes++;
6                  current_best = solution(n);
7              }
8              continue;
9          }
10        expanded_nodes++;
11         for( node a : expand(n) ) {
12             visited_nodes++;
13             if( is_feasible(a) ) {
14                 if( is_better( pessimistic_solution(n), current_best ) ) {
15                     current_best_updates_from_pessimistic_bounds++;
16                     current_best = pessimistic_solution(n);
17                 }
18                 if( is_promising (a, current_best )) {
19                     explored_nodes++;
20                     q.push(a);
21                 } else no_promissing_discarded_nodes++;
22             } else no_feasible_discarded_nodes++;
23         }
24 ...
```

Características

- La estrategia puede proporcionar:
  - Todas las soluciones factibles
  - Una solución al problema
  - La solución óptima al problema
  - Las n mejores soluciones
- Objetivo de esta técnica
  - Mejorar la eficiencia en la exploración del espacio de soluciones
- Desventajas/Necesidades
  - Encontrar una buena cota optimista (problema relajado)
  - Encontrar una buena solución pesimista (estrategias voraces)
  - Encontrar una buena estrategia de exploración (cómo ordenar)
  - Mayor requerimiento de memoria que los algoritmos de Vuelta Atrás
  - las complejidades en el peor caso suelen ser muy altas
- Ventajas
  - Suelen ser más rápidos que Vuelta Atrás

Esquema de Ramificación y Poda

- Muchas veces, para ver si un nodo es prometedor, se hace comparando la mejor solución obtenida con una solución optimista de nodo.

```
1 bool is_promising( const node &n, const solution &current_best) {
2   return is_better( optimistic_solution(n), current_best );
3 }
```

se pueden hacer podas agresivas cambiándola por:

```
1 bool is_promising( const node &n, const solution &current_best) {
2   return is_significantly_better(optimistic_solution(n), current_best);
3 }
```

¡Cuidado! puede que se pierda la solución óptima

Índice

Índice

El viajante de comercio

Dado un grafo ponderado $g = (V, A)$ con pesos no negativos, el problema consiste en encontrar un *ciclo hamiltoniano* de mínimo coste.

- Un *ciclo hamiltoniano* es un recorrido en el grafo que recorre todos los vértices sólo una vez y regresa al de partida.
- El coste de un ciclo viene dado por la suma de los pesos de las aristas que lo componen.

El viajante de comercio

- Expresamos la solución mediante una tupla $X = (x_1, x_2, \ldots, x_n)$ donde $x_i \in \{1, 2, \ldots, n\}$ es el vértice visitado en i-ésimo lugar.
  - Asumimos que los vértices están numerados, $V = \{1, 2, \ldots, n\}, \quad n = |V|$
  - Fijamos el vértice de partida (para evitar rotaciones):
    - $x_1 = 1;\ x_i \in \{2, 3, \ldots, n\} \quad \forall i : 2 \leq i \leq n$
- Restricciones
  - No se puede visitar dos veces el mismo vértice: $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
  - Existencia de arista: $\forall i : 1 \leq i < n$, $\texttt{weight}(g, x_i, x_{i+1}) \neq \infty$
  - Existencia de arista que cierra el camino: $\texttt{weight}(g, x_n, x_1) \neq \infty$
- Función objetivo:

$$\min \sum_{i=1}^{n-1} \texttt{weight}(g, x_i, x_{i+1}) + \texttt{weight}(g, x_1, x_n)$$

El viajante de comercio

```
1  unsigned travelling_salesman( const graph &g) {
2      struct Node {
3          unsigned opt_bound, length;
4          vector<short> x;
5          unsigned k;
6      };
7      struct is_worse {
8          bool operator() (const Node& a, const Node& b) {
9              return a.opt_bound > b.opt_bound;
10         }
11     };
12     priority_queue< Node, vector<Node>, is_worse > pq;
13     vector<short> x(g.num_cities());
14     for( unsigned i = 0; i < g.num_cities(); i++ )  x[i] = i;
15     unsigned shortest = pessimistic_bound( g, x, 1);
16     unsigned opt_bound = optimistic_bound( g, x, 1);
17     pq.emplace( opt_bound, 0, x, 1 ); // c++20!
18     while( !pq.empty() ) {
19       Node n = pq.top();
20       pq.pop();
21     /* ... Next slide ... */
22     }
23     return shortest;
24 }
```

Dentro del bucle ...

```
1  /* ... */
2      if( n.k == g.num_cities() ) {
3        shortest = min( shortest, n.length + g.dist(n.x[n.k-1],n.x[0]) );
4        continue;
5      }
6      for( unsigned c = n.k; c < n.x.size(); c++ ) {
7        swap( n.x[n.k], n.x[c] );
8
9        unsigned new_length = n.length + g.dist(n.x[n.k-1],n.x[n.k]);
10       unsigned opt_bound = new_length + optimistic_bound(g,n.x,n.k+1);
11       unsigned pes_bound = new_length + pessimistic_bound(g,n.x,n.k+1);
12
13       shortest = min( shortest, pes_bound);
14
15       if( opt_bound <= shortest )
16         pq.emplace( opt_bound, new_length, n.x, n.k+1 );
17
18       swap( n.x[n.k], n.x[c] );
19     }
20 /* ... */
```

Cambiando la prioridad ...

Por cota optimista

```
1 struct is_worse {
2     bool operator() (const node& a, const node& b) {
3         return a.opt_bound > b.opt_bound;
4     }
5 };
```

Por distancia recorrida

```
1 struct is_worse {
2     bool operator() (const node& a, const node& b) {
3         return a.length > b.length;
4     }
5 };
```

Por distancia media recorrida

```
1 struct is_worse {
2     bool operator() (const node& a, const node& b) {
3         return a.length/a.k > b.length/b.k;
4     }
5 };
```

Algunos ejemplos de ejecución

Promedio del número de iteraciones necesitado para resolver 100 instancias del problema del viajante de comercio con 15 ciudades

- **Cota optimista**: minumum spanning tree de las ciudades restantes
- **Cota pesimista**: algoritmo voraz basado en la ciudad más cercana

Algoritmo	**cota opt.**	**dist. recorrida**	**dist. media**
Vuelta atrás	23 478	23 478	23 478
Ramificación y poda	10 798	12 285	11 421
factor de aceleración	2.17	1.91	2.05

Índice

La función compuesta mínima

Dadas dos funciones $f(x)$ y $g(x)$ y dados dos números cualesquiera x e y, encontrar la función compuesta mínima que obtiene el valor y a partir de x tras aplicaciones sucesivas e indistintas de $f(x)$ y $g(x)$

- Ejemplo: Sean $f(x) = 3x$, $g(x) = \lfloor x/2 \rfloor$, y sean $x = 3$, $y = 6$
  - Una transformación de 3 en 6 con operaciones f y g es:

$$(g \circ f \circ g \circ f \circ f \circ g)(3) = 6 \qquad \text{(5 composiciones)}$$

  - La mínima (aunque no única) es:

$$(f \circ g \circ g \circ f)(3) = 6 \qquad \text{(3 composiciones)}$$

La función compuesta mínima

Solución:

- $X = (x_1, x_2, \ldots, x_?) \quad x_i \in \{0, 1\} \begin{cases} 0 \equiv & \text{se aplica } f(x) \\ 1 \equiv & \text{se aplica } g(x) \end{cases}$
  - $(x_1, x_2, \ldots, x_k) \equiv (x_k \circ \cdots \circ x_2 \circ x_1)$
  - El tamaño de la tupla no se conoce a priori
    - Se pretende minimizar el tamaño de la tupla solución (función objetivo)
    - asumiremos un máximo de M composiciones (evitar ramas infinitas)
- Llamanmos $F(X, k, x)$ al resultado de aplicar al valor x la composición representada en la tupla X hasta su posición k

$$F(X, k, x) = (x_k \circ \cdots \circ x_2 \circ x_1)(x) = x_k(\ldots x_2(x_1(x))\ldots)$$

- Restricciones:
  - $F(X, k, x) \neq F(X, i, x)\ \forall i < k$, para recalculos
  - $k < M$, para evitar búsquedas infinitas
  - siempre se puede calcular $F(X, k, x)$
  - $k < v_b$, tupla "prometedora"

Composición de funciones

```
1  const size_t NOT_FOUND = numeric_limits<size_t>::max();
2
3  int composition( unsigned M, int first, int last ) {
4      using Node = tuple< short, int>; // steps, present
5
6      struct is_worse {
7          bool operator() (const Node& a, const Node& b) {
8              return get<0>(a) > get<0>(b);
9          }
10     };
11     priority_queue< Node, vector<Node>, is_worse > pq;
12
13     unsigned best = NOT_FOUND;
14     pq.emplace( 0, first );
15
16     while( !pq.empty() ) {
17         Node n = pq.top();
18         pq.pop();
19
20     /* ... next slide ... */
21
22     }
23     return best;
24 }
```

Dentro del bucle ... (sin podas)

```
1  /*...*/
2      unsigned k = get<0>(n);
3      int present = get<1>(n);
4
5      if( present == last && k < best )  // select the best
6          best = k;
7
8      if( k ==  M )                      //  limit bound
9          continue;
10
11     for( short i = 0; i < 2; i++ ) {
12         int next = F1( i, present );
13         pq.emplace( k+1, next );
14     }
15 /*...*/
```

Dentro del bucle ... (poda: mejor en curso)

poda: mejor en curso

```
1  /*...*/
2      unsigned k = get<0>(n);
3      int present = get<1>(n);
4
5      if( present == last && k < best ) { // select the best
6          best = k;
7          continue;
8      }
9
10     if( k >= best )          // best solution in progress bound
11         continue;
12
13     if( k ==  M )            // limit bound
14         continue;
15
16     for( short i = 0; i < 2; i++ ) {
17         int next = F1( i, present );
18         pq.emplace( k+1, next );
19     }
20 /*...*/
```

Podando con memoria

```
1  int composition( unsigned M, int first, int last ) {
2
3      using Node = tuple< short, int>; // steps, hit
4
5      struct is_worse {
6          bool operator() (const Node& a, const Node& b) {
7              return get<0>(a) > get<0>(b);
8          }
9      };
10     priority_queue< Node, vector<Node>, is_worse > pq;
11     unordered_map<int, Default<unsigned, NOT_FOUND>> best;
12
13     pq.emplace( 0, first );
14     while( !pq.empty() ) {
15         node n = pq.top();
16         pq.pop();
17
18     /* ... next slide ... */
19
20     }
21     return best[last];
22 }
```

Dentro del bucle ...

```
1  /* ... */
2      unsigned k = get<0>(n);
3      int present = get<1>(n);
4
5      if( best[present] <= k )  // update the best
6          continue;
7      best[present] = k;
8
9      if( k >= best[last] )   // (extended) best solution in progress bound
10         continue;
11
12     if( k == M )            // limit bound
13         continue;
14
15     for( short i = 0; i < 2; i++ ) {
16         int next = F1( i, present );
17         pq.emplace( k+1, next );
18     }
19 /* ... */
```

Ejemplo de ejecución

Número de iteraciones para alcanzar el valor 11 desde 1 (con $M = 20$)

Algoritmo	**Vuelta Atrás**	**Ramificación y Poda**
básico	65 535	65 535
mejor en curso	9 073	8 311
memorizando	565	329

Índice

El Puzzle

- Disponemos de un tablero con n^2 casillas y de $n^2 - 1$ piezas numeradas del uno al $n^2 - 1$. Dada una ordenación inicial de todas las piezas en el tablero, queda sólo una casilla vacía (con valor 0), a la que denominamos *hueco*.
- El objetivo del juego es transformar dicha disposición inicial de las piezas en una disposición final ordenada, en donde en la casilla (i, j) se encuentra la pieza numerada $n(i-1)+j$ y en la casilla (n, n) se encuentra el hueco
- Los únicos movimientos permitidos son los de las piezas adyacentes al hueco (horizontal y verticalmente), que pueden ocuparlo. Al hacerlo, dejan el hueco en la posición en donde se encontraba la pieza antes del movimiento. Otra forma de abordar el problema es considerar que lo que se mueve es el hueco, pudiendo hacerlo hacia arriba, abajo, izquierda o derecha. Al moverse, su casilla es ocupada por la pieza que ocupaba la casilla a donde se ha *movido* el hueco

El Puzzle

- Por ejemplo, para el caso n = 3 se muestra a continuación una disposición inicial junto con la disposición final:

1	5	2
4	3	
7	8	6

Disposición inicial

1	2	3
4	5	6
7	8	

Disposición final

El Puzzle

- Regla: una pieza se puede mover de A a B si:
  - A está al lado de B
  - B es el hueco

- Según se relaje el problema tenemos dos funciones de coste diferentes:
  1. Calcular el número de piezas que están en una posición distinta de la que les corresponde en la disposición final
  2. Calcular la suma de las distancias de Manhattan desde la posición de cada pieza a su posición en la disposición final
- La distancia de Manhattan entre dos puntos del plano de coordenadas (x_1, y_1) y (x_2, y_2) viene dada por la expresión:

$$|x_1 - x_2| + |y_1 - y_2|$$