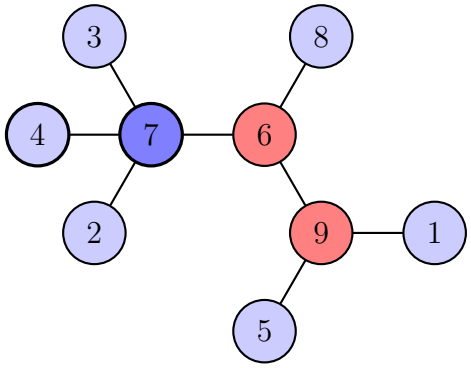
Problemas y Algoritmos



9 7 7 7 9 6 6

Por Luis E. Vargas Azcona
Algunas imagenes por Roberto López

Acuerdo de Licencia

Prefacio

El propósito general de este libro, es el de introducir al lector en la resolución de problemas de programación así como en el diseño de algoritmos.

Si bien se presentan algoritmos, el objetivo no es saturar el conocimiento del lector con una gran cantidad de algoritmos sino mostar sus fundamentos, mostrar también maneras inteligentes de usarlos para resolver problemas, y sobre todo, lograr que el lector sea capaz de diseñar sus propios algoritmos.

Muchos libros de algoritmos se limitan a explicar algoritmos sin detenerse en su fundamento matemático y sin decir las aplicaciones que tiene en solución de problemas.

Algunos otros si explican el fundamento de los algoritmos, pero resultan inadecuados para estudiantes con poco conocimiento matemático ya que suelen dar por hecho que el lector ya sabe teoría de conjuntos y matemáticas discretas.

Este libro está dirigido a estudiantes con gusto de programar y resolver problemas pero que todavía no adquieren las bases matemáticas necesarias para poder leer libros de algoritmos con suficiente fundamento matemático; por lo que se pretende, no solamente explicar los algoritmos, sino también darle al lector las herramientas necesarias para entender, analizar y diseñar.

Por ello, una gran cantidad de páginas se dedican a establecer las bases matemáticas que sirven como soporte para la resolución de problemas de programación.

Los temas tratados en este libro son indispensables para comprender el gran campo de estudio de la solución de problemas mediante el uso de las computadoras, constituyen los fundamentos de una cantidad interminable de conocimientos y técnicas que estan en constante desarrollo por la investigación en las ciencias de la computación.

En el libro se muestran implementaciones en C++ que no usan memoria dinámica ni estructuras, esto es por los siguientes motivos:

- A veces al tener el pseudocódigo de un algoritmo, no resulta claro cómo implementarlo sin escribir demasiado. Por ello es preferible mostrar algunas implementaciones de ejemplo para que el lector conozca al menos una implementación corta de cada algoritmo.

- Muchos elementos del lenguaje C++ son muy conocidos e incluso estan presentes en otros lenguajes, con una syntaxis casi idéntica a la de C++, por ello, si se utilizan pocos elementos del lenguaje, es posible que un lector que no esté familiarizado con este lenguaje pueda igualmente entender los códigos.

- No está por demás decir que este libro fue escrito pensando en participantes de un concurso llamado olimpiada de informática, en ese concurso los participantes tienen tiempo limitado para implementar y depurar sus programas y las implementaciones son mas fáciles de depurar con memoria estática.

Al Estudiante

Puede parecer extraño encontrar un libro de algoritmos con estas características, ya que dedica muchas páginas a los fundamentos matemáticos y omite varias cosas que varios libros de algoritmos orientados a licenciatura no dejarían pasar por alto, como lo son *quicksort*, *shellsort*, listas circulares, listas doblemente ligadas y tablas de dispersión entre otras cosas.

El motivo de esto es simple: este libro no pretende ser una referencia ni mucho menos un repaso de los temas de licenciatura; pretende aportar las herramientas básicas para aplicar la programación en la resolución de problemas.

Por esto mismo puede llegar a ser un recurso valioso ya que permite abordar la programación desde un punto de vista mas creativo y menos repetitivo, y al mismo tiempo el enfoque a resolver nuevos problemas es la base de la inovación.

Este libro esta hecho para leerse casi como si fuera una novela, un capítulo tras otro, y si se decide saltarse uno o mas capítulos se corre el riesgo de que mas adelante no se pueda seguir bien. La diferencia con una novela es la gran cantidad de problemas de ejemplo, los cuales conviene intentar resolver antes de leer la solución.

Al Instructor

La teoría necesaria para leer este libro de principio a fin es muy poca, incluso un alumno de preparatoría la debería de saber. Sin embargo, muchos problemas que se mencionan, por su dificultad, estan lejos de ser adecuados para cualquier alumno de preparatoria e incluso pueden darle dificultades a graduados de la universidad; pero resultan bastante adecuados para aquellos que buscan algo mas interesante que problemas de rutina o para quienes se preparan para concursos de programación.

El libro está escrito de la manera en la que me gustaría presentar los temas si fuera yo el instructor.

Es decir, procura motivar todos los conceptos antes de abordarlos formalmente, intenta nunca dejar *huecos*, en la teoría; y sobre todo, buscar aplicaciones creativas a cada tema que se aborda evitando decir muy rápido la solución y dando tiempo para especular.

Para resolver un problema, es necesario plantearse un diálogo consigo mismo de preguntas y respuestas. Dicho diálogo también se debe de presentar entre alumno e instructor y en muchas partes del libro se intenta plasmar el diálogo abordando los razonamientos que podrían llevar al lector a resolver cada uno de los ejemplos.

Al Olímpico

Este libro lo escribí pensando principalmente en los olímpicos, por lo que todo lo que se menciona aquí tiene aplicación directa o indirecta con la olimpiada de informática.

La olimpiada esta centrada en resolver problemas mediante la programación y el diseño de algoritmos; para ello se requieren bases matemáticas sólidas y un conocimiento profundo(aunque no necesariamente amplio) de los algoritmos.

Es por eso que este libro dedica muchas páginas a dejar claras las bases matemáticas y a explorar las propiedades de algoritmos y no solo en introducirlos.

Los ejemplos se deben de intentar resolver por cuenta propia para aprender de los propios errores e ir desarrollando poco a poco un método propio para resolver problemas. Si de pronto no logras resolver un ejemplo, puedes empezar a leer la solución para darte una idea y continuar por cuenta propia.

Índice general

Parte I

Recursión

La recursión es uno de los temas mas básicos en el estudio de los algoritmos. De hecho, muchas veces(no siempre) es el primer tema tratado en lo que se refiere a resolver problemas de programación.

Esto de "primer tema" puede parecer un poco confuso, ya que para llegar a recursión es necesario de antemano ya saber programar.

Es primer tema en lo que se refiere a resolver problemas, lo cual no trata de saber cómo escribir una solución en una computadora, trata de saber cómo encontrar soluciones.

El conocimiento de un lenguaje de programación se limita a expresiones lógicas para expresar las ideas en una computadora y eso no es encontrar una solución sino simplemente saber describir la solución.

Podría parecer para algunos una pérdida de tiempo leer acerca de cómo encontrar soluciones a problemas, y pueden también estarse preguntando ¿no basta con ser inteligente para poder encontrar la solución a un problema?.

En teoría es cierto que cualquiera podría llegar a la solución de un problema simplemente entendiendo el problema y poniéndose a pensar. Sin embargo, en la práctica, pretender resolver un problema de cierto nivel sin haber resuelto nunca antes problemas mas fáciles resulta imposible.

En el entendimiento claro y la buena aplicación de la recursión descansan las bases teóricas y prácticas de buena parte(casi me atrevería a decir que de la mayoría) de los algoritmos que se aprenden mas adelante.

No quiero decir con esto que las implementaciones recursivas sean la respuesta para todo ni para la mayoría; pero un razonamiento partiendo de una recursión es con mucha frecuencia utilizado para implementar algoritmos no recursivos.

Capítulo 1

Inducción Matemática

La inducción matemática no es mas que el uso de razonamientos recursivos para comprobar cosas. Aún no hemos definido que quiere decir "recursivo", sin embargo el buen entendimiento de la inducción puede ser un buen escalón para la recursión en general.

También hay que tomar en cuenta que la inducción es un tema muy importante(y desgraciadamente muy subestimado) para el entendimiento correcto de los algoritmos y en algún momento es conveniente tratarlo, por ello comenzaremos con el uso y la definición de la inducción matemática, que es algo bastante simple, para luego generalizarlo en la recursión.

En el momento de resolver problemas, es muy importante observar propiedades, y si no se está seguro si una propiedad es cierta, la inducción puede servir más de lo que se pueda imaginar.

Es importante mencionar que la inducción sirve para probar *proposiciones*, es decir, enunciados que declaran que algo es verdadero o falso(nunca ambas cosas a la vez ni tampoco un término medio).

En todo el libro trabajaremos con proposiciones, ya que es la forma mas simple de pensar y de hacer buenos razonamientos sin tener duda alguna al respecto.

Empezaremos probando un ejemplo sencillo, luego pasaremos a ejemplos mas complejos para posteriormente definir formalmente la inducción matemática.

También, aprovecharemos esta sección para poner como ejemplos algunas propiedades que luego serán muy útiles en el momento de resolver algunos problemas.

Antes de continuar es importante mencionar que en este libro se utilizarán puntos suspensivos(...) en algunas fórmulas.

Estos puntos suspensivos deben de completar la fórmula con el patrón

que se muestra. Por ejemplo:

$1+2+3+...+9+10$ significa $1+2+3+4+5+6+7+8+9+10$

1.1. Ejemplos de Inducción

Ejemplo 1.1.1. Prueba que para todo entero $n \geq 1$

$$1+2+3+4+5+...+n-1+n = \frac{(n)(n+1)}{2}$$

Solución Si analizamos un poco ,la proposición "para todo entero $n \geq 1, 1+2+3+4+5+...+n-1+n = \frac{(n)(n+1)}{2}$", quiere decir que dicha propiedad debe ser cierta para $n=1$, para $n=2$, para $n=3$, para $n=4$, para $n=5$, etc...

Podemos comenzar verificando que es cierto para los primeros enteros positivos:

$$\frac{(1)(1+1)}{2} = 1$$

$$\frac{(2)(2+1)}{2} = 3 = 1+2$$

$$\frac{(3)(3+1)}{2} = 6 = 1+2+3$$

$$...$$

Esta claro que si tratamos de aplicar este procedimiento para todos los números enteros positivos nunca acabaríamos a menos que encontráramos un número en el que no se cumpliera la proposición(lo cual no podemos asegurar).

Aquí muchos se sentirían tentados a pensar *esto es cierto porque funciona en todos los casos que verifiqué*. Esa no es la salida correcta, y no solo es un capricho teórico, ya que en la práctica se pueden cometer errores bastante serios si ingenuamente se cree que verificar varios casos pequeños es suficiente para estar seguro de algo.

Por ejempo, la proposición "x^2+x+41 es primo cuando x es entero positivo" puede resultar engañosa, ya que si verificamos con varios valores de x, el resultado parecerá que siempre es primo

$$\text{para x=1 sucede que } (1)^2+(1)+41 = 43 \quad (1.1)$$
$$\text{para x=2 sucede que } (2)^2+(2)+41 = 47 \quad (1.2)$$
$$\text{para x=3 sucede que } (3)^2+(3)+41 = 53 \quad (1.3)$$
$$\text{para x=4 sucede que } (4)^2+(4)+41 = 61 \quad (1.4)$$

Notamos que 43, 47, 53 y 61 son primos, por tanto esto nos puede tentar a creer que esa expresión produce un primo para cualquier entero positivo x. Pero ahora, examinemos que sucede cuando $x = 40$:

$$(40)^2+40+41 = (40)(40)+40+41 = 40(1+40)+41 = 40(41)+41 = 41^2 \quad (1.5)$$

Es obvio que 41^2 no es primo; por lo tanto la proposición es falsa. En realidad pudimos haber verificado con 39 casos($x = 1, x = 2, ..., x = 39$) y en todos habría funcionado.

Este fue un claro ejemplo de como muchas veces, lo que parece funcionar siempre, no siempre funciona, en casos pequeños puede parecer que siempre funciona, pero al llegar a casos mas grandes(algunos tan grandes que no se pueden verificar manualmente), es inevitable que el error se haga presente.

Incluso hay ocaciones en las que se pueden verificar miles de casos(sin exagerar) sin encontrar un ejemplo donde la proposición no se cumpla y que a pesar de todo eso, la proposición sea falsa.

Otra razón para demostrar las cosas es que la solución completa de un problema(la solución de un problema incluye las demostraciones de todas sus proposiciones) puede servir para descubrir propiedades que sirven para resolver muchos otros problemas. Y es ahí donde resolver un problema realmente sirve de algo.

Volviendo al tema, podemos transformar este problema de comprobar que $1+2+3+4+5+...+n-1+n = \frac{(n)(n+1)}{2}$ en comprobar este conjunto infinito de proposiciones:

$$\frac{(1)(1+1)}{2} = 1, \frac{(2)(2+1)}{2} = 1+2, \frac{(3)(3+1)}{2} = 1+2+3, \frac{(4)(4+1)}{2} = 1+2+3+4, ...$$

Para hacerlo basta con probar las siguientes dos cosas:

- Que la primera proposición es verdadera.
- Que si una proposición es verdadera, la siguiente también lo es.

Probar que la primera proposición es verdadera es realmente fácil:

$$\frac{(1)(1+1)}{2} = \frac{(1)(2)}{2} = \frac{2}{2} = 1$$

Ahora, suponiendo que para alguna n

$$\frac{(n)(n+1)}{2} = 1 + 2 + 3 + 4 + ... + n - 1 + n$$

vamos a probar que:

$$\frac{(n+1)(n+2)}{2} = 1 + 2 + 3 + 4 + ... + n - 1 + n + n + 1$$

Es decir, vamos a probar que si se cumple con alguna n, se debe de cumplir con $n + 1$.

Partimos de la ecuación inicial

$$\frac{(n)(n+1)}{2} = 1 + 2 + 3 + 4 + ... + n - 1 + n$$

Sumamos $n + 1$ en ambos lados de la ecuación

$$\frac{(n)(n+1)}{2} + n + 1 = 1 + 2 + 3 + 4 + ... + n + n + 1$$

$$\frac{(n)(n+1)}{2} + \frac{(2)(n+1)}{2} = 1 + 2 + 3 + 4 + ... + n + n + 1$$

Y sumando por factor común se concluye que:

$$\frac{(n+2)(n+1)}{2} = 1 + 2 + 3 + 4 + ... + n + n + 1$$

La proposición es correcta con 1, y si es correcta con 1 entonces será correcta con 2, si es correcta con 2 lo será con 3, si es correcta con 3 entonces lo será con 4, y así sucesivamente, podemos asegurar que se cumple con todos los números enteros positivos.□

Ejemplo 1.1.2. En una fiesta hay n invitados, se asume que si un invitado conoce a otro, éste último conoce al primero. Prueba que el número de invitados que conocen a un número impar de invitados es par.

Solución Nos encontramos ahora con un problema un poco mas abstracto que el anterior. ¿Cómo tratar este problema?.

Nuevamente imaginaremos el caso más simple que puede haber: una fiesta en la que nadie se conozca.

En este caso, ¿cuántos invitados conocen a un número impar de invitados?, la respuesta es obvia: ninguno, es decir, 0. Recordemos entonces que el 0 es par, si p es un número par $p+1$ es impar, y $p+2$ es par.

Ahora, si imaginamos en la fiesta que dos personas se presentan mutuamente, habrá pues, 2 personas que conocen a un solo invitado y las demás no conocen a nadie. El número de invitados que conocen a un número impar de invitados será 2.

Nuevamente, si otras 2 personas que no se conocen entre sí se presentan mutuamente, entonces el número de invitados que conocen a un número impar de personas aumentará a 4.

Pero después de ello, ¿qué sucedería si una persona que conoce a un solo invitado se presenta con una persona que no conoce a nadie? La persona que no conoce a nadie conocería a un solo invitado, mientras que la persona que ya conocía a un solo invitado, pasará a conocer a dos invitados.

Nótese que una persona que conocía a un número impar de invitados(conocía a 1) pasó a conocer a un número par de invitados(acabó conociendo a 2), y la que conocía a un número par de invitados(no conocía a ninguno) pasó a conocer a un número impar de invitados(acabó conociendo a 1) ¡el número de personas que conocen a un número impar de invitados no cambió!.

Si nos olvidamos de las personas que se acaban de conocer, y solo sabemos que el número de personas que conocen a un número impar de invitados es par, si dos invitados que no se conocían, de pronto se conocieran, hay 3 posibilidades:

- Una persona conoce a un número impar de invitados y la otra conoce a un número par de invitados.

  En este caso ambos aumentarán su número de conocidos en 1, el que conocía a un número impar de invitados pasará a conocer a un número par de invitados, el que conocía un número par de invitados, pasará a conocer un número impar de invitados. Por ello el número de personas que conocen a un número impar de invitados no cambia y sigue siendo par.

- Ambas personas conocen a un número par de invitados.

  En este caso, ambas personas pasarán a conocer a un número impar de invitados. El número de personas que conocen a un número impar de invitados aumenta en 2, por ello sigue siendo par.

- Ambas personas conocen a un número impar de invitados.

  En este caso, ambas personas pasarán a conocer un número par de invitados. El número de personas que conocen a un número impar de invitados disminuye en 2, por ello sigue siendo par.

Entonces, si al principio de sus vidas nadie se conoce, el número de personas que conocen a un número impar de invitados es par, y conforme se van conociendo, el número seguirá siendo par. Por ello se concluye que siempre será par.□

Ejemplo 1.1.3. Prueba que para todo entero $n \geq 4$, $n! > 2^n$.

Solución Tal vez después de haber leído las soluciones de los dos primeros problemas te haya sido más fácil llegar a la solución de este.

$$4! = (1)(2)(3)(4) = 24$$
$$2^4 = (2)(2)(2)(2) = 16$$

Ya comprobamos que $4! > 2^4$, ahora, suponiendo que para alguna n

$$n! > 2^n$$

proseguimos a comprobar que

$$(n+1)! > 2^{n+1}$$

Partiendo de la desigualdad inicial:

$$n! > 2^n$$

multiplicamos por $n+1$ el miembro izquierdo de la desigualdad y multiplicamos por 2 el lado derecho de la desigualdad. Como $n+1 > 2$, podemos estar seguros que el lado izquierdo de la desigualdad seguirá siendo mayor.

$$(n!)(n+1) > (2^n)2$$

Sintetizando:

$$(n+1)! > 2^{n+1}$$

Ello implica que si se cumple para un número n, también se cumple para $n+1$, como se cumple para 4, entonces se cumplirá para 5, y si se cumple para 5, entonces se cumplirá para 6, etc. Se puede inferir que se cumplirá para todos los números enteros mayores o iguales a 4.

Ejemplo 1.1.4. Prueba por inducción que para todo entero positivo n

$$1^2 + 2^2 + 3^2 + 4^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Solución Nuevamente comenzaremos por el caso mas simple, cuando $n = 1$

$$\frac{1(1+1)(2+1)}{6} = \frac{6}{6} = 1$$

Ahora, suponiendo que para alguna n

$$1^2 + 2^2 + 3^2 + 4^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

veremos que sucede si se le suma $(n+1)^2$ a ambos miembros de la ecuación

$$1^2 + 2^2 + ... + n^2 + (n+1)^2 = \frac{n(n+1)(2n+1)}{6} + \frac{6(n+1)(n+1)}{6}$$

$$1^2 + 2^2 + ... + n^2 + (n+1)^2 = \frac{(n+1)(n(2n+1)+6n+6)}{6}$$

$$1^2 + 2^2 + ... + n^2 + (n+1)^2 = \frac{(n+1)(2n^2+7n+6)}{6}$$

$$1^2 + 2^2 + ... + n^2 + (n+1)^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6}$$

Nuevamente observamos que se cumple con 1, y si se cumple con n también se cumplirá con $n+1$ y por ello con todo número mayor n, por lo tanto se cumple con todos los enteros positivos.

Ejemplo 1.1.5. Muestra que para cualquier cantidad de dinero mayor a 7 centavos puede ser formada usando solo monedas de 3 centavos y de 5 centavos.

Solución La cantidad de 8 centavos puede ser formada con una moneda de 3 y una de 5; la cantidad de 9 centavos puede ser formada con 3 monedas de 9 centavos; la cantidad de 10 centavos puede ser formada con 2 monedas de 5 centavos.

Suponiendo que para algún entero positivo a, tal que $a > 7$ fuera posible formar las cantidades a, $a+1$ y $a+2$, si a cada una de esas cantidades se les añade una moneda de 3 centavos, resuelta que también será posible formar las cantidades $a+3$, $a+4$ y $a+5$, es decir, para cualquier tercia de números consecutivos que se pueda formar, tal que el menor de ellos sea mayor que 7, la siguiente tercia también se podrá formar.

Con ello podemos asumir que cualquier tercia de números consecutivos mayores a 7 se podrá formar y por ende, cualquier número entero positivo mayor a 7.

Ejemplo 1.1.6. Prueba que si se tienen n personas, es posible elegir de entre $2^n - 1$ grupos de personas distintos para hacer una marcha.

Por ejemplo, con 3 personas A, B y C se pueden elegir 7 grupos:

$$\begin{aligned} A \\ B \\ C \\ A, B \\ A, C \\ B, C \\ A, B, C \end{aligned}$$

Solución Consideremos el caso de que solo hay una persona, con esta persona, solamente se puede elegir un grupo(de un solo integrante) para la marcha.

$$2^1 - 1 = 1$$

Ahora, suponiendo que con n personas se pueden elegir $2^n - 1$ grupos, para algún entero n, un desconocido va pasando cerca de las n personas y se une a ellas, puede formar parte de la marcha, o bien no formar parte.

Por cada grupo que se podía formar con a lo más n personas habrá otros 2 grupos con a lo más $n+1$ personas(uno en el caso de que el nuevo integrante participe y otro en el caso de que no participe).
Como la marcha se puede crear solamente con la persona que acaba de llegar.
Entonces con $n+1$ personas habrá $2^{n+1} - 1$ grupos que se pueden elegir.
Con ello queda demostrada la proposición.□

1.2. Errores Comunes

A veces al estar intentando probar algo por inducción se llega a caer en ciertos errores, por ejemplo:

Proposición Todos los perros son del mismo color

Pseudo-prueba Si tenemos un solo perro, es trivial que es de su propio color; ahora, si para algun entero positivo n todos los conjuntos con exactamente n perros constaran de perros del mismo color entonces para cualquier conjunto P con n perros, los cuales llamaremos:

$$p_1, p_2, ..., p_n$$

Existirían 2 perros:

$$p_1, p_{n+1}$$

Tal que se cumpla que p_1 forma parte del conjunto P y p_{n+1} pueda ser cualquier perro que no esté en el conjunto P (a menos que P tuviera todos los perros que existen en cuyo caso se cumpliría la proposición).

Por construccion tendríamos que p_1 es del mismo color que p_{n+1}(ya que todos los conjuntos con n o menos perros tienen solamente perros del mismo color), y que p_1 es del mismo color que todos los perros de P, por lo que $p_1, p_2, ..., p_{n+1}$ serían perros del mismo color, y con esto se comprueba que si cualesquiera n perros son todos del mismo color entonces cualesquiera $n+1$ perros serían del mismo color.

Con inducción queda probado que todos los perros son del mismo color.□

Error Aquí se está suponiendo que se tienen n perros del mismo color, pero implícitamente se está asumiendo que $n \geq 2$ cuando se supone que cualquier pareja de perros va a constar de perros del mismo color.

Está bastante claro que esta *prueba* es incorrecta, sin embargo errores similares pero menos obvios pueden suceder en la práctica.

Siempre hay que verificar que si un caso se cumple el siguiente también lo hará, así como verificar que se este usando el caso base adecuado(en el ejemplo anterior se usó un caso base demasiado pequeño).□

Existe otro error(menos común) en la inducción:

Proposición Todos los números enteros son iguales.

Pseudo-prueba Si para algún número entero k, $k = k + 1$ entonces, sumando 1 en ambos lados de la ecuación, $k + 1 = k + 2$, por lo que por inducción queda demostrado.□

Error Aunque se está tomando en cuenta la segunda condición de la inducción (que si un caso se cumple el siguiente también) no se está tomando en cuenta la primera (que debe de existir un caso en el que se cumpla).□

1.3. Definición de Inducción

Después de estos ejemplos le será posible al lector abstraer lo que tienen en común y así hacerse de una idea clara de lo que es la inducción.

La inducción es un método para demostrar una proposición matemática basándose en la verificación de la forma mas simple de la proposición y luego

comprobando que una forma compleja de la proposición se cumple siempre y cuando una forma mas simple se cumpla.

O dicho de una manera mas formal, es un método para demostrar una sucesión infinita de proposiciones $P_0, P_1, P_2, ...$ demostrando que P_0 es cierta y posteriormente demostrando que si P_k es cierta entonces P_{k+1} también lo es.

1.4. Problemas

1.4.1. Sumas

Demuestra por inducción que:

1. $$1 + 2^2 + 3^2 + 4^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

2. $$1 + 3 + 5 + ... + (2n - 1) = n^2$$

3. $$1 - 2 + 3 - 4 + 5 - 6 + ...n = (-1)^{n+1}\lceil\frac{n}{2}\rceil$$

4. $$1 + 2 + 4 + ... + 2^n = 2^{n+1} - 1$$

5. Cualquier cantidad de dinero mayor que 34 centavos puede ser formada usando solamente monedas de 5 y de 9 centavos.

1.4.2. Tablero de Ajedrez

Demuestra por inducción que:

1. Para todo entero n y todo par m, un tablero de ajedrez de $n \times m$ tiene exactamente el mismo número de cuadros blancos que negros.

2. Para todos los impares n y m, un tablero de ajedrez de $n \times m$ tiene exactamente un cuadro blanco más que cuadros negros.

Un *trimino* es una pieza con forma de "L" compuesta de tres cuadros adyacentes en un tablero de ajedrez. Un arreglo de triminos es una *cobertura* del tablero de ajedrez si cubre todos los cuadros del tablero sin traslape y sin que un trimino quede parcialmente afuera del tablero.

Demuestra por inducción que:

1. Para $n \geq 1$ cualquier tablero de ajedrez de $2^n \times 2^n$, **con un cuadro faltante**, puede ser cubierto con triminos, sin importar en dónde esté el cuadro faltante.

2. Para $n \geq 1$ cualquier tablero de ajedrez de $2^n \times 2^n$, **con un cuadrado faltante**, puede ser cubierto con triminos de solo 3 colores(de manera que 2 triminos del mismo color no tengan bordes en común), sin importar en dónde esté el cuadro faltante.

1.4.3. Chocolate

Tiempo Límite:1 Segundo

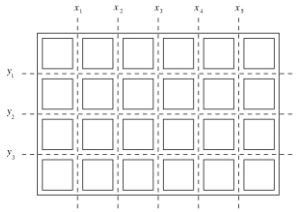
Supongamos que tenemos una barra de chocolate de m x n piezas cuadradas de 1x1(es una suposición, por lo tanto no puedes comertela) y debes partirla en cuadrados de 1 x 1.

Las partes del chocolate pueden ser cortadas a través de cortes horizontales y/o verticales como se muestra en la figura. Un corte(ya sea horizontal o vertical) de un pedazo del chocolate siempre divide ese pedazo en dos pedazos mas pequeños.

Como todo cuesta en esta vida, cada corte que realizes en el chocolate también tendrá un costo, dicho costo se puede expresar como un número entero positivo. Este costo no depende del tamaño del pedazo que se corte, sino que depende de la recta horizontal o vertical por la cual se esté cortando.

Denotaremos los costos de cortar por cada recta vertical como $x_1, x_2, x_3, ..., x_{m-1}$ y los costos de cortar por cada recta horizontal como $y_1, y_2, y_3, ..., y_{n-1}$.

El costo de cortar la barra entera es la suma de los costos de todos los cortes requeridos.



Por ejemplo, si cortamos el chocolate a lo largo de las rectas horizontales y después cada pedazo obtenido lo cortamos a lo largo de las rectas verticales, el costo total por cortar la barra será $y_1 + y_2 + y_3 + 4(x_1 + x_2 + x_3 + x_4 + x_5)$.

Problema

Escribe un programa que dado el tamaño de la barra de chocolate, determine el costo mínimo para cortarla en cuadrados de 1x1.

Entrada

Descripción

Línea 1: Dos enteros positivos m y n separados por un espacio
Siguientes m -1 líneas: Los valores de x_1, x_2, x_3, ..., x_{m-1}
Siguientes n -1 líneas: Los valores de $y_1, y_2, y_3, ..., y_{n-1}$

Ejemplo

6 4
2
1
3
1
4
4
1
2

Salida

Descripción

Línea 1: Un solo número entero: el costo mínimo de cortar todo el chocolate en cuadrados de 1x1

Ejemplo

42

Límites

$2 \leq m, n \leq 1000$
Ninguno de los costos superará a 1000

Referencias

Fuente: X Polish Olympiad in Informatics 2002/2003
Autor: Marcin Kubica
Traductor: Luis Enrique Vargas Azcona

1.5. Sugerencias

Esta sección está dedicada a dar pistas para encontrar algunas soluciones de los problemas del capítulo. Suele ser mejor leer una sugerencia y volver a intentar el problema que pasar directo a la solución.

Sumas

Sugerencia para el quinto inciso: intenta demostrar que si puedes formar 5 números consecutivos n, $n+1$, $n+2$, $n+3$ y $n+4$, entonces también puedes formar $n+5$, $n+6$, $n+7$, $n+8$ y $n+9$.

Tablero de Ajedrez

Sugerencias para el conteo de cuadros blancos y negros:

1. Intenta demostrarlo para un tablero de 2 x n y luego usar esa demostración como caso pase para uno de n x m.

2. Intenta usar el inciso anterior agregando/quitando una columna.

Sugerencias para los problemas de triminos:

1. Intenta juntar 4 tableros de 2^n x 2^n para obtener un tablero de 2^{n+1} x 2^{n+1}. ¿Puedes usar el hecho de que puedes poner el vacío donde quieras para unir los 4 tableros?.

2. Luego de ver casos pequeños, ¿notas algo en los bordes?, ¿Puedes intercambiar algunos colores para lograr juntar los tableros como en el inciso anterior?.

Chocolate

¿Qué sucede si en lugar de buscar cómo cortar el chocolate supones que el chocolate ya está cortado y quieres unir las piezas con un procedimiento similar?, ¿en qué cambia el problema?

Capítulo 2

Definición y Características de la Recursión

Ya vimos con la inducción que a veces una proposición se puede verificar en su forma más simple y luego probar que si se cumple para una de cierta complejidad también se cumplirá para otra con más complejidad.

Ahora, la recursividad o recursión se trata de algo parecido, se trata de definir explícitamente la forma más simple de un proceso y definir las formas mas complejas de dicho proceso en base a formas un poco más simples. Es decir:

Definición 2.0.1 (Recursión)**.** Forma de definir un objeto o un proceso definiendo explícitamente su forma mas simple, y definiendo sus formas mas complejas con respecto a formas mas simples.

Veremos cual es la utilidad de este tipo de definiciones con los siguientes ejemplos:

2.1. Factorial

$n!$ se lee como *ene factorial* y
$n! = (1)(1)(2)(3)(4)(5)...(n-1)(n)$
Por ejemplo $5! = (1)(1)(2)(3)(4)(5) = 120$ y se lee como *cinco factorial*
Pero puede parecer incómodo para una definición seria tener que usar los puntos suspensivos(...) y depender de cómo la interprete el lector.
Por ello, vamos a convertir esa definición en una definición recursiva.
La forma mas simple de la función factorial es:

$$0! = 1$$

Y ahora, teniendo $n!$, ¿Cómo obtenemos $(n+1)!$?, simplemente multiplicando $n!$ por $n+1$.

$$(n+1)! = (n!)(n+1)$$

De esta forma especificamos cada que se tenga un número, cómo obtener el siguiente y la definición queda completa, ésta ya es una definición recursiva, sin embargo, se ve un tanto sucia, puesto que define a $(n+1)!$ en términos de $n!$. Y para ir más de acuerdo con la idea de la recursividad hay que definir $n!$ en términos de $(n-1)!$.

Así que, teniendo $(n-1)!$, para obtener $n!$ hay que multiplicar por n. De esa forma nuestra definición recursiva queda así:

Definición 2.1.1 (Factorial)**.**

$$0! = 1$$

$$n! = (n-1)!n$$

Al igual que las matemáticas, una computadora también soporta funciones recursivas, por ejemplo, la función factorial que acaba de ser definida puede ser implementada en lenguaje C de la siguiente manera:

Código 2.1: Función recursiva factorial

```
1  int factorial(int n){
2      if(n == 0) return 1;
3      else return factorial(n-1)*n;
4  }
```

Para los estudiantes de programación que comienzan a ver recursión suele causar confusión ver una función definida en términos de sí misma. Suele causar la impresión de ser una definición cíclica. Pero ya que construimos la función paso a paso es posible que ello no le ocurra al lector.

De todas formas es buena idea ver cómo trabaja esta función en un depurador o ir simulándola a mano.

Podemos notar algunas cosas interesantes:

- Si llamamos a factorial(5), la primera vez que se llame a la función factorial, n será igual a 5, la segunda vez n será igual a 4, la tercera vez n será igual a 3, etc.

- Después de que la función regresa el valor 1, n vuelve a tomar todos los valores anteriores pero esta vez en el orden inverso al cual fueron llamados, es decir, en lugar de que fuera 5, 4, 3, 2, 1, ahora n va tomando los valores 1, 2, 3, 4, 5; conforme el programa va regresando de la recursividad va multiplicando el valor de retorno por los distintos valores de n.

- Cuando la función factorial regresa por primera vez 1, la computadora recuerda los valores de n, entonces guarda dichos valores en algún lado.

La primera observación parece no tener importancia, pero combinada con la segunda se vuelve una propiedad interesante que examinaremos después. La tercera observación hace obvia la necesidad de una estructura para recordar los valores de las variables cada que se hace una llamada a función.

Dicha estructura recibe el nombre de pila o *stack*. Y por el solo hecho de contener información, la pila requiere memoria de la computadora y tiene un límite de memoria a ocupar(varía mucho dependiendo del compilador y/o sistema operativo), si dicho límite es superado, el programa terminará abruptamente por acceso a un área restringida de la memoria, este error recibe el nombre de desbordamiento de pila o *stack overflow*.

A veces, al abordar un problema, es mejor no pensar en la pila dado que el tamaño de la pila nunca crecerá mucho en ese problema; otras veces hay que tener cuidado con eso; y en ocasiones, tener presente que la pila recuerda todo puede ayudar a encontrar la solución a un problema.

Para observar mas claramente las propiedades mencionadas, conviene echar un vistazo a esta versión modificada de la función factorial:

Código 2.2: Función recursiva factorial modificada

```
1  int factorial(int n){
2                  int fminus1;
3                  printf("%d\n", n);
4                  if(n == 0) return 1;
5                  fminus1=factorial(n-1);
6                  printf("%d %d\n", n, fminus1);
7                  return fminus1*n;
8          }
```

Si llamas factorial(5) con el código de arriba obtendrás la siguiente salida en pantalla:

```
5
4
3
2
1
0
1 1
2 1
3 2
4 6
5 24
```

Allí se ve claramente cómo n toma los valores en el orden inverso cuando va regresando de la recursividad, como la línea 3 se ejecuta 6 veces mientras se van haciendo las llamadas recursivas y como la linea 6 se ejecuta ya se hicieron todas las llamadas recursivas, no antes.

2.2. Imprimir Números en Binario

Con la función factorial ya vimos varias propiedades de las funciones recursivas cuando se ejecutan dentro de una computadora.

Aunque siendo sinceros, es mucho mas práctico calcular el factorial con un for que con una función recursiva.

Ahora, para comenzar a aprovechar algunas ventajas de la recursividad se tratará el problema de imprimir números en binario, el cual es fácilmente extendible a otras bases.

Cada vez que observemos un número con subíndice, vamos a interpretar el subíndice como la base de numeración en la cual el número está escrito.

Por ejemplo 101_2 significa el número binario 101, es decir, 5 en sistema decimal.

Estos son los primeros números enteros positivos en sistema binario:

$1_2, 10_2, 11_2, 100_2, 101_2, 110_2, 111_2, 1000_2, 1001_2, 1010_2, 1011_2, ...$

Esto no debe causar confusión en el momento en el que veamos los subíndices para denotar sucesiones, por ejemplo $x_1, x_2, x_3, ..., x_n$, y esto es porque en expresiones tales como x_1, se tiene que x no está escrito en ninguna base.

Ejemplo 2.2.1. Escribe una función recursiva que imprima un número entero positivo en su formato binario(no dejes ceros a la izquierda).

Solución La instancia mas simple de este proceso es un número binario de un solo dígito(1 ó 0). Cuando haya mas de un dígito, hay que imprimir primero los dígitos de la izquierda y luego los dígitos de la derecha.

También hay que hacer notar algunas propiedades de la numeración binaria:

- Si un número es par, termina en 0, si es impar termina en 1.
  Por ejemplo $101_2 = 5$ y $100_2 = 4$
- Si un número se divide entre 2(ignorando el residuo), el cociente se escribe igual que el dividendo, simplemente sin el último dígito de la derecha.

  Por ejemplo $\frac{10011001_2}{2} = 1001100_2$

Tomando en cuenta todo lo anterior concluimos que si $n < 2$ entonces hay que imprimir n, si no, hay que imprimir $n/2$ y luego imprimir 1 si n es impar y 0 si n es par. Por lo tanto, la función recursiva quedaría de esta manera:

Código 2.3: Función recursiva que Imprime un Número Binario

```
1 void imprime_binario(int n){
2         if(n>=2){
3                 imprime_binario(n/2);
4                 printf("%d", n%2);
5         }else{
6                 printf("%d", n);
7         }
8 }
```

□

Aquí vemos que esta función recursiva si supone una mejoría en la sencillez en contraste si se hiciera iterativamente(no recursivo), ya que para hacerlo iterativamente se necesitaría llenar un arreglo y en cada posición un dígito, y luego recorrerlo en el orden inverso al que se llenó.

Pero hay que hacer notar que un proceso recursivo funciona ligeramente mas lento que si se hiciera de manera iterativa. Esto es porque las operaciones de añadir elementos a la pila y quitar elementos de la pila toman tiempo.

Por eso se concluye que la recursión hay que usarla cuando simplifique sustancialmente las cosas y valga la pena pagar el precio con un poco menos de rendimiento.

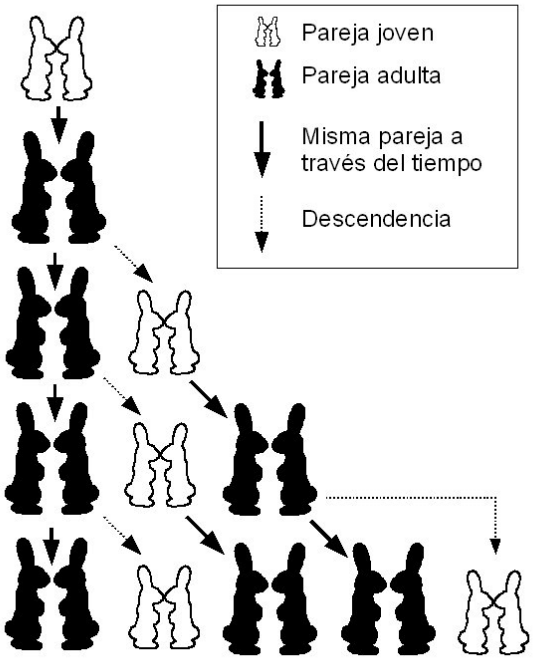


Figura 2.1: Reproducción de los Conejos de Fibonacci

2.3. Los Conejos de Fibonacci

Había una vez cierto matemático llamado Leonardo de Pisa, apodado Fibonacci, que propuso el siguiente problema:

Ejemplo 2.3.1. Alguien compra una pareja de conejos(un macho y una hembra), luego de un mes de haber hecho la compra esos conejos son adultos, después de dos meses de haber hecho la compra esa pareja de conejos da a luz a otra pareja de conejos(un macho y una hembra), al tercer mes, la primera pareja de conejos da a luz a otra pareja de conejos y al mismo tiempo, sus primeros hijos se vuelven adultos.

Cada mes que pasa, cada pareja de conejos adultos da a luz a una nueva pareja de conejos, y una pareja de conejos tarda un mes en crecer. Escribe una función que regrese cuántos conejos adultos se tienen pasados n meses de la compra.

Solución Sea $F(x)$ el número de parejas de conejos adultos pasados x meses. Podemos ver claramente que pasados 0 meses hay 0 parejas adultas y pasado un mes hay una sola pareja adulta. Es decir $F(0) = 0$ y $F(1) = 1$.

Ahora, suponiendo que para alguna x ya sabemos $F(0)$, $F(1)$, $F(2)$, $F(3)$, ..., $F(x-1)$, en base a eso ¿cómo podemos averiguar $F(x)$?

Si en un mes se tienen a parejas jóvenes y b parejas adultas, al siguiente mes se tendrán $a+b$ parejas adultas y b parejas jóvenes.

Por lo tanto, el número de conejos adultos en un mes n, es el número de conejos adultos en el mes n-1 más el número de conejos jóvenes en el mes n-1.

Como el número de conejos jóvenes en el mes n-1 es el número de conejos adultos en el mes n-2, entonces podemos concluir que:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

Como curiosidad matemática, posiblemente alguna vez leas u oigas hablar sobre la serie o sucesión de Fibonacci, cuando suceda, ten presente que la serie de Fibonacci es

$$F(0), F(1), F(2), F(3), F(4), F(5), ...$$

O escrita de otra manera:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$$

Además de servir como solución a este problema, la serie de Fibonacci cumple también con muchas propiedades interesantes que pueden servir para resolver o plantear otros problemas.

Por el momento no se mostrarán aquí, ya que ese no es el objetivo del libro, pero si te interesa puedes investigarlo, en internet hay bastante referente a eso.

El siguiente código en C muestra una implementación de una función recursiva para resolver el problema planteado por Fibonacci:

Código 2.4: Función recursiva de la serie de Fibonacci

```
1 int F(int n){
2         if(n==0){
3                 return 0;
4           }else if(n==1){
5                 return 1;
```

```
6               }else{
7                       return F(n-1)+F(n-2);
8               }
9
10  }
```

□

Si corres el código anterior en una computadora, te darás cuenta que el tamaño de los números crece muy rápido y con números como 39 o 40 se tarda mucho tiempo en responder, mientras que con el número 50 parece nunca terminar.

Hemos resuelto matemáticamente el problema de los conejos de Fibonacci, sin embargo ¡esta solución es irrazonablemente lenta!.

Capítulo 3

Recursión con Memoria o Memorización

La recursión con Memoria o Memorización es un método para evitar que una misma función recursiva se calcule varias veces ejecutándose bajo las mismas condiciones; consiste en tener en una estructura(por lo general un arreglo de una o varias dimensiones) para guardar los resultados ya calculados.

3.1. Mejorando el Rendimiento de Fibonacci

Ejemplo 3.1.1. Recordando, la sucesión de Fibonacci se define como

$$F(0) = 0$$

$$F(1) = 1$$

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

Escribe un programa que calcule e imprima los primeros 50 números de la serie de Fibonacci en menos de un segundo.

Solución Ya habíamos visto en la sección anterior que la función recursiva de Fibonacci(véase código 2.4) es extremadamente lenta, vale la pena preguntarnos ¿Qué la hace lenta?.

Si nos ponemos a ver su ejecución en un depurador o la realizamos mentalmente, nos daremos cuenta que para calcular $F(n)$, se está calculando $F(n-1)$ y $F(n-2)$, y para calcular $F(n-1)$ también se está calculando $F(n-2)$.

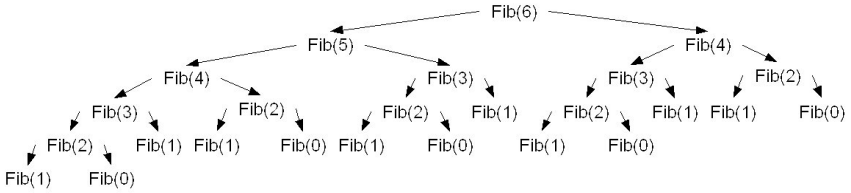


Figura 3.1: Llamadas a función que se realizan para calcular $F(6)$

Una vez que se terminó de calcular $F(n-1)$ ya se había calculado $F(n-2)$, y sin embargo, se vuelve a calcular. Con ello, ya hemos visto que $F(n-2)$ se esta calculando 2 veces.

Extrapolando este razonamiento $F(n-4)$ se está calculando al menos 4 veces, $F(n-6)$ se esta calculando al menos 8 veces, etc. Es decir, para algún $m < n$ tal que m es par $F(n-m)$ se calcula al menos $2^{\frac{m-1}{2}}$ veces(es un ejercicio corto demostrarlo por inducción).

¿Exactamente de cuántas veces estamos hablando que se calcula cada número de Fibonacci?. Por el momento no vale la pena ver eso, es suficiente con saber que es un número exponencial de veces.

Para solucionar esta problemática, vamos a declarar un arreglo v de tamaño 50 y con el tipo de datos long long, ya que a nadie le extrañaría que $F(49) > 2^{32}$.

Luego, cada que se ejecute la función $F(n)$, verificará si $v[n]$ ya fue usado(los valores de v se inicializan en 0, ya que es un arreglo global), si ya fue usado simplemente regresará $v[n]$ si no ha sido usado entonces calculará el valor de $F(n)$ y lo guardará en $v[n]$.

Así concluimos nuestros razonamientos para llegar a este código:

Código 3.1: Fibonacci utilizando recursión con memoria

```
1  #include <stdio.h>
2  int v[50];
3  int F(int n){
4          if(v[n]!=0){
5                  return v[n];
6            }if(n==0){
7                  return 0;
8            }if(n==1){
9                  return 1;
10           }v[n]=F(n-1)+F(n-2);
11          return v[n];
12   }
```

```
13  int main(){
14          int i;
15           for(i=0;i<50;i++){
16                   printf("%ld$\$n", F(i));
17            }return 0;
18   }
```

□

Aunque el código anterior es notablemente mas largo que el código 2.4, solamente las líneas de la 2 a la 12 son interesantes, lo demás es solamente para imprimir en pantalla el resultado.

Si ejecutas esta nueva función F en una computadora te darás cuenta que es mucho mas rápida que la función F que se muestra en el código 2.4.

3.2. Error Común en la Memorización

La recursión con memoria solamente sirve cuando la función recursiva que se quiere optimizar no utiliza los valores de variables estáticas ni globales para calcular su resultado; de lo contrario si la función es llamada con los mismos parámetros no se garantiza que se esté calculando exactamente lo mismo mas de una vez y la recursión con memoria no sirve en ese caso.

Es común para alguien que va empezando en la recursión con memoria cometer este tipo de errores por ello hay que tomar precauciones adicionales.

3.3. Triangulo de Pascal

Probablemente alguna vez viste este triángulo:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  .....
```

Este triángulo se conoce como Triángulo de Pascal, la manera de construirlo es bastante simple, se coloca un 1 en el primer renglón, en cada renglón

a partir del segundo, se forman números sumando cada par de números adyacentes del renglón anterior(es decir, cada número es la suma de los dos números que tiene arriba) y se coloca un 1 en cada extremo del renglón.

Vamos a llamar al j-ésimo número del i-ésimo renglón $P(i, j)$.

Ejemplo 3.3.1. Escribe una función recursiva que regrese $P(i, j)$, para $i < 50$ y $j < 50$ y $P(i, j) < 2^31$.

Primero hay que notar que $P(1, 1) = 1$, luego que el primer numero de cada renglón también es 1; es decir $P(i, 1) = 1$, y el último número de cada renglón también es 1, osea $P(i, i) = 1$; en otro caso es la suma de los dos números que tiene arriba, es decir $P(i, j) = P(i - 1, j - 1) + P(i - 1, j)$. De esta manera completamos la función recursiva así:

$$P(i, 1) = 1$$

$$P(i, i) = 1$$

$$P(i, j) = P(i - 1, j - 1) + P(i - 1, j) \text{ para } 1 < j < i$$

Aquí se ignoró $P(1, 1) = 1$ ya que $P(i, 1) = 1$ implica $P(1, 1) = 1$ Nuevamente utilizaremos un arreglo llamado v para guardar los resultados ya calculados, y ya que ningún resultado puede ser 0, cuando $v[i][j]$ sea 0 sabremos que no se ha calculado aún. Ahora el código de la función en C resulta así:

Código 3.2: Triangulo de Pascal utilizando Recursión con Memoria

```
1  int v[51][51];
2  int P(int i, int j){
3          if(j==1 || i==j){
4                  return 1;
5          }if(v[i][j]!=0){
6                   return v[i][j];
7          }
8          v[i][j]=P(i-1, j-1)+P(i-1, j);
9          return v[i][j];
10 }
```

□

El triángulo de Pascal tiene varias propiedades útiles, una muy útil sirve para calcular combinaciones.

Definición 3.3.1 (Combinaciones)**.** Las combinaciones de n en m son la cantidad de formas de escoger m objetos entre un total de n objetos distintos. Se denota como:

$$\binom{m}{n}$$

Por convención diremos que $\binom{n}{0} = 1$ para cualquier entero no negativo n.

La última parte de la definición puede parecer que va contra el sentido común, ya que $\binom{n}{0}$ indica el número de maneras de elegir 0 objetos de un total de n objetos distintos, se suele pensar que si no se elige ningún objeto realmente no se está *eligiendo* nada por lo que $\binom{n}{0}$ debería ser igual a 0 pero en realidad es igual a 1.

El motivo de esta confusión proviene de la palabra *elegir*, no está claro si es válido decir que se eligen 0 objetos. Mas adelante, cuando se introduzca la noción de conjuntos, no sonará extraño el hecho de que se pueden elegir 0 objetos de una sola manera; pero por el momento el lector tendrá que adaptarse a la idea *antinatural* de que es válido elegir 0 objetos.

Ejemplo 3.3.2. Prueba que $\binom{i-1}{j-1} = P(i,j)$.

Solución Recordemos cómo se calcula P:

$$P(i,1) = 1$$

$$P(i,i) = 1$$

$$P(i,j) = P(i-1,j-1) + P(i-1,j) \text{ para } 1 < j < i$$

De esa manera obtenemos 3 proposiciones:

- $P(i,1) = 1$ implica $\binom{i-1}{0} = 1$, es decir, el número de formas de elegir 0 objetos entre un total de $i-1$ objetos distintos es 1.
- $P(i,i) = 1$ implica $\binom{i-1}{i-1} = 1$, es decir, el número de formas de elegir $i-1$ objetos entre un total de $i-1$ objetos distintos es 1.
- $P(i,j) = P(i-1,j-1) + P(i-1,j)$ implica $\binom{i-1}{j-1} = \binom{i-2}{j-2} + \binom{i-2}{j-1}$ para $1 < j < i$ esta proposición es equivalente a:

$$\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$$

La primera proposición es obvia, solo hay una manera de elegir 0 objetos entre un total de $i-1$ objetos distintos, y ésta es no eligiendo ningún objeto.

La segunda proposición también es obvia, la única manera de elegir $i-1$ objetos de entre un total de $i-1$ objetos es eligiendo todos.

La tercera proposición es más difícil de aceptar, digamos que se tienen n objetos numerados de 1 a n; y se quieren elegir m objetos entre ellos, en particular, se puede optar entre elegir el objeto n o no elegir el objeto n.

El número de formas de elegir m objetos de entre un total de n objetos distintos, es el número de formas de hacer eso eligiendo al objeto n más el número de formas de hacer eso sin elegir al objeto n.

Si se elige el objeto n, entonces habrá que elegir $m-1$ objetos de entre un total de $n-1$ objetos distintos; hay $\binom{n-1}{m-1}$ formas distintas de hacerlo.

Si no se elige el objeto n, entonces habrá que elegir m objetos de entre un total de $n-1$ objetos distintos(si ya se decidió no elegir a n, entonces quedan $n-1$ objetos que pueden ser elegidos); hay $\binom{n-1}{m}$ formas de hacerlo.

Por ello concluimos que $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$ lo que prueba tercera proposición.□

3.4. Teorema del Binomio

El triángulo de Pascal tiene muchas propiedades interesantes, una de ellas está estrechamente relacionada con el teorema del binomio; el cual se le atribuye a Newton.

Para dar una idea de a qué se refiere el teorema del binomio; basta ver los coeficientes de las siguientes ecuaciones:

$$(a+b)^0 = 1$$
$$(a+b)^1 = a+b$$
$$(a+b)^2 = a^2+2ab+b^2$$
$$(a+b)^3 = a^3+3a^2b+3ab^2+b^3$$
$$(a+b)^4 = a^4+4a^3b+6a^2b^2+4ab^3+b^4$$
$$(a+b)^5 = a^5+5a^4b+10a^3b^2+10a^2b^3+5ab^4+b^5$$

Después de ver los coeficientes es natural que a uno le llegue a la mente el triángulo de Pascal y eso es exactamente a lo que se refiere el teorema del binomio.

Teorema 1 (Teorema del Binomio)**.** *Para cualquier entero no negativo n:*

$$(a+b)^n = \binom{n}{0}a^nb^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \binom{n}{3}a^{n-3}b^3 + ... + \binom{n}{n}a^0b^n$$

Ejemplo 3.4.1. Demuestra el teorema del binomio

Solución Ya vimos que para n=0 si se cumple el teorema. Ahora, suponiendo que para algún número n se cumple el teorema ¿se aplicará cumplirá $n+1$? Por construcción vamos suponiendo que para alguna n:

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \binom{n}{3}a^{n-3}b^3 + ... + \binom{n}{n}a^0 b^n$$

Multiplicamos ambos miembros de la ecuación por $(a+b)$

$$((a+b)^n)(a+b) = (\binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + ... + \binom{n}{n}a^0 b^n)(a+b)$$

$$(a+b)^{n+1} = \binom{n}{0}a^{n+1}b^0 + (\binom{n}{1}+\binom{n}{0})a^n b^1 + (\binom{n}{2}+\binom{n}{1})a^{n-1}b^2 + ... + \binom{n}{n}a^0 b^{n+1}$$

Recordamos que $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$.

$$(a+b)^{n+1} = \binom{n}{0}a^{n+1}b^0 + \binom{n+1}{1}a^n b^1 + ... + \binom{n+1}{n}a^1 b^n + \binom{n}{n}a^0 b^{n+1}$$

Como $\binom{n}{n} = \binom{n+1}{n+1} = 1$ y $\binom{0}{n} = \binom{0}{n+1} = 1$ entonces:

$$(a+b)^{n+1} = \binom{n+1}{0}a^{n+1}b^0 + \binom{n+1}{1}a^n b^1 + ... + \binom{n+1}{n+1}a^0 b^{n+1}$$

Por inducción el teorema queda demostrado.

□

Debido al teorema del binomio, a las combinaciones tambien se les llama **coeficientes binomiales**.

El Triangulo de Pascal aún tiene muchas más propiedades interesantes, pero por el momento esto es todo lo que hay en el libro, si algún lector se quedó intrigado por este famoso triángulo puede buscar en internet y encontrará muchas cosas.

Capítulo 4

Divide y Vencerás

Ésta célebre frase para estrategias de guerra ha llegado a ser bastante popular en el campo de las matemáticas y sobre todo, en el de las matemáticas aplicadas a la computación.

La estrategia *Divide y Vencerás* se define de una manera bastante simple:

Divide un problema en partes mas pequeñas, resuelve el problema por las partes, y combina las soluciones de las partes en una solución para todo el problema.

Es difícil encontrar un problema donde no se utilice ésta estrategia de una u otra forma.

Sin embargo, aquí se tratará exclusivamente de la estrategia Divide y Vencerás en su forma recursiva:

- *Divide*. Un problema es dividido en copias mas pequeñas del mismo problema.
- *Vence*. Se resuelven por separado las copias mas pequeñas del problema. Si el problema es suficientemente pequeño, se resuelven de la manera mas obvia.
- *Combina*. Combina los resultados de los subproblemas para obtener la solución al problema original.

La dificultad principal en resolver este tipo de problemas radica un poco en cómo dividirlos en copias mas pequeñas del mismo problema y sobre todo cómo combinarlos.

A veces la estrategia recursiva Divide y Vencerás mejora sustancialmente la eficiencia de una solución, y otras veces sirve solamente para simplificar las cosas.

Como primer ejemplo de Divide y Vencerás veremos un problema que puede resultar mas sencillo resolverse sin recursión, pero esto es solo para dar una idea de cómo aplicar la estrategia.

4.1. Máximo en un Arreglo

Ejemplo 4.1.1. Escribe una función que dado un arreglo de enteros v y dados dos enteros a y b, regrese el número mas grande en $v[a..b]$(el número mas grande en el arreglo que esté entre los índices a y b, incluido este último).

Solución Lo mas sencillo sería iterar desde a hasta b con un for, guardar el máximo en una variable e irla actualizando.

Pero una forma de hacerlo con Divide y Vencerás puede ser:

- Si $a < b$, dividir el problema en encontrar el máximo en $v[a..(a+b)/2]$ y el máximo en $v[(a+b)/2+1..b]$ y resolver ambos problemas recursivamente.
- Si $a = b$ el máximo sería $v[a]$
- Una vez teniendo las respuestas de ambos subproblemas, ver cual de ellas es mayor y regresar esa.

Los 3 puntos de éste algoritmo son los 3 puntos de la estrategia Divide y Vencerás aplicados.

Esta claro que este algoritmo realmente encuentra el máximo, puesto que en $v[a..b]$ está compuesto por $v[a..(a+b)/2]$ y $v[(a+b)/2+1..b]$, sin quedar un solo número del intervalo excluido. También sabemos que si un número $x > y$ y $y > z$, entonces $x > z$, por ello podemos estar seguros que alguno de los dos resultados de los subproblemas debe de ser el resultado al problema original.

Para terminar de resolver este problema, hay que escribir el código:

Código 4.1: Máximo en un intervalo cerrado a, b

```
1  int maximo(int v[], int a, int b){
2  int maximo1, maximo2;
3   if(a<b){
4          maximo1=maximo(v, a, (a+b)/2);
5           maximo2=maximo(v, (a+b)/2+1, b);
6           if(maximo1>maximo2){
7                  return maximo1;
```

```
8                       }else{
9                               return maximo2;
10                      }
11              }else{
12                      return v[a];
13              }
14      }
```

□

4.2. Búsqueda Binaria

Ahora, después de haber visto un ejemplo impráctico sobre el uso de Divide y Vencerás, veremos un ejemplo práctico.

Supongamos que tenemos un arreglo v con los números ordenados de manera ascendente. Es decir, $v[a] > v[a-1]$ y queremos saber si un número x se encuentra en el arreglo, y de ser así, ¿dónde se encuentra?

Una posible forma sería iterar desde el principio del arreglo con un for hasta llegar al final y si ninguno de los valores es igual a x, entonces no está, si alguno de los valores es igual a x, guardar dónde está.

¿Pero qué sucedería si quisiéramos saber un millón de veces dónde está algún número(el número puede variar)? Como ya te lo podrás imaginar, el algoritmo anterior repetido un millón de veces se volverá lento.

Es como si intentáramos encontrar el nombre de un conocido en el directorio telefónico leyendo todos los nombres de principio a fin a pesar de que están ordenados alfabéticamente.

Ejemplo 4.2.1. Escribe una función que dado un arreglo ordenado de manera ascendente, y tres enteros a, b y x, regrese -1 si x no está en $v[a..b]$ y un entero diciendo en qué índice se encuentra x si lo está. Tu programa no deberá hacer mas de 100 comparaciones y puedes asumir que $b-a<1000000$.

Solución La solución a este problema se conoce como el algoritmo de la búsqueda binaria.

La idea consiste en ver qué número se encuentra a la mitad del intervalo $v[a..b]$. Si $v[(a+b)/2]$ es menor que x, entonces x deberá estar en $v[(a+b)/2+1..b]$, si $v[(a+b)/2]$ es mayor que x, entonces x deberá estar en v$[a..(a+b)/2]$.

En caso de que $a = b$, si $v[a] = x$, entonces x se encuentra en a.

Así que, los 3 pasos de la estrategia Divide y Vencerás con la búsqueda binaria son los siguientes:

- Si $b > a$, comparar $v[(a+b)/2]$ con x, si x es mayor entonces resolver el problema con $v[(a+b)/2+1..b]$, si x es menor resolver el problema con $v[a..(a+b)/2-1]$, y si x es igual, entonces ya se encontró x.
- Si $b \geq a$, comparar $v[a]$ con x, si x es igual entonces se encuentra en a, si x es diferente entonces x no se encuentra.
- Ya sabiendo en qué mitad del intervalo puede estar, simplemente hay que regresar el resultado de ese intervalo.

Código 4.2: Búsqueda Binaria Recursiva

```
1        int Busqueda_Binaria(int v[], int a, int b, int
             x){
2        if(a>=b){
3                if(v[a]==x)
4                        return a;
5                 else
6                        return -1;
7          }
8         if(v[(a+b)/2]==x){
9                return (a+b)/2;
10         }else if(v[(a+b)/2]<x){
11               return Busqueda_Binaria(v, (a+b)/2+1, b,
                     x);
12         }else{
13               return Busqueda_Binaria(v, a, (a+b)/2-1,
                     x);
14         }
15  }
```

A pesar de que la idea de la búsqueda binaria es puramente recursiva, también se puede eliminar por completo la recursión de ella:

Código 4.3: Búsqueda Binaria Iterativa

```
1  int Busqueda_Binaria(int v[], int a, int b, int x){
2        while(a<b)
3                 if(v[(a+b)/2]==x){
4                        return (a+b)/2;
5                 }else if(v[(a+b)/2]<x){
6                        a=(a+b)/2+1;
7                 }else{
8                        b=(a+b)/2-1;
```

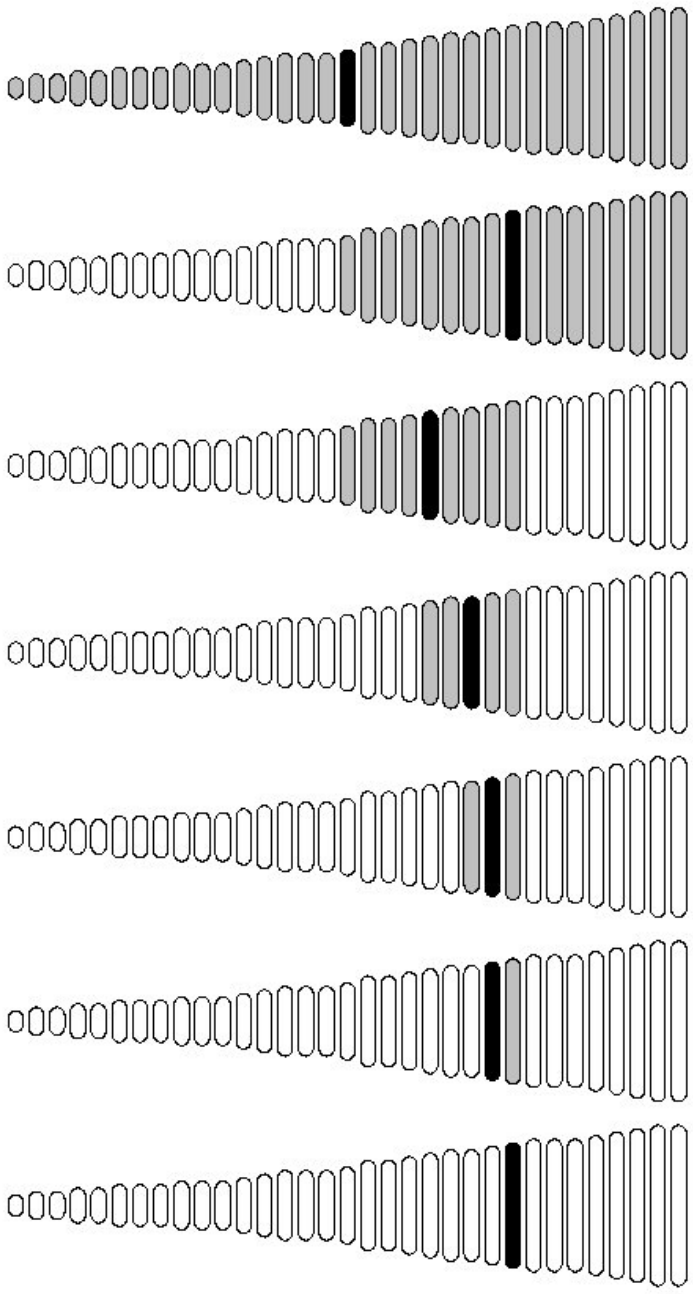


Figura 4.1: Rendimiento de la Búsqueda Binaria. El espacio de búsqueda(intervalo donde puede estar la solución) esta marcado con gris, y el valor que se esta comparando está marcado con negro.

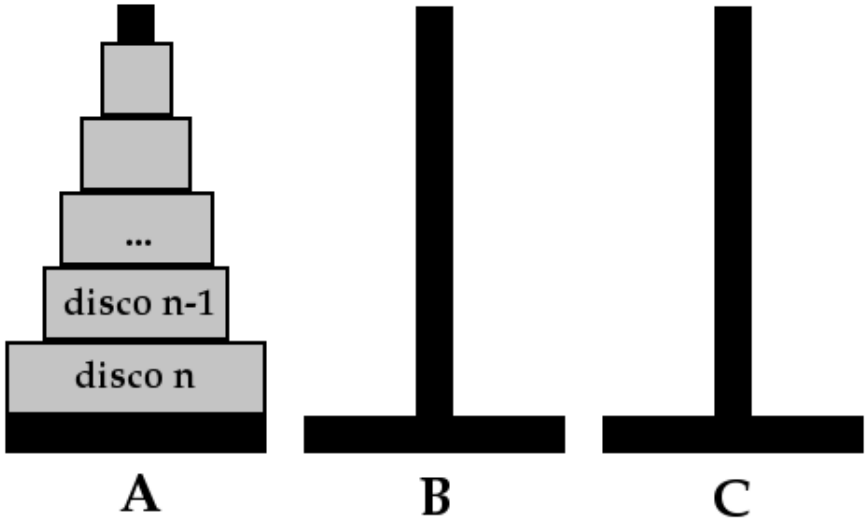


Figura 4.2: Tórres de Hanoi

```
9                       }
10              if(v[a]==x){
11                      return a;
12              }else{
13                      return -1;
14              }
15  }
```

Aunque de que los códigos están del mismo tamaño, muchas veces resulta mas práctica la iterativa, ya que no es necesario declarar una nueva función para realizarla.□

4.3. Torres de Hanoi

El problema de las Torres de Hanoi es un problema utilizado frecuentemente como ejemplo de recursión.

Imagina que tienes 3 postes llamados A, B y C.

En el poste A tienes n discos de diferente diámetro, acomodados en orden creciente de diámetro desde lo más alto hasta lo más bajo.

Solamente puedes mover un disco a la vez desde un poste hasta otro y no esta permitido poner un disco mas grande sobre otro mas pequeño. Tu tarea es mover todos los discos desde el poste A hasta el poste C.

Ejemplo 4.3.1. Escribe una función que reciba como parámetro n y que imprima en pantalla todos los pasos a seguir para mover los discos del poste A al poste C.

Solución Pensando primero en el caso mas pequeño y trivial si $n = 1$, tendríamos un solo disco y solo habría que moverlo de la torre A a la C.

Ahora, suponiendo que para algún n ya sabemos cómo mover $n-1$ discos de una torre a cualquier otra ¿qué deberíamos hacer?

Luego de hacerse esta pregunta es fácil llegar a la conclusión de que primero hay que mover los primeros $n-1$ discos a la torre B, luego el disco n a la torre C, y posteriormente mover los $n-1$ discos de la torre B a la torre C.

Podemos estar seguros que lo anterior funciona ya que los primeros $n-1$ discos de la torre siempre serán mas pequeños que el disco n, por lo cual podrían colocarse libremente sobre el disco n si así lo requirieran.

Se puede probar por inducción el procedimiento anterior funciona si se hace recursivamente.

Así que nuestro algoritmo de Divide y Vencerás queda de la siguiente manera:

Sea x la torre original, y la torre a la cual se quieren mover los discos, y z la otra torre.

- Para $n > 1$, hay que mover $n-1$ discos de la torre x a la z, luego mover un disco de la torre x a la y y finalmente mover $n-1$ discos de la torre z a la y.
- Para $n = 1$, hay que mover el disco de la torre x a la y ignorando la torre z.

Nótese que aquí los pasos de *Divide* y *Combina* se resumieron en uno sólo, pero si están presentes ambos.

El siguiente código muestra una implementación del algoritmo anterior, y utiliza como parámetros los nombres de las 3 torres, utilizando parámetros predeterminados como A, B y C.

Código 4.4: Torres de Hanoi

```
1 int hanoi(int n, char x='A', char y='C', char z='B'){
2         if(n==1){
3                 printf("Mueve de %c a %c.$\$n", x, y);
4         }else{
5                 hanoi(n-1, x, z, y);
6                 printf("Mueve de %c a %c.$\$n", x, y);
```

```
7                   hanoi(n-1, z, y, x);
8           }
9   }
```

□

Una leyenda cuenta que en la ciudad de Hanoi hay 3 postes así y unos monjes han estado trabajando para mover 64 discos del poste A al poste C y una vez que terminen de mover los 64 discos el mundo se acabará.

¿Te parecen pocos 64 discos?, corre la solución a este problema con n=64 y verás que parece nunca terminar(ahora imagina si se tardaran 2 minutos en mover cada disco).

Ejemplo 4.3.2. ¿Cuántas líneas imprime hanoi(n) (asumiendo que esta función está implementada como se muestra en el código 4.4)?

Solución Sea $H(n)$ el número de líneas que imprime $hanoi(n)$.

Es obvio que $H(1) = 1$ puesto que $hanoi(1)$ solamente imprime una línea.

Nótese que cada que se llama a $hanoi(n)$ se está llamando dos veces a $hanoi(n-1)$ una vez en la línea 5 y otra en la línea 7. Además, en la línea 6 imprime un movimiento.

Por lo tanto obtenemos la siguiente función recursiva:

$$H(1) = 1$$

$$H(n) = H(n-1) * 2 + 1$$

Ahora haremos unas cuantas observaciones para simplificar aún mas esta función recursiva:

$$\begin{aligned} H(1) &= 1 = 1 \\ H(2) &= 2 + 1 = 3 \\ H(3) &= 4 + 2 + 1 = 7 \\ H(4) &= 8 + 4 + 2 + 1 = 15 \\ H(5) &= 16 + 8 + 4 + 2 + 1 = 31 \\ H(6) &= 32 + 16 + 8 + 4 + 2 + 1 = 63 \\ &... \end{aligned}$$

Probablemente ya estés sospechando que

$$H(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0$$

Por inducción podemos darnos cuenta que como con $H(1)$ si se cumple y

$$(2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0)2 + 1 = 2^n + 2^{n-1} + 2^{n-2} + ... + 2^0$$

Entonces para cualquier número n la proposición se debe de cumplir.

O también puedes estar sospechando que:

$$H(n) = 2^n - 1$$

De nuevo por inducción

$$H(0) = 2^0 - 1$$

Y suponiendo que para alguna n, $H(n) = 2^n - 1$

$$(2^n - 1)(n) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

Por lo tanto, podemos concluir que la respuesta de este problema es sin lugar a dudas $2^n - 1$.□ Además de haber resuelto este problema pudimos darnos cuenta que

$$2^n - 1 = 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0 \tag{4.1}$$

Esta propiedad de la sumatoria de potencias de 2 hay que recordarla.

Capítulo 5

Búsqueda Exhaustiva

A veces parece que no hay mejor manera de resolver un problema que tratando todas las posibles soluciones. Esta aproximación es llamada Búsqueda Exhaustiva, casi siempre es lenta, pero a veces es lo único que se puede hacer.

También a veces es útil plantear un problema como Búsqueda Exhaustiva y a partir de ahí encontrar una mejor solución.

Otro uso práctico de la Búsqueda Exhaustiva es resolver un problema con con un tamaño de datos de entrada lo suficientemente pequeño.

La mayoría de los problemas de Búsqueda Exhaustiva pueden ser reducidos a generar objetos de combinatoria, como por ejemplo cadenas de caracteres, permutaciones(reordenaciones de objetos) y subconjuntos.

5.1. Cadenas

Ejemplo 5.1.1. Escribe una función que dados dos números enteros n y c, imprima todas las cadenas de caracteres de longitud n que utilicen solamente las primeras c letras del alfabeto(todas minúsculas), puedes asumir que $n < 20$.

Solución Llamemos $cadenas(n, c)$ al conjunto de cadenas de longitud n usando las primeras c letras del alfabeto.

Como de costumbre, para encontrar la solución a un problema recursivo pensaremos en el caso mas simple.

El caso en el que $n = 1$ y $c = 1$. En ese caso solamente hay que imprimir "a", o dicho de otra manera, $cadenas(1, 1) = \{"a"\}$

En este momento podríamos pensar en dos opciones para continuar el razonamiento cómo de costumbre:

$$n = 1 \text{ y } c > 1 \text{ o}$$

$$n > 1 \text{ y } c = 1$$

Si pensáramos en la segunda opción, veríamos que simplemente habría que imprimir las primeras n letras del abecedario.

Si pensáramos en la segunda opción, veríamos que simplemente habría que imprimir n veces "a" y posteriormente un salto de línea.

Ahora vamos a suponer que $n > 1$ y $c > 1$, para alguna n y alguna c, ¿sería posible obtener $cadenas(n, c)$ si ya se tiene $cadenas(n-1, c)$ ó $cadenas(n, c-1)$?

Si nos ponemos a pensar un rato, veremos que no hay una relación muy obvia entre $cadenas(n, c)$ y $cadenas(n, c-1)$, así que buscaremos la relación por $cadenas(n-1, c)$.

Hay que hacer notar aquí que si se busca una solución en base a una pregunta y ésta parece complicarse, es mejor entonces buscar la solución de otra forma y sólo irse por el camino complicado si no hay otra opción.

Volviendo al tema, buscaremos la relación de $cadenas(n, c)$ con $cadenas(n-1, c)$.

A veces algunos ejemplos sencillos pueden dar ideas. Observamos que

$$\begin{aligned} cadenas(1,3) = \{&a,\\ &b,\\ &c,\\ &\} \end{aligned}$$

$$\begin{aligned} cadenas(2,3) = \{&aa, ab, ac,\\ &ba, bb, bc,\\ &ca, cb, cc\\ &\} \end{aligned}$$

$$\begin{aligned} cadenas(3,3) = \{&aaa, aab, aac, aba, abb, abc, aca, acb, acc,\\ &baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc,\\ &caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc,\\ &\} \end{aligned}$$

Luego de observar esto, es posible prestar mas atención a la siguiente propiedad:

Toda cadena de n caracteres puede ser formada por una cadena de $n-1$ caracteres seguida de otro caracter.

Esto quiere decir que para cada caracter que pueda tener la cadena hay que generar todas las cadenas de $n-1$ caracteres y a cada cadena colocarle dicho caracter al final.

Partiendo de esta idea, que puede parecer un tanto complicada de implementar, se puede sustituir por la idea de primero colocar el ultimo caracter de la cadena, y luego generar todas las cadenas de $n-1$ caracteres posibles.

Para evitar meter muchos datos en la pila, es mejor tener la cadena guardada en un arreglo global de tipo char.

Código 5.1: Generador de Cadenas de Caracteres

```
1   char C[21];
2   void cadenas(int n, int c){
3   int i;
4   if(n==0){
5           printf("%s$\$n", C);
6   }else{
7           for(i='a';i<'a'+c;i++){
8                   C[n]=i;
9                   cadenas(n-1, c);
10          }
11  }
```

Ejemplo 5.1.2. ¿Cuántas cadenas de longitud n que utilizan solamente las primeras c letras del abecedario(todas minúsculas) existen? O dicho de otra forma ¿Cuántas líneas imprime el código 5.1?

Solución Llamémosle $cad(n,c)$ al número de líneas que imprime el código 11. Es obvio que $cad(1,c)$ imprime c líneas.

Nótese que si $n>1$, $cadenas(n,c)$ llama c veces a $cadenas(n-1,c)$. Por lo tanto

$$cad(1,c) = c$$

$$cad(n,c) = cad(n-1,c)c \text{ para } n>1$$

Ahora por inducción es fácil darnos cuenta que

$$cad(n,c) = c^n$$

□

Un algoritmo de cambio mínimo para generar cadenas binarias(de 0s y 1s) es aquel que genera todas las cadenas binarias de determinada longitud en un orden tal que cada cadena difiera de su predecesor en solamente un caracter.

El siguiente código es una implementación de un algoritmo de cambio mínimo que genera las 2^n cadenas binarias.

Código 5.2: Generador de cadenas binarias con cambio mínimo

```
1  void genera(int n){
2    if(n==0){
3              imprime_cadena();
4      }else{
5              genera(n-1);
6              C[i]=!C[i];
7              genera(n-1);
8      }
9  }
```

El código anterior genera las cadenas en un arreglo llamado C, y asume que siempre será lo suficientemente grande para generar todas las cadenas, la línea 3 llama a una función para imprimir la cadena generada, dicha función no se muestra porque ocuparía demasiado espacio y no tiene relevancia en el algoritmo.

Puede parecer un tanto confusa la línea 6, pero hay que recordar que !0 devuelve 1 y !1 devuelve 0.

Ejemplo 5.1.3. Demuestra que el código 5.2 genera todas las cadenas binarias y que cada cadena que genera difiere de la anterior en solamente un dígito.

Solución Este problema requiere que se comprueben 2 cosas: una es que el algoritmo genera todas las cadenas binarias de longitud n, y otra es que las genera con cambio mínimo.

Si $n = 1$ basta con echar un vistazo al código para darse cuenta que funciona en ambas cosas.

Podemos observar también, que luego de ejecutarse la línea 6 se llama a $genera(n-1)$ y se sigue llamando recursivamente a la función genera sin pasar por la línea 6 hasta que n toma el valor de 0 y se imprime la cadena en la línea 3; de esta forma podemos asegurar que genera las cadenas con cambio mínimo.

Para probar el hecho de que se imprimen todas las cadenas binarias de longitud n, hay que hacer la siguiente observación que no es muy obvia:

No importa si el arreglo C no está inicializado en 0s, siempre y cuando contenga únicamente 0s y 1s. La prueba de esto es que dada una cadena binaria de longitud n, se puede generar a partir de ella cualquier otra cadena binaria de longitud n, simplemente cambiando algunos de sus 1s por 0s y/o algunos de sus 0s por 1s. Si el algoritmo produce todas los conjuntos de cambios que existen entonces generará todas las cadenas sin importar la cadena inicial ya que a partir de una cadena se puede formar cualquier otra luego de aplicar una serie de cambios.

La capacidad de poder hacer este tipo de razonamientos tanto es muy importante, tan importante es poder reducir un problema como dividirlo.

De esa forma, suponiendo que para algún $n-1$ la función produce todas las cadenas de $n-1$ caracteres, ¿las producirá para n?.

Dado que la función no hace asignaciones sino solamente cambios(línea 6), podemos estar seguros que si se producen todas las cadenas llamando a la función con $n-1$ significa que el algoritmo hace todos los cambios para $n-1$.

También se puede observar que la línea 5 llama a $genera(n-1)$ con $C[n] = a$ para alguna $0 \leq a \leq 1$, y la línea 7 llama a $genera(n-1)$ con $C[n] =!a$ (si $a = 0$ entonces $C[n] = 1$ y si $a = 1$ entonces $C[n] = 0$).

Como $C[n]$ está tomando todos los valores posibles y para cada uno de sus valores esta generando todas las cadenas de tamaño $n-1$ podemos concluir que el algoritmo genera todas las cadenas binarias.

□

Hay que destacar una propiedad que se hayó en la solución:

Teorema 2. *Sea $cad(n, c)$ el número de cadenas que se pueden formar de longitud n con un alfabeto de c letras.*

$$cad(n, c) = c^n$$

5.2. Conjuntos y Subconjuntos

Los conjuntos son estructuras que estan presentes en todas las áreas de las matemáticas, y juegan un papel central en las matemáticas discretas y por ende en las ciencias de la computación.

La idea básica de un conjunto es una colección de elementos, para los cuales todo objeto debe pertenecer o no pertenecer a la colección.

Hay muchos ejemplos de la vida cotidiana de conjuntos, por ejemplo, el conjunto de los libros de una biblioteca, el conjunto de muebles de la casa, el conjunto de personas entre 25 y 30 años, etc.

Así como existen conjuntos tan concretos, las matemáticas(y por ende las ciencias de la computación) tratan por lo general con conjuntos más abstractos.

Por ejemplo el conjunto de los números entre 0 y 10, el conjunto de los números pares, el conjunto de los polígonos regulares, entre otros. De hecho casi todos los objetos matemáticos son conjuntos.

Los conjuntos, por lo general se describen con una lista de sus elementos separados por comas, por ejemplo, el conjunto de las vocales:

$$\{a, e, i, o, u\}$$

El conjunto de los números pares positivos de un solo dígito:

$$\{2, 4, 6, 8\}$$

Dado que un conjunto es una agrupación de elementos, no importa el orden en el que se escriban los elementos en la lista. Por ejemplo:

$$\{1, 5, 4, 3\} = \{4, 5, 1, 3\}$$

Los conjuntos suelen representarse con letras mayúsculas y elementos de los conjuntos con letras minúsculas.

Si un objeto a es un elemento de un conjunto P, entonces se dice que a pertenece a P y se denota de la siguiente manera:

$$a \in P$$

y si un objeto a no pertenece a P se denota de la siguiente manera

$$a \notin P$$

Por ejemplo

$$1 \in 3, 1, 5 \text{ pero } 1 \notin 3, 4, 2$$

En resumen:

Definición 5.2.1 (Conjunto)**.** Un conjunto es una colección o agrupación de objetos, a los que se les llama elementos. Los conjuntos, por lo general se describen con una lista de sus elementos separados por comas.

Si un objeto a es un elemento de un conjunto P, entonces se dice que a pertenece a P y se denota de la siguiente manera:

$$a \in P$$

y si un objeto a no pertenece a P se denota de la siguiente manera

$$a \notin P$$

Algunos conjuntos muy renombrados son:

- El conjunto de los números reales $\mathbb{R}$
- El conjunto de los números enteros $\mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$
- El conjunto de los números naturales $\mathbb{N} = \{0, 1, 2, 3, 4, 5, ...\}$

Respecto a este último conjunto, hay algunos matemáticos que aceptan el 0 como parte de los números naturales y hay matemáticos que no lo aceptan.

Por lo general los que no aceptan el 0 como parte de los números naturales es porque en teoría de números no se comporta como el resto de ellos, y además, el 0 fue un número desconocido para muchas civilizaciones.

Pero en las ciencias de la computación se suele aceptar al 0 como número natural, y no es por mero capricho, esto tiene una razón importante: tanto en el manejo de conjuntos como en la inducción, el 0 se comporta como número natural.

Se dice que un conjunto A es subconjunto de B, si todos los elementos de A también son elementos de B. O dicho de otra manera:

Definición 5.2.2 (Subconjunto)**.** Se dice que A es subconjunto de B si para todo elemento c tal que $c \in A$ se cumple que también $c \in B$ y se denota como $A \subseteq B$.

Por ejemplo:

El conjunto de las vocales es un subconjunto del conjunto del alfabeto.

El conjunto de los números pares es un subconjunto del conjunto de los números enteros.

$$\{a, b\} \subseteq \{b, c, d, a, x\}$$

$$\mathbb{N} \subseteq \mathbb{Z}$$

$$\mathbb{Z} \subseteq \mathbb{R}$$

Los subconjuntos son muy utilizados en la búsqueda exhaustiva, ya que muchos problemas pueden ser reducidos a encontrar un subconjunto que cumpla con ciertas características.

Ya sea para poder resolver un problema con búsqueda exhaustiva o para poder encontrar una mejor solución partiendo de una búsqueda exhaustiva es preciso saber cómo generar todos los subconjuntos.

Ejemplo 5.2.1. Supongamos que hay un arreglo global C que es de tipo entero. Escribe una función que reciba un entero n como parámetro e imprima todos los subconjuntos del conjunto de primeros n elementos del arreglo C.

Solución Este problema de generar todos los subconjuntos se puede transformar en el problema de generar cadenas de caracteres de una manera bastante sencilla:

Cada elemento de C puede estar presente o ausente en un subconjunto determinado.

Vamos a crear entonces un *arreglo de presencias* al que llamaremos P, es decir $P[i]$ será 0 si $C[i]$ está ausente, y será 1 si $C[i]$ está presente.

El problema entonces se reduce a generar todas las posibles cadenas binarias de longitud n en P. La técnica de expresar un problema en términos de otro es indispensable para resolver este tipo de problemas.

Código 5.3: Generación de subconjuntos

```
1  int imprime_subconjuntos(int n, int m=-1){
2          int i;
3          if(m<n){
4                  m=n;
5          }if(n==0){
6                  for(i=0;i<m;i++)
7                  if(P[i]==1)
8                          printf("%d ", C[i]);
9                  printf("\n");
10         }else{
11                 P[n-1]=0;
12                 imprime_subconjuntos(n-1, m);
13                 P[n-1]=1;
14                 imprime_subconjuntos(n-1, m);
15         }
16 }
```

El código anterior imprime todos los subconjuntos de C, se omite la declaración de P para no gastar espacio y se utiliza el parámetro m como el número de elementos de los que hay que imprimir subconjuntos para que no se confunda con el parámetro n que es el número de elementos de los que hay que generar los subconjuntos.

Nótese que si se llama a la función dejando el parámetro predeterminado para m, posteriormente m tomará el valor de n.

Ejemplo 5.2.2. Considera la misma situación que en el ejemplo 5.2.1, escribe una función que reciba como parámetros n y m, e imprima todos los subconjuntos de los primeros n elementos del arreglo C tal que cada subconjunto contenga exactamente m elementos.

Solución Como ya vimos en el ejemplo anterior, este problema se puede reducir a un problema de generación de cadenas binarias. Así que vamos a definir $P[i]$ como 1 si $C[i]$ esta presente y como 0 si $C[i]$ esta ausente del subconjunto.

Sea además $S(n, m)$ los subconjuntos de los primeros n elementos del arreglo tal que cada subconjunto tenga exactamente m elementos.

Por ello el problema se reduce a encontrar todas las cadenas binarias de longitud n que contengan exactamente m 1s.

Antes de encontrar el algoritmo para resolver esto es necesario hacer notar las siguientes propiedades:

- Si $m > n$ no hay cadena binaria que cumpla con los requisitos, ya que requiere tener exactamente m 1s, pero su longitud es demasiado corta(menor que m).
- Si $m = 0$ solo hay un subconjunto: el conjunto vacío.
- Si $n > 0$ y $n > m$ entonces $S(n, m)$ esta compuesto únicamente por $S(n-1, m)$ y por todos elementos de $S(n-1, m-1)$ añadiéndoles $C[n]$ a cada uno(véase Ejemplo 11).

Con estas propiedades ya se puede deducir el algoritmo.

Al igual que en el ejemplo anterior, utilizaremos un parámetro auxiliar al que llamaremos l que indicará de qué longitud era la cadena inicial. La intención es que a medida que se vayan haciendo las llamadas recursivas, el programa vaya formando una cadena que describa el subconjunto en P.

Dicho de otra manera definiremos una función llamada $imprime_subconjuntos(n, m, l)$, que generará todos los subconjuntos de tamaño m de los primeros n elementos del arreglo C. E imprimirá cada subconjunto representado en $P[0..n-1]$ junto con los elementos definidos en $P[n..l-1]$(los que ya se definieron en llamadas recursivas anteriores), es decir, imprimirá $P[0..l-1]$ para cada subconjunto.

Si $m > n$ entonces simplemente hay que terminar la ejecución de esa función pues no hay nada que imprimir.

Si $m = 0$ solamente queda el conjunto vacío, entonces solo hay que imprimir el subconjunto representado en $P[0..l-1]$.

En otro caso hay que asegurarse de que $C[n-1]$ este ausente en el subconjunto y llamar a $imprime_subconjuntos(n-1, m, l)$, esto generará todos los subconjuntos de los primeros n elementos del arreglo C, con exactamente m elementos, donde $C[n-1]$ no esta incluido.

Posteriormente, hay que poner a $C[n]$ como presente y llamar a $imprime_subconjuntos(n-1, m-1, l)$ para generar los subconjuntos donde $C[n]$ esta incluido.

Código 5.4: Generación de todos los subconjuntos de C con m elementos

```
1  void imprime_subconjuntos(int n, int m, int l=-1){
2          int i;
3          if(l<n)
4                  l=n;
5          if(m>n){
6                  return;
7          }if(m==0){
8                  for(i=0;i<l;i++)
9                          if(P[i]==1)
10                                 printf("%d ", C[i]);
11                 printf("$\$n");
12         }else{
13                 P[n-1]=0;
14                 imprime_subconjuntos(n-1, m, l);
15                 P[n-1]=1;
16                 imprime_subconjuntos(n-1, m-1, l);
17         }
18 }
```

Con este ejemplo cerramos el tema de subconjuntos.

5.3. Permutaciones

Así como a veces un problema requiere encontrar un subconjunto que cumpla con ciertas características, otras veces un problema requiere encontrar una secuencia de objetos que cumplan con ciertas características sin que ningún objeto se repita; es decir, una permutación.

Una permutación se puede definir como una cadena que no utiliza 2 veces un mismo elemento.

Las permutaciones, al igual que los conjuntos, suelen representarse como una lista de los elementos separada por comas pero delimitada por paréntesis. Sin embargo, a diferencia de los conjuntos, en las permutaciones el orden en el que se listan los elementos si importa. Por ejemplo, la permutación $(3, 2, 5)$ es diferente a la permutación $(5, 3, 2)$ y diferente a la permutación $(2, 3, 5)$.

Si se tiene un conjunto A, una permutación de A es una cadena que utiliza cada elemento de A una sola vez y no utiliza elementos que no pertenezcan a A.

Por ejemplo, sea A el conjunto (a, b, c, d), una permutación de A es (a, c, d, b) otra permutación es (d, a, c, b).

Ejemplo 5.3.1. Sea A un conjunto con n elementos diferentes. ¿Cuántas permutaciones de A existen?

Solución Un conjunto con un solo elemento o ningún elemento tiene solamente una permutación.

Si para algún n se sabe el número de permutaciones que tiene un conjunto de n elementos, ¿es posible averiguar el número de permutaciones con un conjunto de $n+1$ elementos?

Consideremos una permutación de un conjunto con n elementos(aqui a_i representa el i-ésimo número de la permutación):

$$(a_1, a_2, a_3, a_4, ..., a_n)$$

Suponiendo que quisiéramos insertar un nuevo elemento en esa permutación, lo podríamos poner al principio, lo podríamos poner entre a_1 y a_2, entre a_2 y a_3, entre a_3 y a_4, ... , entre a_{n-1} y a_n, o bien, al final. Es decir, lo podríamos insertar en $n+1$ posiciones diferentes.

Nótese entonces que por cada permutación de un conjunto de n elementos, existen $n+1$ permutaciones de un conjunto de $n+1$ elementos conservando el orden de los elementos que pertenecen al conjunto de n elementos.

Y también, si a una permutación de un conjunto con $n+1$ elementos se le quita un elemento, entonces queda una permutación de un conjunto con n elementos.

Sea $p(n)$ el numero de permutaciones de un conjunto con n elementos, podemos concluir entonces que $p(0) = 1$ y $p(n) = p(n-1)n$, esta recurrencia es exactamente igual a la función factorial.

Por ello el número de permutaciones de un conjunto con n elementos diferentes es $n!$.

Teorema 3. *El número de permutaciones de un conjunto con n elementos diferentes es* $n!$.

Ejemplo 5.3.2. Escribe un programa que dado n, imprima todas las permutaciones del conjunto de los primeros n números enteros positivos.

Solución Antes que nada tendremos un arreglo $p[] = \{1, 2, 3, ..., n\}$

Si $n = 0$ entonces solamente hay una permutación.

Si $n > 0$, hay que generar todas las permutaciones que empiecen con $p[0]$, las que empiecen con $p[1]$, las que empiecen con $p[2]$, las que empiecen con $p[3]$, ... , las que empiecen con $p[n-1]$(sobra decir a estas alturas que una permutación de un conjunto con n números es un solo número seguido de una permutación de un conjunto con $n-1$ números).

Esto se puede hacer recursivamente, ya que la función generará todas las permutaciones con n elementos, sin importar qué elementos sean. Es decir, si el programa intercambia el valor de $p[i]$ con el valor de $p[n-1]$ y manda a llamar a la función con $n-1$, la función generará todas las permutaciones de los primeros $n-1$ elementos con el antiguo valor de $p[i]$ al final. Así que hay que repetir este procedimiento de intercambiar y mandar llamar a la función para toda i entre 0 y $n-1$.

Así que el programa queda de esta manera:

Código 5.5: Generación de permutaciones

```
1  #include <stdio.h>
2  int *p;
3  int N;
4  void permutaciones(int n){
5          int i, aux;
6          if(n==0){ //Si n=0 imprime la permutacion
7                  for(i=0;i<N;i++)
8                          printf("%d ", p[i]);
9                  printf("\n");
10         }else{ //Sino, realiza los intercambios
               correspondientes y entra en recursion
11                 for(i=0;i<n;i++){
12                         aux=p[i]; //Intercambia los
                               valores de p[n-1] y p[i]
13                         p[i]=p[n-1];
14                         p[n-1]=aux;
15                         permutaciones(n-1); //Hace la
                               llamada recursiva
16                         aux=p[i]; //Pone los valores de
                               p[n-1] y p[i] en su lugar
17                         p[i]=p[n-1];
18                         p[n-1]=aux;
19                 }
20         }
21 }
22 int main(){
23         int i;
24         scanf("%d", \&N); //Lee el tamaño del conjunto
               de la entrada
25         p=new int[N];
26         for(i=0;i<N;i++) //Inicializa el conjunto
```

```
27                      p[i]=i+1;
28              permutaciones(N);//Ejecuta la recursion
29              return 0;
30      }
```

Nuevamente el código de la función es bastante corto aunque el resto del programa hace el código mas largo.

Parte II

Análisis de Complejidad

Durante la parte I se vieron muchas formas de aplicar la recursión en la resolución de problemas y algunas consecuencias matemáticas útiles de la recursión.

Sin embargo, los problemas mas interesantes en el diseño de algoritmos surgen al tomar en cuenta el análisis de complejidad, a estas alturas resulta muy difícil seguir avanzando sin contar con esta herramienta, y a diferencia de la recursión, el análisis de complejidad si requiere conocimientos previos para entenderse adecuadamente.

Durante un par de capítulos se hablará muy poco de programación, luego la programación volverá, pero tendremos que esperar para regresar al diseño de algoritmos ya que nos concentraremos en el análisis de algoritmos.

Quizá está curva de aprendizaje pudiera parecer desmotivante puesto que el objetivo es dominar el diseño de algoritmos, pero no hay que olvidar que los fundamentos no solo sirven para entender el análisis de complejidad, también pueden servir en el momento de resolver problemas.

Otra cosa que hay que tomar en cuenta, es que después de esta prolongada "ausencia" del diseño de algoritmos, el diseño de algoritmos regresará con una infinidad de nuevos problemas planteados y será posible avanzar mucho mas rápido.

Capítulo 6

Técnicas Básicas de Conteo

Como el nombre del capítulo lo indica, el tema a tratar son técnicas de conteo, las cuales son estudiadas por la combinatoria enumerativa; dichas técnicas, además de servir para resolver problemas que involucran el conteo, sirven ampliamente para determinar cuantas veces se ejecutan algunos pasos en los algoritmos.

A continuación veremos algunos ejemplos donde se aplican las reglas básicas de conteo para posteriormente identificarlas de manera objetiva.

Ejemplo 6.0.3. Hay 2 caminos que conducen de la ciudad A a la ciudad B, 3 caminos que conducen de la ciudad A a la ciudad C. Un camino que conduce de la ciudad B a la ciudad D y un camino que conduce de la ciudad C a la ciudad D ¿De cuántas formas se puede ir de la ciudad A a la ciudad D? (vease la figura para tenerlo mas claro).

Solución Para llegar a la ciudad D solamente se puede llegar a través de la ciudad C o de la ciudad B, por cada forma de llegar a la ciudad C hay

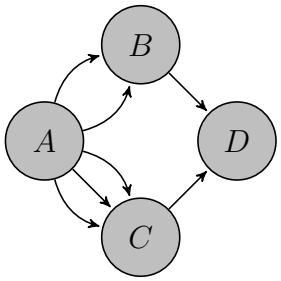


Figura 6.1: Ejemplo 6.0.3

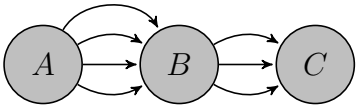


Figura 6.2: Ejemplo 6.0.4

una forma de llegar a la ciudad D y por cada forma de llegar a la ciudad B hay una forma de llegar a la ciudad D. Por lo tanto, el número de formas para llegar a la ciudad D iniciando en la ciudad A es la suma de las formas para llegar a la ciudad B y a la ciudad D, es decir $2 + 3 = 5$.

Ejemplo 6.0.4. Hay cuatro caminos que conducen de la ciudad A a la ciudad B, y tres caminos que conducen de la ciudad B a la ciudad C. ¿Cuántos caminos que pasan por B conducen hasta C?

Solución Vamos a llamar v_1, v_2, v_3 y v_4 a los caminos que van desde A hasta B y w_1, w_2, w_3 a los caminos que van desde la ciudad B hasta la ciudad C.

Para ir desde A hasta C pasando por B es necesario llegar hasta B usando alguno de los 6 caminos, y posteriormente elegir alguno de los tres caminos para ir desde B hasta C.

Es decir, existen los siguientes 12 caminos:

$$v_1, w_1$$

$$v_1, w_2$$

$$v_1, w_3$$

$$v_2, w_1$$

$$v_2, w_2$$

$$v_2, w_3$$

$$v_3, w_1$$

$$v_3, w_2$$

$$v_3, w_3$$

$$v_4, w_1$$

$$v_4, w_2$$

$$v_4, w_3$$

Aquí puede observarse que por cada camino que empieza con v_1 hay algun camino terminado en w_1, alguno terminado en w_2 y alguno terminado en w_3, por cada camino que empieza con v_2 también hay 3 caminos terminados en w_1, w_2 y w_3 respectivamente; lo mismo para v_3 y v_4. Por lo tanto hay $(4)(3) = 12$ caminos.

Ejemplo 6.0.5. Hay x caminos que conducen de la ciudad A a la ciudad B, y y caminos que conducen de la ciudad B a la ciudad C. ¿Cuántos caminos que pasan por B conducen hasta C?

Solución Por cada una de las x formas para ir de la ciudad A hasta la ciudad B existen y formas para ir de la ciudad B hasta la ciudad C. Por lo tanto existen xy formas para ir desde A hasta C pasando por B.

Código 6.1: Función f

```
1 int f(int A, int B){
2         int i, k, r=0;
3         for(i=0;i<A;i++){
4                 for(k=0;k<B;k++){
5                         r++;
6                 }
7         }
8         return r;
9 }
```

Ejemplo 6.0.6. ¿Cuál es el valor de retorno de la función f en el código anterior?

Solución Este problema se puede reducir en saber cuántas veces se ejecuta la línea 5.

Hay que hacer notar que la línea 3 se ejecuta A veces y por cada vez que se ejecuta la línea 3, la línea 5 se ejecuta B veces. Por lo tanto el número de veces que se ejecuta la línea 5(y por ende el valor de retorno de la función) es AB.

6.1. Reglas Básicas de Conteo

A pesar de que los problemas de combinatoria enumerativa son bastante variados, la mayoría pueden resolverse utilizando las llamadas **regla de la suma**, **regla del producto**, **biyeccion** y **recursión**.

Ya dedicamos batante tiempo a ver lo que es recursión, y además estas alturas ya deberías ser capás de utilizar la regla de la suma y el producto cuando te encuentres con un problema de combinatoria enumerativa, sin embargo, hay que identificarlas para poder justificar claramente nuestros razonamientos:

Teorema 4 (Regla de la Suma)**.** *Si cierto objeto x puede ser elegido de n maneras diferentes y otro objeto y puede ser elegido de n_1 maneras diferentes, entonces el número de formas de elegir x o y es $n + n_1$.*

Una manera mas precisa de decirlo, es que para dos conjuntos disjuntos A y B, el número de elementos que pertenecen a A o a B es $|A| + |B|$.

Aunque la regla anterior es bastante evidente tanto en su definición como en su aplicación, la siguiente regla es menos obvia de ver y aplicar que la anterior.

Teorema 5 (Regla del Producto)**.** *Si cierto objeto x puede ser elegido de n maneras diferentes y otro objeto y puede ser elegido de m maneras diferentes, entonces el número de formas de elegir x y posteriormente elegir y es nm.*

Una manera mas precisa de decirlo, es que para dos conjuntos A y B, el número de pares ordenados (a, b) tales que $a \in A$ y $b \in B$ es $|A||B|$.

Por ejemplo, si queremos formar palabras de longitud 2 únicamente con las letras a, b y c el número de formas de hacerlo es 9: aa, ab, ac, ba, bb, bc, ca, cb y cc. Si se quicieran formar palabras de longitud 3 únicamente con las letras a, b y c el número de formas de hacerlo sería $3^3 = 27$.

Una aplicación computacional es la siguiente: ¿ Alguna vez te has preguntado por qué en C y C++ los enteros con signo soportan valores desde -2147483648 hasta 2147483647 y qué tiene que ver eso con que sean tipos de datos de 32 bits?.

Cada bit es un dígito que puede ser solamente 0 o 1 y el conjunto de dígitos binarios que sean iguales a 1 determina el número almacenado. Por ello hay 32 dígitos y cada uno puede ser elegido de dos maneras, por regla del producto la cantidad de números que pueden ser almacenados en 32 bits es: $2^{32} = 4294967296$, y ahora, un *int* puede tener cualquiera de $2147483647 - (-2147483648) + 1 = 4294967296$ valores diferentes y por ese mismo motivo los enteros sin signo(también de 32 bits) soportan valores desde 0 hasta 4294967295.

La unica regla que falta es la de las biyecciones; su uso también es bastante natural, sin embargo la forma de utilizarlo involucra bastante creatividad y es mas complicado de definir.

La idea básica de la biyección es la de contar algo distinto a lo que inicialmente se quiere contar y luego demostrar que lo que se contó es del mismo *tamaño* que lo que inicialmente se quería contar.

Por ejemplo, para contar cuantos subconjuntos tiene un conjunto de n elementos, contamos cuantas cadenas hay de longitud n tales que contengan solamente 0s y 1s.

La manera de demostrar que contar el número de subconjuntos equivale a contar el número de cadenas con 0s y 1s se basa en decir que a cada cadena le corresponde un único subconjunto y viceversa.

Es decir, dos conjuntos A y B tienen la misma cantidad de elementos, si es posible asignarle a cada elemento de A una única pareja en B, y a cada elemento en B asignarle una única pareja en A. Al conjunto de todas las parejas se le conoce como biyección.

Siendo mas precisos:

Teorema 6 (Regla de la Biyección)**.** *Si para dos conjuntos A y B existe un conjunto de pares ordenados P tal que para cada $a \in A$ existe un único $b \in B$ tal que $(a, b) \in P$ y para cada $b \in B$ existe un único $a \in A$ tal que $(a, b) \in P$ entonces se dice que P es una biyección entre A y B y además $|A| = |B|$.*

6.2. Conjuntos, Subconjuntos, Multiconjuntos

Ya en la sección 5.2 habíamos definido conjuntos y subconjuntos, aquí exploraremos algunas de sus propiedades y definiremos multiconjunto.

La siguiente propiedad es fundamental en las tecnicas de conteo:

Teorema 7 (Número de Subconjuntos)**.** *Si un conjunto C tiene n elementos, el número de subconjuntos de C es exactamente 2^n.*

Demostración. Cada uno de los n elementos puede estar dentro o fuera del subconjunto, nótese que para todo subconjunto $S \subset C$ existe un subconjunto $T \subset C$ tal que S y T no tienen elementos en común y cada elemento de C está o bien en S o en T.

De esta manera, todo elemento de C puede ser elegido de dos formas: para formar parte de S o para formar parte de T y como C tiene n elementos, por regla del producto se concluye que C tiene exactamente 2^n subconjuntos.

□

Ahora vamos a introducir un nuevo concepto, se trata del concepto de los multiconjuntos.

La idea de un multiconjunto es poder representar un conjunto con elementos repetidos, ya que para modelar ciertos sistemas se pueden tener elementos

que compartan las mismas propiedades; por ejemplo en el ajedrez, todos sabemos que cada jugador cuenta inicialmente con 16 piezas, sin embargo algunas de esas piezas son idénticas y con ellas se podría hacer exactamente lo mismo; otro ejemplo interesante puede ser el dinero que se traiga en la cartera, puede haber varias monedas que en la práctica son exactamente iguales.

Asi que de manera general, un multiconjunto es un *conjunto con elementos que se pueden repetir*. Pero esta claro que esta definición aunque es bastante comprensible y dice mucho sobre la aplicación de los multiconjuntos es inadmisible de manera formal, por lo que se ha construido esta otra definición:

Definición 6.2.1 (Multiconjunto)**.** Un multiconjunto se define como el par (C, f) donde C es un conjunto y f es una función tal que a cada elemento de C le asigna un número entero positivo.

El proposito de f esta definición es decir cuántas veces aparece cada elemento en el multiconjunto. Lo que ahora necesitamos tener es un análogo a los subconjuntos pero en los multiconjuntos, el cual llamaremos submulticonjunto.

Nuestro sentido común nos dice que si queremos definir un submulticonjunto S de un multiconjunto A entonces todos los elementos de S deben de aparecer en A, y además, es inadmisible que un elemento de S aparezca mas veces en S que en A.

Mas formalmente:

Definición 6.2.2 (Submulticonjunto)**.** Sea $A = (C, f)$ un multiconjunto, $S = (D, g)$ es un submulticonjunto de A si $D \subseteq C$ y además para todo $d \in D$ se cumple que $g(d) \leq f(d)$. Y se denota como:

$$S \subseteq A$$

Lo siguiente que nos debemos preguntar es ¿cuántos submulticonjuntos tiene un multiconjunto finito $A = (C, f)$?. Para contarlos procederemos de la misma manera que para contar los subconjuntos.

Es decir, cada elemento $c \in C$ puede estar desde 0 hasta $f(c)$ veces en el submulticonjunto. De esta manera cada elemento de C puede ser elegido de $f(c) + 1$ formas.

Usando regla del producto concluimos que el número de submulticonjuntos de A es $(f(a_1) + 1)(f(a_2) + 1)(f(a_3) + 1)...(f(a_n) + 1)$ donde $C = \{a_1, ..., a_n\}$.

Una observación interesante es que según esa *fórmula* si cada elemento de A aparece una sola vez, es decir si $f(a_i) = 1$ para toda i entonces el número

de submulticonjuntos es 2^n, lo cual coincide con nuestras observaciones en los conjuntos.

6.3. Permutaciones

Ya se habían mencionado las permutaciones, sin embargo, solamente se definieron de manera que no pudiera haber elementos repetidos. Aquí se verán las permutaciones de una forma más general.

Definición 6.3.1 (Permutación)**.** Una permutación es un reacomodo de objetos o símbolos en secuencias diferentes.

Las permutaciones, al igual que los conjuntos, suelen representarse como una lista de los elementos separada por comas pero delimitada por paréntesis. Sin embargo, a diferencia de los conjuntos, en las permutaciones el orden en el que se listan los elementos si importa.

Por ejemplo, hay 6 permutaciones del conjunto $C = \{X, Y, Z\}$:

$$(X, Y, Z)$$
$$(X, Z, Y)$$
$$(Y, X, Z)$$
$$(Y, Z, X)$$
$$(Z, X, Y)$$
$$(Z, Y, X)$$

Pero solamente hay 3 permutaciones del multiconjunto $C = \{A, A, B\}$:

$$(A, A, B)$$
$$(A, B, A)$$
$$(B, A, A)$$

Las permutaciones de multiconjuntos se conocen como **permutaciones con repetición**.

Ya habíamos visto la siguiente propiedad pero vale la pena volverlo a incluir aquí ya que es un principio fundamental de conteo.

Teorema 8 (Número de permutaciones sin repetición)**.** *Un conjunto con n elementos tiene exactamente $n!$ permutaciones.*

Demostración. Un conjunto con un solo elemento o ningún elemento tiene solamente una permutación.

Si para algún n se sabe el número de permutaciones que tiene un conjunto de n elementos, ¿es posible averiguar el número de permutaciones con un conjunto de n+1 elementos?

Consideremos una permutación de un conjunto con n elementos(aqui a_i representa el i-ésimo número de la permutación):

$$(a_1, a_2, a_3, a_4, ..., a_n)$$

Suponiendo que quisiéramos insertar un nuevo elemento en esa permutación, lo podríamos poner al principio, lo podríamos poner entre a_1 y a_2, entre a_2 y a_3, entre a_3 y a_4, ... , entre a_{n-1} y a_n, o bien, al final. Es decir, lo podríamos insertar en $n+1$ posiciones diferentes.

Nótese entonces que por cada permutación de un conjunto de n elementos, existen $n+1$ permutaciones de un conjunto de $n+1$ elementos conservando el orden de los elementos que pertenecen al conjunto de n elementos.

Y también, si a una permutación de un conjunto con $n+1$ elementos se le quita un elemento, entonces queda una permutación de un conjunto con n elementos.

Sea $p(n)$ el numero de permutaciones de un conjunto con n elementos, podemos concluir entonces que $p(0) = 1$ y $p(n) = p(n-1)n$, esta recurrencia es exactamente igual a la función factorial.

Por ello el número de permutaciones de un conjunto con n elementos diferentes es $n!$.

□

Ejemplo 6.3.1. Para una empresa se requiere que se ocupen los siguientes cargos: presiente, vicepresidente, secretario, vicesecretario, barrendero. Solamente puede(y debe) haber un presidente, un vicepresidente, un secretario, un vicesecretario pero no hay límite de barrenderos. Si 6 personas van a ocupar cargos ¿de cuántas formas pueden ocuparlos?.

Solución Dado que hay 5 cargos diferentes y 4 de ellos son únicos, el número de barrenderos es $6 - 4 = 2$.

Primero veremos de cuantas formas se pueden elegir a los barrenderos. Si etiquetamos a un cargo de barrendero como A y a otro cargo como B, entonces podemos elegir de entre 6 personas al barrendero A y luego podemos elegir de entre las 5 restantes al barrdendero B, eso da un total de $(6)(5) = 30$ formas de elegirlos, pero dado que si nos olvidamos de las etiquetas A y B, el órden en el que se eligen en realidad no importa, veremos que cada forma

de elegirlos se está contando dos veces, por lo que el número de formas de elgir a los barrenderos es $(6)(5)/2 = 15$.

Una vez elegidos a los barrenderos quedan 4 cargos diferentes y 4 personas diferentes, nótese que por cada permutación de las personas existe una manera de asignarle los cargos(a la primer persona presidente, a la segunda vicepresidente, a la tercera secretario y a la cuarta vicesecretario). Por ello el número de formas de asignar 4 cargos diferentes a 4 personas diferentes es 4!=16.

Por regla del producto el número total de formas de asignar los puestos de trabajo es (15)(16) o dicho de otra forma:

$$(\frac{6!}{4!2!})((6-2)!) = \frac{6!}{2!}$$

Ejemplo 6.3.2. Para una empresa se requiere que se ocupen los siguientes cargos: presiente, vicepresidente, secretario, vicesecretario, barrendero. Solamente puede(y debe) haber un presidente, un vicepresidente, un secretario, un vicesecretario pero no hay límite de barrenderos. Si 8 personas van a ocupar cargos ¿de cuántas formas pueden ocuparlos?.

Solución El problema es exactamente el mismo que el del ejemplo anterior, solamente que ahora el número de barrenderos es $8 - 4 = 4$.

Si los 4 puestos de barrenderos fueran todos diferentes entre sí entonces está claro que el número de formas de asignar los cargos sería 8!.

Pero por cada manera de asignar 8 personas a 8 cargos diferentes existen $(8-4)! = 4!$ maneras de asignar 8 personas a 4 cargos diferentes y un cargo repetido 4 veces. Esto es porque las personas que estuvieran ocupando los cargos de barrenderos serían 4 y podrían reordenar de 4! formas.

De esta manera la solución es:

$$\frac{8!}{4!}$$

Ejemplo 6.3.3. Un restaurant requiere 3 meseros, 4 cocineros y 4 barrenderos, si 20 personas quieren trabajar y estan igualmente capacitadas para ocupar cualquiera de los cargos, ¿de cuantas formas pueden ocuparlos?

Solución Primero que nada intentaremos transformar este problema a algo parecido al problema anterior, debido a que es mas fácil tratar con un problema previamente resuelto.

El número de personas que ocuparán al menos uno de los cargos es un total de $3 + 4 + 4 = 11$, por lo que 9 personas se quedarán con las ganas

de trabajar(pero sin poder hacerlo). Así que para resolver este problema imaginaremos un nuevo cargo en el cual se quedarán las 9 personas que no consigan ninguno de los otros.

Si las 20 personas se acomodaran en una fila sería posible asignarles a los primeros 3 el cargo de meseros, a los siguientes 4 el cargo de cocineros, a los siguientes 4 el cargo de barrenderos y a los ultimos 9 nuestro cargo imaginario; en efecto, si se quieren asignar cargos de alguna manera, sin importar la que sea, siempre es posible ordenar a las 20 personas en una fila de tal forma que se asignen los cargos de manera deseada utilizando el criterio que ya se mencionó.

Pero aún existen varias formas de ordenar a la fila que producen la misma asignanción de empleos. Por ahora solo sabemos que el número de formas de ordenar la fila es 20!.

Ahora, para cualquier manera de ordenar la fila, es posible reordenar a los meseros de 3! formas distintas, es posible reordenar a los cocieneros de 4! formas distintas, es posible reordenar a los barrenderos de 4! formas distintas y además es posible reordenar a los que ocupan el cargo imaginario de 9! formas diferentes y todo esto produciendo la misma asignación de empleos; por ejemplo, si se cambian de órden las primeras 3 personas de la fila sin mover a las demás las 3 personas seguirán siendo meseros y las demás personas seguirán ocupando los mismos cargos.

Por regla del producto, para cualquier manera de ordenar la fila, es posible reordenarla sin cambiar la asignación de empleos de (3!)(4!)(4!)(9!) formas distintas.

Por lo que el número total de maneras de asignar los empleos es:

$$\frac{20!}{3!4!4!9!}$$

□

Viendo la solución del ejemplo anterior, se puede encontrar fácilmente una demostración análoga del siguiente teorema, el cual resulta obvio después de ver la solución del ejemplo anterior:

Teorema 9 (Permutaciones con Repetición)**.** *Sea $M = (C, f)$ un multiconjunto donde $C = c_1, c_2, ..., c_n$ son los elementos que posee y $f(c_i)$ es el número de veces que se encuentra cualquier elemento c_i en M. El número de permutaciones del multiconjunto M es:*

$$P^n_{f(c_1)f(c_2)...f(n)} = \frac{n!}{f(c_1)!f(c_2)!...f(n)!}$$

6.4. Combinaciones

Como ya habíamos bisto en la Parte I, las combinaciones, o coeficientes binomiales son el número de subconjuntos con exactamente k elementos escogidos de un conjunto con exactamente n elementos para una k y una n dadas. Y se denota de la siguiente manera:

$$\binom{n}{k}$$

Ya habíamos se había visto en la Parte I cómo calcular las combinaciones utilizando el triángulo de Pascal. Pero es conviente conocer una fórmula cerrada para calcularlas y no solamente una función recursiva.

Recordando los ejemplos de la sección de permutaciones con repetición, podemos transformar este problema a tener una empresa con 2 puestos de trabajo uno con k plazas y otro con $n-k$ plazas y encontrar el número de formas en las que n trabajadores pueden ocupar todos los puestos. O dicho de otra manera, encontrar el número de formas de ordenar k elementos de un tipo con $n-k$ elementos de otro tipo.

Teorema 10 (Combinaciones de n en k)**.**

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

6.5. Separadores

Muchos problemas de combinatoria se resuelven agregando al planteamiento unos objetos llamados **separadores**, para los cuales existe una biyección entre lo que se quiere contar y ellos. Es difícil ilustrar de manera general lo que son los separadores. Así que se mostrará en ejemplos como usarlos.

Ejemplo 6.5.1. Se tienen 20 canicas idénticas y se quieren guardar en 3 frascos diferentes sin importar si uno o dos de los frascos quedan vacíos. ¿De cuántas formas se puede hacer esto?

Solución En lugar de imaginar 20 canicas, imagina 20 guiones alineados en una recta de la siguiente manera:

- - - - - - - - - - - - - - - - - - - -

Ahora, vamos a insertar 2 bárras verticales entre los guiones. A dichas barras verticales les llamaremos separadores.

Una posible manera de insertarlas sería así:

- - - - - - - - - - | - - - | - - - - - - -

Lo anterior puede ser interpretado como poner 10 canicas en el primer frasco, 3 canicas en el segundo frasco y 7 canicas en el tercer frasco, dicho de otra manera, los guiones anteriores al primer separador se ponen en el primer frasco, los guiones entre el primero y segundo separador se ponen en el segundo frasco y los guiones posteriores al segundo separador se ponen en el tercer frasco.

Asi que por cada manera de insertar dos separadores entre un total de 20 guiones existe una manera de guardar 20 canicas identicas en un total de 3 frascos diferentes y viceversa, es decir ¡Hay una biyección entre el número de separadores y el número de formas de repartir las caincas!.

Además, el número de formas de poner los dos separadores, es el número de permutaciones con repetición de 20 objetos de un tipo y 2 objetos de otro tipo, lo cual es equivalente a:

$$P_{20,2}^{20+2} = \frac{22!}{20!2!} = \binom{22}{2}$$

Ejemplo 6.5.2. ¿De cuántas formas se pueden formar en una fila 20 marcianos diferentes y 7 jupiterianos diferentes de manera que no haya ningún par de jupiterianos adyacentes?

Consideremos primero el caso donde hay 20 marcianos idénticos y 7 jupiterianos idénticos.

Queremos saber el número de formas de dividir a 20 marcianos en 8 grupos de manera que en cada uno de los grupos haya al menos un marciano. Y luego el primer grupo se puede colocar frente al primer jupiteriano, el segundo grupo frente al segundo jupiteriano, y asi sucesivamente, quedando el último grupo detras del último jupiteriano.

Para hacer eso primero hay que asignarle un marciano a cada grupo, quedando 12 marcianos disponibles para repartirse en 8 grupos; el número de formas de hacerlo se puede calcular añadiendo 7 separadores a los 12 marcianos, con lo cual resultan $\binom{12+7}{7}$ formas de hacerlo.

Ya calculamos el número de maneras de ordenarlos si fueran identicos, pero dado que son diferentes, por cada manera de ordenarlos sin considerar las diferencias marciano-marciano y jupiteriano-jupiteriano existen 20!7! maneras diferentes de reordenarlos sin cambiar el tamaño de ninguno de los 8

grupos. Por lo tanto, el número de formas en las que se pueden acomodar 20 marcianos diferentes y 7 jupiterianos diferentes en una fila sin que haya dos jupiterianos juntos es:

$$\binom{19}{7}(20!)(7!)$$

Ejemplo 6.5.3. Si 6 caballos estan compitiendo en una carrera, ¿de cuántas formas pueden llegar a la meta? (dos formas se consideran diferentes solo si el órden en el que llegan a la meta es diferente? (nota: puede haber empates).

Solución Es obvio que si no hubiera empates solamente podrían llegar a la meta de $6! = 720$ formas, pero también se deben de considerar otros casos.

En general puede haber desde 1 hasta 6 grupos diferentes y en cada grupo un conjunto de caballos que empatan en la carrera, y por supuesto, cada grupo debe de tener al menos un caballo.

Por regla de la suma sabemos que la solución será el número de formas con un grupo de empates más número de formas con 2 grupos de empate, más número de formas con 3 grupos de empate, ... , más número de formas con 6 grupos de empates.

Ya se vió en el ejemplo anterior que el número de maneras para repartir n objetos identicos entre m grupos diferentes de manera que en cada grupo hubiera al menos un objeto es lo mismo que el número de maneras de repatir $n-m$ objetos entre m grupos sin importar si algunos grupos se quedan vacíos. Por ello el número de formas en las que 6 caballos identicos pueden llegar a la meta con i grupos de empate es el número de formas de colocar $6-i$ objetos identicos con $i-1$ separadores:

$$\binom{(6-i)+(i-1)}{i-1} = \binom{5}{i-1}$$

Y como el número de permutaciones de cualquier conjunto con 6 elementos es 6! y por regla del producto el número de formas en las que 6 caballos diferentes pueden llegar a la meta con i grupos de empate es:

$$\binom{5}{i-1}(6!)$$

Por lo tanto, aplicando la regla de la suma, el número de formas en las que 6 caballos pueden llegar a la meta es:

$$\binom{5}{0}(6!) + \binom{5}{1}(6!) + \binom{5}{2}(6!) + \binom{5}{3}(6!) + \binom{5}{4}(6!) + \binom{5}{5}(6!)$$

$$6!(\binom{5}{0} + \binom{5}{1} + \binom{5}{2} + \binom{5}{3} + \binom{5}{4} + \binom{5}{5})$$

$$6!(2^5)$$

Capítulo 7

Funciones

Las funciones son un tema fundamental dentro de las ciencias de la computación al igual que en el resto de las matemáticas; son necesarias para entender correctamente la mayoría de los algoritmos y en ellas se centra el análisis de complejidad.

El concepto de función muy comúnmente es malentendido o usado de una manera incorrecta. A veces se tiene la creencia de que una función necesariamente se puede expresar como una ecuación o que todos los valores de las funciones son conocidos; y para los programadores es mas común pensar en las funciones como máquinas que reciben parámetros y devuelven valores.

Para ciertas aplicaciones son útiles esas análogías, pero para lo que viene después se requiere entender perfectamente lo qué es una función y cómo aparecen de manera natural mientras se resuelven problemas.

No es de sorprender entonces que se les dedique un capítulo a las funciones, ya que en los próximos capítulos estaremos trabajando con ellas y resulta razonable dedicar un tiempo a definir realmente con qué estamos trabajando.

7.1. Las Funciones como Reglas

Como se decía unos párrafos atrás, las funciones muchas veces son vistas como máquinas que reciben parámetros y devuelven valores; aunque esta forma de verlas es útil en la programación, es muy limitada en la resolución de problemas. Una forma parecida de ver las funciones es como *reglas*, es decir, una función puede ser vista como una regla que a cada elemento de un conjunto X le asigna un elemento(y solo uno) de un conjunto Y.

Los siguientes ejemplos sirven para ilustrar mejor esta manera de ver las funciones:

1. La regla que le asigna a cada número real su mitad.

2. La regla que le asigna a cada número entero positivo n su sumatoria descrita por:
$$\frac{(n)(n+1)}{2}$$

3. La regla que le asigna una CURP a cada ciudadano.

4. La regla que le asigna a cada par de números a y b su promedio $\frac{a+b}{2}$.

5. La regla que le asigna $\sqrt{2}$ a 53, $\frac{1}{2}$ a 42 y π a $\frac{3}{4}$.

6. La regla que le asigna un tiempo de ejecución a cada conjunto de entradas en un programa.

Con estos ejemplos debe de quedar claro que una función puede ser cualquier regla que le asigne elementos de un conjunto Y a los elementos de un conjunto X, y "cualquier regla" se refiere a que las reglas pueden no ser descritas por alguna ecuación o expresión algebrárica, pueden no estar definidas en todos los números, pueden incluso no estar definidas en números(como en el ejemplo 3), y es posible que en algunos casos(como en el ejemplo 5) no se conozca a qué números u *objetos* se puede aplicar.

Si una función le asigna un elemento de un conjunto Y a cada elemento de un conjunto X entonces se dice que el dominio de la función es de dominio X y la codominio Y.

Aunque no es un requisito, las funciones se suelen designar por la letra f, y cuando hay mas de una función se suelen usar también las letras g y h. Si f es una función, x una variable tal que $x \in X$, entonces el valor que f asocia con x se expresa como $f(x)$ y se lee como "f de x".

La función del primer ejemplo se puede expresar como:

$$f(x) = \frac{x}{2} \text{ para todo número real } n$$

La del segundo ejemplo se puede definir como

$$f(n) = \frac{(n)(n+1)}{2} \text{ para todo entero positivo } n$$

El tercer ejemplo no se puede definir completamente con esta notación, ya que no hay expresión algebrarica que represente una CURP o un ciudadano, y tampoco conocemos un procedimiento para encontrar la CURP; solamente sabemos que el dominio es el conjunto de ciudadanos y la imagen es el conjunto de las CURPs.

Por este motivo es necesario definir mas notación. Si f es una función, D su dominio y C su codominio, se puede denotar de la siguiente manera:

$$f : D \longrightarrow C$$

Así ya podemos expresar el tercer ejemplo como una función:

$f : D \longrightarrow C$ donde D es un conjunto de ciudadanos y C es un conjunto de CURPs

El cuarto ejemplo tambien puede resultar peculiar, ya que muchas veces se espera que el dominio de una función sea un solo número y no dos. Para ser precisos el dominio de la función del cuarto ejemplo es el conjunto de todos los pares órdenados de números reales (a, b), el cual se representa como $\mathbb{R}^2$.

Asi que el cuarto ejemplo se puede describir así:

$$f(a, b) = \frac{a+b}{2} \text{ para todo par de números (a, b) } \in \mathbb{R}^2$$

Para el quinto ejemplo no hay mas remedio que enumerar los valores conocidos:

$$\begin{aligned} f(x) &= \sqrt{2}, \text{ si } x = 53 \\ &= \frac{1}{2}, \text{ si } x = 42 \\ &= \pi, \text{ si } x = \frac{3}{4} \end{aligned}$$

El sexto ejemplo habla de una función que corresponde al tiempo que tarda en ejecutarse un programa con determinada entrada; esta función cobrará mucha importancia mas adelante y se le conoce como la función de tiempo.

Después de usar mucho tiempo las funciones, es natural comenzar a pensar en abreviaturas, como con el primer ejemplo "$f(x) = \frac{x}{2}$ para todo número real n" podría parecernos muy largo, así que muchas veces por practicidad se omite la descripción del dominio de la función y se deberá asumir que el dominio es el conjunto de todos los números reales para los cuales la función tiene sentido.

Por ejemplo para la función $f(x) = \frac{1}{x}$ se sobreentiende que el dominio es todos los números reales excepto el 0. Sin embargo existen muchos intentos por abreviar aún más esta notación pero la mayoría resultan inadecuados.

Por ejemplo sustituir $f(x) = \frac{x}{2}$ por $\frac{x}{2}$ equivaldría a hablar de un número real en lugar de una función; la única manera aceptable de abreviar esto

aún más es como $x \to \frac{x}{2}$, esto no representa una gran mejoría en cuanto a la abreviación y en ocaciones puede parecer menos claro, la única ventaja que parece tener esta notación es que no hay que asignarle un nombre a la función y puede servir cuando se esté trabajando con muchas funciones y no sea necesario nombrar a todas.

Luego de ver estos ejemplos es clara la ventaja de ver las funciones como *reglas* y no solamente como *máquinas*, ya que es mas claro imaginar una regla implícita que una máquina implícita, y no queda claro que una máquina simpre devuelva la misma salida para una entrada dada.

7.2. El Concepto Formal de Función

A pesar de que se pueden inferir muchas cosas imaginando a las funciones como *reglas*, el concepto de *regla* es subjetivo, es preferible definir algo en base a cosas ya conocidas y con propiedades bien definidas.

La palabra *regla* puede significar distintas cosas en distintos contextos. Algo que sabemos muy bien cómo se comportan son los conjuntos y los pares ordenados. Definiremos una función a partir de esto para que no quede duda de qué es ó cómo se comporta una función.

Primero que nada necesitamos definir un concepto bastante importante en muchas áreas de las matemáticas, y este es el del *producto cartesiano*, este concepto se refiere al conjunto de todos los pares ordenados formados por elementos de dos conjuntos dados.

Por ejemplo, el producto cartesiano de $\mathbb{Z}$ y $\mathbb{Z}$ son todos los puntos del plano cartesiano cuyas cordenadas son enteras.

El producto cartesiano de $\{1, 2\}$ y $\{3, 4\}$ es $\{(1, 3), (1, 4), (2, 3), (2, 4)\}$, nótese que $(1, 3)$ si es parte del producto cartesiano pero $(3, 1)$ no lo es, por ello decimos que el producto cartesiano es **no conmutativo**.

El hecho de que el producto cartesiano sea **no conmutativo** hace que este concepto sea aplicable a gran cantidad de cosas y algunas de ellas no estan muy relacionadas con las ciencias de la computación, como lo es el producto cartesiano de los nombres y los apellidos, con el cual se obtiene el conjunto de todos los nombres completos válidos.

Definición 7.2.1 (Producto Cartesiano)**.** El producto cartesiano de dos conjuntos A y B es el conjunto de todos los pares ordenados (a, b) tal que $a \in A$ y $b \in B$, y se denota como:

$$A \times B$$

A cualquier subconjunto de $A \times B$ se le denomina **relación** de A en B, el concepto de relación es muy importante, ya que casi todo lo que estudian las matemáticas(independientemente de las otras propiedades que tengan) son relaciones.

Por ejemplo el conjunto de todos los pares de números (a, b) tal que b es multiplo de a es una relación de $\mathbb{Z}$ en $\mathbb{Z}$ y se le llama divisibilidad. El conjunto de los pares de números (a, b) tales que $a < b$ son la relación "menor que"; incluso las funciones son relaciones.

Definición 7.2.2 (Función)**.** Una función f con dominio en A y codominio en B es un subjunto de $A \times B$ tal que:

$$\text{Si } (a, b) \in f \text{ y } (a, c) \in f \text{ entonces } b = c$$

$$\text{Si } a \in A \text{ entonces existe } b \in B \text{ tal que } (a, b) \in f$$

Admitimos las siguiente notación:

- Una función f con dominio en A y codominio en B se denota como $f : A \longrightarrow B$.
- $(a, b) \in f$ se denota como $f(a) = b$.
- $f(a)$ es aquel número tal que $(a, f(a)) \in f$.
- $x \to f(x)$ se refiere al conjunto de todos los pares ordenados $(x, f(x))$ para todo x en el dominio de f.

Muchas veces hay que tener presente la definición anterior, sin embargo, sigue siendo mas útil para resolver problemas el hecho de ver una función como una regla.

En general este tipo de definiciones sirven para aclarar aspectos oscuros respecto a lo que puede ser y lo que no puede ser una función evitando así llegar a contradicciones.

Capítulo 8

Análisis de Complejidad

Comunmente se suele pensar que la computadora efectúa sus operaciones con una velocidad infinitamente rápida y con una capacidad de almacenamiento infinita. Y esto es natural, ya que una computadora puede realizar millones de operaciones cada segundo así como almacenar información que no cabría ni en 100 bibliotecas.

Sin embargo, en las matemáticas existen funciones cuyos valores crecen a una velocidad impresionante; por ejemplo, $f(x) = 2^x$, mientras valores como $f(8) = 256, f(9) = 512, f(10) = 1024$ son relativamente pequeños, tenemos que $f(30) = 1099511627776$, y si decimos que $f(x)$ representa el número de sumas que debe realizar un programa, nos encontramos con que si $x = 29$ el programa tardaría medio segundo en correr(en una computadora con 1.5 Ghz), si $x = 30$ el programa tardaría un segundo, si $x = 31$ el programa tardaría 2 segundos, si $x = 42$ el programa tardaría mas de una hora, si $x = 59$ el programa tardaría 182 años en ejecutarse y si $x = 100$ el programa tardaría en ejecutarse casi 31 mil veces la edad del universo.

Por todo esto, ni siquiera las computadoras se "salvan" de tener que limitar el número de operaciones que realizan; pero la manera en que las computadoras se comportan es muy diferente a la manera en la que nos comportamos los humanos.

Mientras que un humano puede comenzar a resolver un problema de cierta manera y luego darse cuenta de que existen formas más rápidas de hacerlo, una computadora no, la computadora ejecutará el algoritmo para el que fue programada de principio a fin; es por eso que los humanos necesitamos saber qué tan rápidos son los algoritmos antes de pedirle a una computadora que los ejecute.

La mayor parte del estudio de diseño de algoritmos está enfocado en encontrar algoritmos suficientemente rápidos, y para lograr eso se requiere

una gran cantidad de análisis, pero sobre todo se requiere saber qué clase de algoritmo se está buscando, y para saber eso es indispensable poder medir la velocidad del algoritmo, o dicho de otra manera, su complejidad.

Dado que las computadoras son rápidas, es muy difícil darse cuenta si un programa es rápido o no con entradas pequeñas, así que el análisis de complejidad se enfoca a las entradas *grandes*.

El análisis de complejidad es una técnica para analizar *qué tan rápido* crecen las funciones, y nos centraremos en una función que mide el número máximo de operaciones que puede realizar un algoritmo, a dicha función le llamaremos *función de tiempo*.

Pero antes de analizar la complejidad de las funciones daremos varios ejemplos, el primero de ellos corresponde a una aplicación en la física.

8.1. Un ejemplo no muy computacional

Imagina que sobre una silla de madera tienes un recipiente lleno de agua con forma cilíndrica.

Las patas de la silla tienen una base cuadrada de 5 centímetros por 5 centímetros, el radio del recipiente es de 20 centímetros y su altura es de otros 20 centímetros. Lo cual hace que el recipiente con agua tenga un peso de poco más de 25 kilogramos.

La silla resiste muy bien ese peso, pero ahora imagina que conseguimos un modelo a gran escala formado con los mismos materiales de la silla y el recipiente. Digamos que cada centímetro del viejo modelo equivale a un metro del nuevo modelo, es decir, es una silla 100 veces mas grande y un recipiente 100 veces mas grande.

La pregunta es ¿el nuevo modelo resistirá?. Y la respuesta es simple: no.

En el modelo original $20cm^2$ sostenían $25kg$, es decir, cada centímetro del modelo original sostenía $1{,}25kg$, sin embargo en el nuevo modelo, el area de las patas suma un total de $4(500cm)(500cm) = 1000000cm^2$ y el volúmen del recipiente es $\pi(2000cm)^2(2000cm) = 8{,}042496 * 10^{26}cm^3$ por lo que el peso del nuevo recipiente es $8{,}042496 * 10^{23}kg$, y de esa manera, ¡cada centímetro cuadrado de las nuevas patas tiene que soportar un peso de $804249600000000000kg$!.

En general si un cuerpo que se encuentra sobre el suelo aumenta su tamaño, se dice que el área que esta en contacto con el suelo crece de manera cuadrática, mientras que su peso aumenta de manera cúbica.

Eso mismo se aplica a los animales, durante muchos años las películas nos han mostrado insectos gigantes y lo seguirán haciendo, sin embargo, sabiendo algo de análisis de complejidad podemos darnos cuenta que eso es imposible,

ya que el área de las patas aumenta de manera cuadrática y su peso de manera cúbica.

8.2. Algunos Ejemplos de Complejidad en Programas

Al igual que las áreas y los volúmenes que se mencionan en la sección anterior, el tiempo de ejecución de los programas también puede crecer de maneras cuadráticas, cúbicas, lineales, logarítmicas, etc., en esta sección se analizará el tiempo de ejecución de varios programas.

Supón que tienes la siguiente función:

Código 8.1: Función fnc

```
1 int fnc(int n){
2         int i, k, r=0;
3         for(i=0;i<n;i++){
4                 for(k=0;k<n;k++){
5                         r+=i*k;
6                 }
7         }
8         return r;
9 }
```

Suponiendo que tu computadora tardara un segundo en procesar $fnc(12000)$, ¿Cuánto tiempo crees que tardaría en procesar $fnc(24000)$?

Los que están manejando por primera vez el análisis de complejidad posiblemente contestarían "2 segundos", pero la realidad es otra: se tardaría 4 segundos.

Si miramos bien el código a la variable r se le está sumando el producto de cada par ordenado (i, k) donde $0 \leq i < n$ y $0 \leq k < n$, por regla del producto podemos concluir que se estan realizando n^2 multiplicaciones.

También nos podemos percatar de que el bucle del centro itera n veces por cada iteración del bucle exterior, y esa es otra manera en la que podemos ver que el bucle interior itera un total de n^2 veces.

Por simplicidad ignoraremos las sumas y las comparaciones (esas toman mucho menos tiempo que las multiplicaciones) y nos concentraremos en las multiplicaciones, si cada multiplicación tarda en ejecutarse un tiempo t, entonces al llamar a $fnc(12000)$ el tiempo total de ejecución sería $(12000)^2t$ y al llamar a $fnc(24000)$ el tiempo total de ejecución sería $(24000)^2t$. Así que si $(12000)^2t = 1s$, ¿cuánto será $(24000)^2t$? Una forma de calcularlo sería

obtener el valor de t, pero para este propósito nos dice más resolverlo de la siguiente manera:

$(24000)^2t = (2 * 12000)^2t = 4(12000)^2t$

Sustituyendo 12000^2t por $1s$ obtenemos:

$4(1s) = 4s$

Hemos comprobado que el tiempo de ejecución serían 4 segundos, pero ahora intentemos generalizar un poco.

Ejemplo 8.2.1. Si en una computadora $fnc(a)$ tarda m segundos en ejecutarse, ¿cuánto tiempo tardará $fnc(2 * a)$ en ejecutarse en la misma computadora?

Solución Tenemos que $fnc(a)$ tarda un tiempo de a^2t en ejecutarse. Y $f(2 * a)$ tarda un tiempo de $4a^2t$ en ejecutarse. Sustituyendo a^2t por m tenemos que tardaría $4m$ segundos.

Si repitiéramos el ejemplo anterior pero con $fnc(10 * a)$ ¡veríamos que el tiempo de ejecución ascendería a $100m$ segundos !

□

Ahora considera la siguiente función:

Código 8.2: Función cubo

```
1 double cubo(double x){
2         return x*x*x;
3 }
```

Aquí sin importar qué tan grande sea el valor de x, $cubo(x)$ siempre tomará mas o menos el mismo tiempo en ejecutarse, a diferencia de la función anterior, $cubo(a)$ tardaría el mismo tiempo en ejecutarse que $cubo(10 * a)$.

Un mismo algoritmo puede ser implementado de muchas maneras en un mismo lenguaje, y dependiendo de la implementación, del compilador y del hardware en el que corra el programa, el tiempo de ejecución sería mayor o menor, sin embargo, sin importar todo eso, siempre *aumentará con la misma velocidad.*

Es decir, si vamos a tratar con algoritmos, nos vamos a concentrar en encontrar algoritmos cuyo tiempo de ejecución aumente lo menos posible a medida que la entrada se hace más y más grande.

Por ello, cuando se trata de entradas grandes, lo mas importante es identificar qué tan rápido crecen los tiempos de ejecución y no cuál es el número exacto de operaciones o el tiempo exacto de ejecución.

8.3. Función de Tiempo

Si tuviéramos todo el tiempo y las energías del mundo para resolver un problema sería posible implementar una solución, generar casos de prueba y luego ver si su tiempo de ejecución no es excede del límite; y en caso de que se exceda pensar en otra solución y empezar desde el principio.

Sin embargo, no tenemos todo el tiempo del mundo y seguramente nadie tendría paciencia y energías ilimitadas para implementar muchos algoritmos sin saber de antemano cual va a funcionar en tiempo.Esto hace necesario saber qué tan rápido funcionaría un algoritmo desde antes de implementarlo.

El tiempo de ejecución de un algoritmo no es algo fácil de medir, sobre todo antes de implementarlo; ya que depende directamente de todas las operaciones que se vayan a hacer en el código y de la velocidad de la computadora en cada tipo de operación en específico. El solo hecho de pensar en el número exacto de sumas que realizará una computadora en determinado algoritmo puede ser más tardado que implementar el mismo algoritmo.

Y además de todo esto, un mismo algoritmo puede tardarse diferente cantidad de tiempo con diferentes entradas. Y a veces las variaciones son con entradas del mismo tamaño.

Está claro que es imposible lidiar con todos estos datos cada que se quiera resolver un problema, y es necesario ignorar algunos de ellos para poder analizar de alguna forma la rapidez de un algoritmo.

Como se había mencionado unos capítulos atrás, vamos a considerar una función $E \to T_1(E)$ donde E es el conjunto de los datos de entrada y $T_1(E)$ es el número de operaciones que realiza el algoritmo con esos datos de entrada.

Sería algo deseable poder trabajar con una función con dominio en los enteros y codominio en los enteros, ya que resulta muy difícil trabajar con conjuntos de datos de entrada. Para solucionar esto utilizaremos una función de **cota superior** ó *función pesimista.*

La idea es definir una nueva función $T_2 : \mathbb{N} \longrightarrow \mathbb{N}$ de manera que $T_1(E) \leq T_2(n)$ cuando $n \geq |E|$ para todo conjunto de datos de entrada E.

Otra cosa que hay que definir es que $T_2(n) \leq T_2(n+1)$, eso hará que el análisis se vuelva significativamente mas simple, no es difícil comprobar que siempre existirá una función que cumpla con estas características.

T_2 es una cota superior ya que para cualquier entrada E podemos estar seguros que el número de operaciones que realizará el programa será igual o menor a $T_2(|E|)$ y es pesimista porque lo que menos quisieramos(el peor caso) para una entrada E es que $T_1(E) = T_2(|E|)$.

Esta aproximación del "pesimismo" puede parecer inapropiada en la práctica, pero se hablará de ella por varios motivos:

- Cuando se resuelve un problema, la solución debe de funcionar en todos los casos que describa el problema, si en lugar de analizar el peor caso, analizaramos un caso *promedio*; no estaríamos realmente resolviendo el problema.
- Existen varias maneras de analizar la complejidad, y la que se trata en este libro es la mas simple de todas, y es bueno conocerla como introducción a las otras.
- Hay gran cantidad de algoritmos donde $T_1(E)$ es proporcional a $T_2(|E|)$ para casi cualquier E.
- Algunas veces los usuarios de las aplicaciones que programemos pueden llegar a intentar darle entradas diseñadas específicamente para que nuestro algoritmo se vuelva lento(por ejemplo en un concurso de programación, o en una intrusión a un sistema).

De ahora en adelante llamaremos a T_2 como "la función de tiempo" y la representaremos simplemente como T.

8.4. La Necesidad del Símbolo O

Definiendo la función de tiempo hemos podido reducir una gran cantidad de datos a unos pocos con un enfoque ligeramente pesimista. Pero aún así medir el tiempo sigue siendo muy difícil.

Por ejemplo en la siguiente función:

Código 8.3: maximoSecuencial

```
1  void maximoSecuencial(int v[], int n){
2          int i, r;
3          r=v[0];
4          for(i=1;i<n;i++){
5                  if(v[i]>r){
6                          r=v[i];
7                  }
8          }
9          return r;
10 }
```

Podemos ver que ese código realiza al menos una asignación y a lo mas n asignaciones a la variable r, así que siguiendo la línea del pesimismo vamos a asumir que siempre se realizan n asignaciones a la variable r. Además se

realizan $n-1$ comparaciones entre i y n, $n-1$ comparaciones entre $v[i]$ y r y $n-1$ incrementos, asi que:

$$\begin{aligned} T(n) &= n + (n-1) + (n-1) + (n-1) \\ &= 4n - 3 \end{aligned}$$

El conteo realizado, aunque nos condujo a una expresión algebrárica bastante simple y en poco tiempo, habría sido muy difícil de realizar si solamente conocieramos el algoritmo y no la implementación, por lo que nuevamente caeríamos en la necesidad de tener que implementar el algoritmo antes de saber si es eficiente.

Además, múltiples implementaciones del mismo algoritmo pueden tener funciones de tiempo diferentes.

Una cosa que podríamos hacer es seguir con la línea del pesimismo y asumir que la búsqueda secuencial va a iterar n veces, que en cada iteración va a realizar 100 operaciones(se está eligiendo un número arbitriariamente grande para el número de operaciones dentro de un ciclo) y que lo que está fuera del ciclo va a realizar otras 100 operaciones, así que antes de implementarlo sería posible definir la siguiente función de tiempo:

$$T(n) = 100n + 100$$

Bien podríamos elegir repetidas veces el número 100 para buscar cotas superiores. Pero nos meteríamos en algunas complicaciones con la aritmética. Por ejemplo con el siguiente código:

Código 8.4: Función cosa

```
1 void cosa(int v[], int n){
2         int i, k, r;
3         for(i=0;i<n;i++){
4                 for(k=0;k<v[i]%77;k++){
5                         r+=v[i]%(k+1);
6                 }
7         }
8         return r;
9 }
```

Aquí podríamos pensar que por cada vez que se ejecuta el bucle exterior, el bucle interior se ejecuta 77 veces, que el bucle exterior se ejecuta n veces, y que cada vez que se ejecuta el bucle interior gasta 100 operaciones. Por lo que la función de tiempo sería:

$$\begin{aligned} T(n) &= n(77)(100) \\ &= 7700n \end{aligned}$$

Sabemos que solamente exageramos al elegir el 100 como el número de operaciones que se hace el bucle interior cada que itera, así que el $7700n$ no tiene ningún significado real para nosotros, mas bien sería mas útil dejarlo como $100(77)n$.

El siguiente código, muy parecido al anterior puede causarnos mas molestias.

Código 8.5: Función cosas

```
1  void cosas(int v[], int n){
2          int i, k, r=0, h;
3          for(i=0;i<n;i++){
4                  for(k=0;k<v[i]%77;k++){
5                          r+=v[i]%(k+1);
6                  }
7                  for(h=0;h<n;h++){
8                          for(k=0;k<v[i]%88;k++){
9                                  r+=v[i]%(k+1);
10                         }
11                 }
12         }
13         for(k=0;k<v[i]%55;k++){
14                 r+=v[i]%(k+1);
15         }
16         return r;
17 }
```

Después de ser pesimistas llegaríamos a la conclusión de que

$$\begin{aligned} T(n) &= (100)(n)(77+88n)+(100)(55) \\ &= (100)(88n^2+77n+55) \end{aligned}$$

Observamos que el 100 ahora multiplica a todos los términos. Y si nos ponemos a pensar un poco, el 100 siempre multiplicará a todos los términos si mantenemos la estrategia que hemos seguido hasta ahora.

Así que en lugar de utilizar el 100, puede resultar mas cómodo utilizar otro símbolo, por ejemplo O, para representar un número mayor a lo que cualquier código sin un bucle anidado puede tardar en ejecutarse.

Por lo que las funciones de tiempo en estos ejemplos se pueden expresar como:

$$\begin{aligned} T(n) &= O(n+1) \text{ para maximoSecuencial} \\ T(n) &= O(77n) \text{ para la función cosa} \\ T(n) &= O(88n^2 + 77n + 55) \text{ para la función cosas} \end{aligned}$$

8.5. Definición de la notación O-mayúscula

A pesar de que adoptando el uso de O como una constante de cota superior, aún se puede hacer todavía mas sencillo el análisis. Considera el siguiente ejemplo:

Código 8.6: Función funcion1

```
1 void funcion1(int n){
2         int i, k, r, h;
3         for(i=0;i<n;i++){
4                 for(k=0;k<5;k++){
5                         r++;
6                 }
7         }
8         return r;
9 }
```

Usando el análisis aprendido en la sección anterior podemos concluir que $T(n) = O(5n)$. Pero a continuación se muestra una función que obviamente hace el mismo número de operaciones:

Código 8.7: Función funcion2

```
1  void funcion2(int n){
2          int i, k, r, h;
3          for(i=0;i<n;i++){
4                  k=0;
5                  r++;
6                  k++; k<5;
7                  r++;
8                  k++; k<5;
9                  r++;
10                 k++; k<5;
11                 r++;
```

```
12                      k++; k<5;
13                      r++;
14                      k++; k<5;
15              }
16              return r;
17      }
```

Y aquí, basándonos en la misma manera de analizar obtendríamos la siguiente cota superior: $T(n) = O(n)$. Esto no quiere decir que debamos deshacernos de la idea de usar el símbolo O, sino que los bucles que iteran un número constante de veces pueden contabilizarse como si iteraran una sola vez.

Parece algo descabellado hacer esto en un inicio, pero el ejemplo anterior lo prueba, es lo mismo un ciclo corto que itera un número constante de veces que un código largo que itera solo una vez.

El objetivo de analizar de esta manera los algoritmos se trata de conocer qué tan rápido crece el número de operaciones a medida que la entrada se hace mas grande.

Por ejemplo, considerense 3 funciones de tiempo T, T' y T'' tales que: $T(n) = O(n)$, $T'(n) = O(100n)$ y $T''(n) = O(n^2)$. Tenemos que $T(2n) = 2T(n)$, $T'(2n) = 2T'(n)$ y $T''(2n) = 4T''(n)$.

Aquí podemos ver que T y T' crecen al mismo ritmo, mientras que T'' crece a un ritmo mucho mayor.

Por lo tanto, decir que $T(n) = O(100n)$ es equivalente a decir $T(n) = O(n)$, en ambos casos nos estamos refiriendo a una cota superior que *crece*, al mismo ritmo que n.

Un ejemplo nos servirá para convencernos de que el ritmo de crecimiento es mas importante que el funcionamiento en casos pequeños: un algoritmo A que realiza exactamente $100n$ operaciones será mucho mas rápido que un algoritmo B que realiza n^2 operaciones, donde n es el tamaño de la entrada.

Aunque para valores pequeños, como por ejemplo $n = 3$ o $n = 10$, el algoritmo A es mas rápido, cuando $n = 10000$, por ejemplo, las cosas cambian, ya que $100(10000) = 10^6$ y $(10000)^2 = 10^8$, eso significaría que con una entrada de tamaño 10 mil, el algoritmo A sería 100 veces mas rápido que el algoritmo B.

Luego de justificar por que ignorar las constantes, hemos llegado a un punto donde debemos redefinir nuestra notación, ya que si decimos que $T(n) = O(100n)$ y $T(n) = O(n)$ concluiríamos que $O(100n) = O(n)$ y que $100 = 1$; así que no podemos seguir tratando a O como si fuera un número real.

En lugar de decir que $T(n) = O(n)$ diremos que "$T(n)$ es de órden $O(n)$", y la notación $O(f(n))$, mas que indicarnos que un número real O multiplica al valor de la función f evaluado en n, significa que es posible elegir una constante K de manera que $n \to Kf(n)$ sea una cota superior de la función de tiempo.

Luego de haber reformulado conceptos, es necesario formalizarlo en la siguiente definición:

Definición 8.5.1. Sea f una función con dominio en $\mathbb{N}$ se dice que es de complejidad o de órden $O(g(n))$ si y solo sí existe alguna constante K tal que:

$$f(n) < Kg(n) \text{ para todo } n \in \mathbb{N}$$

Nótese que al hablar de $O(g(n))$ no es necesario expresarlo como $n \to O(g(n))$ para decir que se trata de la función g y no de el valor de g evaluada en n, esto es porque dentro de esta notación no aparecen constantes, solamente variables; salvo que se trate de una función constante en cuyo caso, siempre se expresa como $O(1)$.

Ejemplo 8.5.1. Encuentra la complejidad de fnc, utiliza la notación O-mayúscula.

Solución Si supusiéramos que fnc es de orden $O(n)$, entonces debería de existir alguna k tal que kn siempre sea mayor al número de operaciones que realiza $fnc(n)$, pero podemos darnos cuenta que para cualquier valor entero de k que elijamos, el tiempo de ejecución de $fnc(k+1)$ siempre será mayor que $k(k+1)$, así que fnc no puede ser de orden $O(n)$.

Ahora supongamos que fnc es de complejidad $O(n^2)$ y sea m el numero de operaciones que realiza $fnc(n)$, entonces $fnc(2*n)$ realizará $4m$ operaciones, $fnc(4*n)$ realizará $16m$ segundos en ejecutarse. Si a k le asignamos un valor mayor que m, por ejemplo $(m+1)$, entonces podemos ver que $(m+1)n^2$ siempre será mayor que $fnc(n)$, con esto queda demostrado que la complejidad de fnc es $O(n^2)$.

Código 8.8: Función suma_modular

```
1 int suma_modular(int n){
2         int i, r=0;
3         for(i=0;i<n%100;i++){
4                 r+=i;
5         }
6 }
```

Ejemplo 8.5.2. Analiza la complejidad de *suma_modular*. Utiliza la notación O-mayúscula.

Solución Como podemos ver, el ciclo *for* siempre itera menos de 100 veces, y en cada iteración se realizan siempre el mismo número de operaciones sin importar el valor de n.

Sea C el número de operaciones que se realizan en cada iteración, D el número de operaciones requeridas para llamar a la función y para regresar un valor, podemos darnos cuenta que $CD+1$ siempre será mayor que el numero de operaciones que realiza *suma_modular*, y además $CD+1$ es constante, por lo tanto el tiempo de ejecución de *suma_modular* crece en $O(1)$.

8.6. Múltiples Complejidades en Notación O-Mayúscula

Podrías estarte preguntando, *¿fnc* no será también de orden $O(n^3)$ o de orden $O(n^2+n)$ o de orden $O(n^4)$, etc... ?

La respuesta a esta pregunta es que sí, si un algoritmo tiene una complejidad $O(g(n))$ también se puede decir que ese algoritmo es de cualquier complejidad mayor a $O(g(n))$ solo que por practicidad se suele tomar la función de crecimiento más pequeña que se conozca del algoritmo.

8.7. Cuándo Un Algoritmo es Factible y Cuándo Buscar Otro

Ya vimos lo qué es la notación O-mayúscula pero aún no se habla del tema de cómo saber si un algoritmo de cierta complejidad resuelve el problema.

Una computadora con $1{,}5Ghz$ en un segundo realiza poco mas de 200 millones de sumas. Puede ser un buen punto de referencia saber eso, pero está claro que una iteración por lo general consiste de mas operaciones que una sola suma por este motivo es necesario conocer en qué tamaños de las entradas suelen funcionar los algoritmos con diferentes complejidades.

En la siguiente tabla se muestran varias complejidades y valores máximos de n que suelen ser los indicados en la mayoría de los problemas de programación si se requiere que el programa funcione en menos de un segundo(esto se puede extrapolar para tiempos mayores con sencillos cálculos algebráricos).

También existen algoritmos en los que cada iteración requiere de muchas operaciones y el valor máximo de n debe de ser más pequeño; pero con esos

"casos especiales" solamente se aprende a lidiar por medio de la experiencia.

Tabla de Complejidades

Complejidad	**Valor Máximo de n**
$O(1)$	∞
$O(logn)$	$2^{50000000}$
$O(\sqrt{n})$	10^{15}
$O(n)$	50000000
$O(nlogn)$	5000000
$O(n^2)$	5000
$O(n^3)$	500
$O(n^4)$	80
$O(2^n)$	20
$O(n!)$	11

Capítulo 9

Reglas para Medir la Complejidad

Aunque comience a parecer algo tedioso calcular las complejidades de los algoritmos, es fácil en la mayoría de los casos, ya que existen ciertas técnicas usadas para calcular la complejidad de los algoritmos, las cuales descubriremos en las próximas páginas.

Cuando decimos que un algoritmo A es de complejidad $O(f(n))$ nos referimos a que si $T(n)$ es la función del tiempo que tarda en ejecutarse el algrotimo A con una entrada de tamaño n entonces la complejidad de la función $T(n)$ es la misma que la de la función $f(n)$, o dicho de otra forma $O(T(n)) = O(f(n))$.

9.1. Regla de la Suma

La primera de las reglas para calcular complejidad es la regla de la suma(no debe confundirse con la regla combinatoria de la suma). Esta regla implica que al ejecutar dos algoritmos diferentes, uno después del otro, la complejidad de ejecutar ambos algoritmos será igual a la complejidad de ejecutar el algoritmo mas lento de ellos.

De una manera mas precisa, si el tiempo para ejecutar un algoritmo es $n \to T(n) + T'(n)$ entonces su complejidad es $O(max(T(n), T'(n)))$, o bien $O(T(n) + T'(n)) = O(max(T(n), T'(n)))$.

Esta regla a primera vista puede causar escepticismo y a veces negación a usarse, pero basta con ver que los valores de $f(x) = x^3 + 3x^3 + 100$ y de $g(x) = x^3$ se vuelven casi iguales conforme crece x. Por ejemplo:

$$f(1000) = 1003000100$$

$$g(1000) = 1000000000$$

Por ello, si un algoritmo que realiza $g(n)$ operaciones tarda 10 segundos en ejecutarse cuando $n = 1000$, un algoritmo que realize $f(n)$ operaciones tardaría 10,03 segundos.

Es decir, para todo ε existe un δ tal que $|f(a) - g(a)| < \varepsilon$ siempre que $a > \delta$.

Aunque después de haber visto el ejemplo anterior puedas entender un poco la razón de la regla de la suma, es conveniente conocer su demostración para estar seguros que realmente se aplica a la notación que definimos como $O-$ mayúscula.

Teorema 11 (Regla de la suma)**.** *Si una función a es de complejidad* $O(f(n))$ *y una función b es de complejidad* $O(g(n))$ *la complejidad de* $a+b$ *es* $O(max(f(n), g(n)))$.

O dicho de otra forma $O(f(n) + g(n)) = O(max(f(n), g(n)))$

Demostración. Sin perder la generalidad supongamos que $f(m) \geq g(m)$ para alguna m.

Por definición existen dos constantes J y K tal que

$$\begin{aligned} Jf(n) &> a(n) \\ Kg(n) &> b(n) \\ \text{y } J &> K \text{ para toda } n \end{aligned}$$

En consecuencia tenemos que

$$\begin{aligned} Jf(m) + Jf(m) &\geq Jf(m) + Kg(m) > a(m) + b(m) \\ 2Jf(m) &> a(m) + b(m) \end{aligned}$$

Analogamente, si $g(m) \geq f(m)$ entonces existe J tal que $2Jg(m) > a(m) + b(m)$.

Como J es constante, entonces podemos concluir que la complejidad de ejecutar el algoritmo a y posteriormente ejecutar el algoritmo b es $O(max(f(n), g(n)))$, es decir $O(f(n) + g(n)) = O(max(f(n), g(n)))$.

□

Dado que trabajamos con funciones de tiempo creciente, lo mas frecuente sería encontrarnos con que $O(f(n)+g(n))$ es o bien $O(f(n))$ o bien $O(g(n))$. Es decir, $f(n) \leq g(n)$ para toda n ó $f(n) \geq g(n)$ para toda n.

Pero no siempre es así, puede haber valores de n para los cuales $f(n) > g(n)$ y valores de n para los cuales $f(n) < g(n)$. Por lo tanto, no siempre nos "salvaremos" de expresar una complejidad de la forma $O(f(n) + g(n))$, un ejemplo de esto son los algoritmos de búsqueda que se analizarán al final de la parte V.

Código 9.1: Función burbuja

```
1  void burbuja(int arreglo[], int n){
2          int i, k;
3          for(i=0;i<n;i++){
4                  for(k=0;k+1<n;k++)
5                          if(arreglo[k]<arreglo[k+1])
6                                  swap(arreglo[k],
                                       arreglo[k+1]);
7          }
8          for(i=0, k=n;i>k;i++,k--)
9                  swap(arreglo[i], arreglo[k]);
10         return;
11 }
```

Ejemplo 9.1.1 (Complejidad del Algoritmo Burbuja)**.**

Mide la complejidad de la función burbuja implementada en el código anterior. Utiliza la notación O-mayúscula.

Solución El *for* de la línea 3 itera n veces, por cada iteración del *for* de la línea 3, el *for* de la línea 4 itera $n-1$ veces, por lo tanto, el número de veces que se ejecuta la línea 5 es $(n)(n-1)$ y el número de intercambios que se realizan en la línea 6 nunca es mayor a $(n)(n-1)$.

Luego, el número de operaciones cada vez que itera *el for* de la línea 4 es menor o igual que:

$$2(n)(n-1) = 2n^2 - 2n$$

Nótese que $2n^2 > 2n^2 - 2n$ para cualquier valor de n positivo, por lo que el tiempo de ejecución desde la línea 3 hasta la línea 7 es de orden $O(n^2)$, o dicho de otra manera, cuadrático.

El *for* de la línea 8 itera $\frac{n}{2}$ veces, y el número de intercambios que se realizan también es $\frac{n}{2}$, eso significa que cada una de las $\frac{n}{2}$ iteraciones realiza el mismo número de operaciones, dicho número de operaciones lo vamos a denotar como m.

Por ello, el número de operaciones que se ejecutan entre las líneas 8 y 9 es:

$$m\frac{n}{2} = \frac{m}{2}n$$

Dado que $\frac{m}{2}$ es una constante(es fácil ver que existe una cantidad de tiempo constante que siempre será mayor que el tiempo que tarden en ejecutarse

$\frac{m}{2}$ operaciones), el tiempo de ejecución de las líneas 8 y 9 es de complejidad $O(n)$.

Como primero se ejecuta un algoritmo de $O(n^2)$ (líneas 1 a 7) y posteriormente se ejecuta un algoritmo de $O(n)$ (líneas 8 y 9), por regla de la suma, el tiempo de ejecución es de complejidad $O(max(n, n^2)) = O(n^2)$.

9.2. Producto por Constante

La siguiente regla es un tanto obvia, pero no por eso se debe despreciar.

La regla dice que si dos funciones T y t son tales que $T(n) = Kt(n)$ para alguna constante K, entonces su complejidad es la misma, es decir $O(T(n)) = O(t(n))$.

Teorema 12. *Si una función $f(n) = Kt(n)$ donde K es una constante, la complejidad de f es la misma que la de t, dicho de otra manera $O(Kf(n)) = O(f(n))$ para cualquier constante K.*

Demostración. Sean t y f funciones tales que la complejidad de t es $O(f(n))$, por definición existe una constante K' tal que $K'f(n) > t(n)$ para toda n, multiplicando ambos lados de la desigualdad por la constante no negativa K, tenemos que $(K)(K')f(n) > (K)t(n)$.

Como $(K)(K')$ es constante, entonces $O(Kf(n)) = O(f(n))$.

□

9.3. Regla del Producto

Considera el siguiente código:

Código 9.2: f y g

```
1  int f(int n){
2          int i, r=0;
3          for(i=1;i<=n;i++){
4                  r+=n;
5          }
6          return r;
7  }
8  int g(int n){
9          int i, k, r=0;
10         for(k=1;k<=n;k++){
11                 for(i=1;i<=n;i++){
```

```
12                                  r+=f(k)*f(i);
13                          }
14                  }
15                  return r;
16  }
```

Intentar calcular el número exacto de iteraciones que se realizan en la función anterior es algo muy difícil, pero a la vez si analizaramos el número de veces que la función g manda a llamar a la función f en lugar del tiempo que tarda en ejecutarse la función g, nos daremos cuenta que ese número se calcula con la función $v(n) = 2n^2$ la cual es una función de órden $O(n^2)$.

Vamos a denotar n como la entrada de f y n' como la entrada de g, recordando que el número de llamadas a f desde g es de órden $O(n'^2)$ y la complejidad de f es $O(n)$, por definición es posible elegir 2 constantes K y K' tales que:

$$Kn^2 > 2n^2 \text{ y } K'n' > n' \tag{9.1}$$

Por la ecuación anterior podemos concluir que:

$$Kn^2K'n' > 2n^2n' \tag{9.2}$$

Recordando que $n' \leq n$, y por la ecuación anterior

$$\begin{aligned} Kn^2(K'n) &\geq Kn^2K'n' > 2n^2n' \\ (K)(K')n^3 &\geq Kn^2K'n' > 2n^2n' \\ (K)(K')n^3 &> 2n^2n' \end{aligned}$$

De ésto se puede concluir que g corre en un tiempo de orden $O(n^3)$.

De manera análoga se puede demostrar que si $g(n) = f(n)f'(n)$ entonces g es de órden $O(f(n)f'(n))$.

Teorema 13 (Regla del Producto)**.** *Sea $g(x) = f(x)f'(x)$, la complejidad de g es igual al producto de las complejidades de f y f'.*

9.4. Complejidad en Polinomios

Considera el siguiente código:

```
1  for(i=N;i>=0;i--) {  }
2  for(i=0;i<N;i++){
3          for(k=2;k<N;k++){
4                  for(h=0;h<N;h+=5);
```

```
5           }
6           for (k=0;k<N%128;k++) { }
7   }
```

El número de veces que itera *el for* de la línea 2 es N, el número de veces que itera el *for* de la línea 3 es $N(N-2)$, el número de veces que itera el *for* de la línea 4 es $(N)(N-2)(N/5)$, el número de veces que itera el *for* de la línea 6 es $Nmod128$, y el número de veces que itera el *for* de la línea $1esN$.

Por tanto el número total de iteraciones es:

$$N+N(N-2)+N(N-2)(N/5)+Nmod128+N = (1/5)N^3+(3/5)N^2+N+Nmod128$$

Aplicando la regla de la suma sabemos que:

$$O((1/5)N^3+(3/5)N^2+N+Nmod128) = O(max(O((1/5)N^3), O((3/5)N^2), O(N), O(Nmod12$$

Aplicando la regla del producto por constante tenemos que:

$$O((1/5)N^3 + (3/5)N^2 + N + Nmod128) = O(max(N^3, N^2, N, 1)) = O(N^3)$$

Como te habrás dado cuenta, en cualquier algoritmo cuyo tiempo de ejecución se pueda representar mediante un polinomio, lo único que hay que hacer es tomar el término con mayor exponente de la variable y olvidarse del coeficiente.

Y si eres un poco más observador, te podrás dar cuenta que ni siquiera es necesario expandir el polinomio. Es decir:

$$O(N+N(N-2)+N(N-2)(N/5)+Nmod128+N) = O(N+N(N)+N(N)(N)+1+N)$$

Las constantes se pueden ignorar, y de esa manera se vuelve trivial dado un polinomio encontrar su complejidad. Se deja como ejercicio para el lector comprobar que esto se puede hacer con cualquier polinomio.

9.5. Medir antes de implementar

En el capítulo 11 se planteó el objetivo de cómo saber si un algoritmo es rápido sin necesidad de implementarlo. Y hasta este momento solamente nos hemos dedicado a analizar la complejidad de implementaciones.

Sin embargo, ya tenemos las herramientas suficientes para poder analizar la complejidad de muchos algoritmos sin necesidad de su implementación;

para lograr esto simplemente hay que examinar qué ciclos posee el algoritmo y cuáles de esos ciclos estan uno dentro de el otro.

Una vez identificados los ciclos, hay que estimar cual es el número máximo de veces que se puede ejecutar cada uno, ignorando las constantes, obteniendo así un polinomio.

Luego de eso simplemente hay que tomar el término del polinomio con exponente mayor.

Los siguientes ejemplos ilustran un poco esta técnica

Ejemplo 9.5.1. Analiza la complejidad del algoritmo para encontrar el máximo elemento de un arreglo A con n elementos de la siguiente manera:

Para cada elemento i del arreglo A, verifica que i sea mayor que todos los demas elementos de a, si se cumple i será el máximo.

Solución Para saber si un elemento i es el máximo, el algoritmo verifica que $a > k$ para todo $k \in A$ tal que $k \neq i$, como hay n valores posibles de A, entonces esta operación hay que realizarla $n-1$ veces, eso es complejidad $O(n)$.

Y como hay que verificar n elementos distintos y cada elemento toma un tiempo de $O(n)$ verificarlo, la complejidad es $O(n^2)$.

□

Ejemplo 9.5.2. Analiza la complejidad del algoritmo para encontrar el máximo elemento de un arreglo A con n elementos de la siguiente manera:

Primero se inicializa una variable *maximo* con valor $-\infty$

Para cada elemento i del arreglo A, si el valor de *maximo* es menor que el valor de i entonces se sustituye el valor de *maximo* por el valor de i.

Solución Inicializar el valor de *maximo* toma tiempo $O(1)$, comparar el valor de *maximo* con el valor de alguna i tambien toma tiempo $O(1)$, pero hay que hacer dicha comparación n veces, por lo tanto el algoritmo es $O(n)$.

□

A pesar de que ya no se requiere un código fuente para analizar los algoritmos, en este libro se seguirán usando, debido a que describir los algoritmos en español puede resultar confuso, y no vale la pena explicar un pseudocódigo.

9.6. Búsqueda de Cotas Mayores

Aunque la mayoría de veces este proceso es sencillo hay veces en las que no es fácil ver cuantas veces se ejecuta un ciclo aún si se ignoran las constantes, pero para analizar esos casos lo mas conveniente a hacer es elegir una

cota máxima y obtener cierta complejidad aún cuando no sea la complejidad menor del algoritmo ó encontrar una propiedad en el algoritmo que permita determinar cuántas veces se ejecuta(para lo cual no hay una regla general).

Código 9.3: Función cuatro_cuadrados

```
1  int cuatro_cuadrados(int n){
2          int a, b, c, d;
3          int r=0;
4          for(a=0; a*a<=n;a++){
5                  for(b=a; a*a+b*b<=n;b++){
6                          for(c=b;a*a+b*b+c*c<=n;c++){
7                                  for(d=c;a*a+b*b+c*c+d*d
                                       <=n;d++){
8                                          if(a*a+b*b+c*c+d
                                               *d==n){
9                                                  r++;
10                                         }
11                                 }
12                         }
13                 }
14         }
15         return r;
16 }
```

Por ejemplo en el código 9.3 resulta extremadamente difícil darse cuenta cuantas veces itera cada ciclo, excepto el exterior que itera $\sqrt{n}$ veces, en los siguientes ciclos solamente es posible darse cuenta que pueden iterar $\sqrt{n}$ veces pero la mayoría de ocaciones iteran menos.

En este caso se puede decir que n^2 es una **cota mayor** al total de iteraciones en la función cuatro_cuadrados.

Asi que, por regla del producto podemos saber que la función cuatro_cuadrados itera menos de $\sqrt{n}^4 = n^2$ veces, lo que significa que si $T(n)$ es el número total de iteraciones.

$$n^2 > T(n)$$

Sea K una consante tal que cada iteración realiza menos de K operaciones, entonces:

$$T(n) < K(n^2)$$

Por lo tanto, podemos decir que la función cuatro_cuadrados corre en tiempo $O(n^2)$ nótese que n^2 es una cota superior, por lo cual puede ser posible que $O(n^2)$ no sea la menor complejidad de T que se puede hayar.

Al medir complejidades muchas veces es útil usar cotas mayores, ya que esto asegura que el algoritmo va a funcionar en un tiempo menor al calculado usando la cota mayor y eso a veces es suficiente para saber que el algoritmo va a funcionar dentro de las condiciones que se piden en el problema.

Capítulo 10

Complejidades Logarítmicas

En el capítulo anterior se estudiaron múltiples complejidades de la forma $O(n^k)$ y se mostró como gran cantidad de algoritmos corren en tiempos que se pueden expresar como un polinomio y como toda función polinomial tiene una complejidad de la forma $O(n^k)$ donde k es una constante.

Otro tipo de complejidades importantes son las complejidades $O(log\ n)$, aunque suene extraño usar logaritmos en el conteo del número de operaciones, este tipo de complejidades aparecen de una manera muy frecuente.

Cabe mencionar que en los algoritmos, suelen ser mas importantes los logaritmos base 2 que los logaritmos naturales.

10.1. Análisis de la Búsqueda Binaria

El primer algoritmo que veremos con esta complejidad es uno que ya se mencionó en la Parte I, sin embargo nunca se analizó su complejidad.

```
1  int Busqueda_Binaria(int v[], int a, int b, int x){
2          while(a<b)
3                  if(v[(a+b)/2]==x){
4                          return (a+b)/2;
5                  }else if(v[(a+b)/2]<x){
6                          a=(a+b)/2+1;
7                  }else{
8                          b=(a+b)/2-1;
9                  }
10           if(v[a]==x){
11                 return a;
12         }else{
```

```
13                  return -1;
14              }
15      }
```

Ejemplo 10.1.1. Analiza la complejidad de la búsqueda binaria, utiliza la notación O mayúscula.

Solución Todas las iteraciones de este algoritmo ejecutan la línea 3, y las iteraciones se ejecutan mientras no se haya encontrado un $v[\frac{(a+b)}{2}] = x$ y mientras a sea menor que b.

Para conocer el número de iteraciones primero debemos estar seguros de que este código realmente termina y no se queda en un bucle infinito.

Es posible que en v no exista ningún valor igual a x, pero, ¿será posible que se mantenga indefinidamente la condición $a < b$? en seguida demostraremos que esto último no es posible:

Si $a < b$ entonces $b - a > 0$

Vamos a llamar a' y b' a los valores que tomarán a y b en la siguiente iteración. En cada una de las iteraciones ocurre alguna de estas 2 cosas:

$a' = \frac{a+b}{2} + 1$ ó bien $b' = \frac{a+b}{2} - 1$

Por tanto:

$$b' - a' = b - \frac{a+b}{2} - 1$$
$$b' - a' = b - \frac{a}{2} - \frac{b}{2} - 1 - (a+b)mod2$$
$$b' - a' = \lceil \frac{b}{2} \rceil - \lfloor \frac{a}{2} \rfloor - 1 - (a+b)mod2$$
$$\lceil \frac{b}{2} \rceil - \lfloor \frac{a}{2} \rfloor - 1 - (a+b)mod2 < \lfloor \frac{b-a}{2} \rfloor$$
$$b' - a' < \lfloor \frac{b-a}{2} \rfloor$$

o bien

$$b' - a' = \frac{a+b}{2} - 1 - a$$
$$b' - a' = \frac{a}{2} + \frac{b}{2} + amod2 - 1 - a$$
$$\frac{a}{2} + \frac{b}{2} + amod2 - 1 - a < \lfloor \frac{b-a}{2} \rfloor$$
$$b' - a' < \lfloor \frac{b-a}{2} \rfloor$$

Con lo anterior podemos darnos cuenta que luego de cada iteración $a-b$ se reduce a la mitad o menos. Cuando se trata de números reales, ésta situación puede continuar por un tiempo indefinido, pero cuando se trata de números enteros, después de cierto número de operaciones se llegará a la situación inevitable de que $a = b$.

Es fácil de darse cuenta que si inicialmente $b - a = 2^n$, entonces el algoritmo terminará en el peor caso después de n iteraciones, es decir después de $log_2(n)$ iteraciones.

También es fácil probar por inducción que si x no está en v, $busqueda\_binaria(v, a, b, x)$ tarda igual o menos tiempo que $busqueda\_binaria(v, a, b + 1, x)$.

Por lo tanto, $busqueda\_binaria(v, 0, n, x)$ no tardará mas que $busqueda\_binaria(v, 0, m, x)$ donde m es la menor potencia de 2 tal que $n \leq m$.

Con esto concluimos que la complejidad de la búsqueda binaria es $O(log\ n)$.

□

10.2. Bases de logaritmos

Resulta curioso nunca ver las bases de los logaritmos en las complejidades, esto se debe a que todos los logartimos tienen exactamente la misma complejidad.

Teorema 14. *$O(log_k\ f(n)) = O(log_h\ f(n))$ para cualquier función f y cualesquiera dos números reales k y h.*

Demostración. Usando las leyes de los logaritmos sabemos que $log_k\ x = \frac{ln\ x}{ln\ k}$ para cualquier x y que $log_k\ x = \frac{ln\ x}{ln\ h}$ para cualquier x.

Sea $K = max(ln\ h, ln\ k)$ tenemos que:

$$K(ln\ f(n)) \geq log_k\ f(n)$$

y además

$$K(ln\ f(n)) \geq log_h\ f(n)$$

Con esto probamos que $O(log_m\ f(n)) = O(ln\ f(n))$ sin importar cual sea el valor de m, esto quiere decir que todos los logaritmos tienen la misma complejidad.

□

10.3. Complejidades $O(NlogN)$

A diferencia de la búsqueda binaria, la siguiente función no tiene una aplicación practica, sin embargo, pronto veremos que su complejidad aparece de una manera muy natural el gran cantidad de algoritmos:

```
1  void funcion_nlogn(int n){
2          int i=n, k;
3          while(i>0){
4                  i/=2;
5                  for(k=0;k<n;k++){ }
6          }
7  }
```

Ejemplo 10.3.1. Analiza la complejidad de $funcion\_nlogn$. Utiliza la notación O mayúscula.

Solución Tenemos que el *while* itera no mas de $logn + 1$ veces, y que el for itera n veces por cada iteración del *while*.

Por regla del producto tenemos que la complejidad de $funcion\_nlogn$ es $O((logn + 1)(n)) = O(O(logn)O(n)) = O(nlogn)$. □.

Puede parecer extraña la complejidad $O(nlogn)$ pero mas tarde nos daremos cuenta que es bastante común en el ordenamiento.

Capítulo 11

Complejidades en Funciones Recursivas

Las funciones recursivas presentan mas problemas para analizarse que las funciones iterativas, ya que, a diferencia de las funciones iterativas, no es posible saber exactamente *cuántos ciclos se anidan*.

En la Parte I se muestra la búsqueda binaria recursiva y unas páginas atrás se demostró que la complejidad de la búsqueda binaria es logarítmica, ese puede ser un ejemplo de cómo un algoritmo recursivo puede ser notablemente rápido, sin embargo, en la Parte I se mostró que la función recursiva de fibonacci crece muy rápido, por lo que los algoritmos recursivos también pueden ser notablemente lentos.

Pero al usar recursión con memoria, vimos como se puede mejorar de manera considerable la velocidad del algoritmo recursivo para calcular los números de fibonacci.

Por otro lado, resultaría casi imposible simular mentalmente un algoritmo recursivo con el fin de determinar qué tan rápido es.

11.1. Estados

El concepto de "estado" es un concepto que cobrará mucha importancia mas adelante, pero por el momento lo veremos de una manera superficial.

A medida que se ejecuta un algoritmo, los valores de las variables o de los arreglos pueden variar, vamos a llamar "estado" a un conjunto de valores que pueden tener determinadas variables. Puede haber muchos puntos de referencia para determinar los estados.

Por ejemplo, en el código 9.3, podemos decir que los estados son todos los conjuntos de valores de las variables a, b, c y d, o bien tambien podriamos decir que los estados son todos los conjuntos de valores de las variables a, b

y r, etc.

Para medir adecuadamente la complejidad es necesario elegir un conjunto de estados de manera que el programa tarde un tiempo acotable en cambiar de un estado a otro.

La idea es medir la complejidad del algoritmo en un estado, y luego multiplicar esa complejidad por el número de estados; y así no será necesario simular mentalmente la recursión.

El siguiente ejemplo ayudará a comprender mejor lo que se pretende hacer con definir estados:

```
1  int Arreglo[30];
2  void binario(int n){
3          if(n==0){
4                  procesa(Arreglo);
5          }else{
6                  Arreglo[n]=0;
7                  binario(n-1);
8                  Arreglo[n]=1;
9                  binario(n-1);
10         }
11 }
```

Ejemplo 11.1.1. Analiza la complejidad de *binario* en el código anterior.

Solución Vamos a tomar como los estados a todos los posibles valores de n y los primeros n números del arreglo *Arreglo*.

De lo cual nos resulta que el número de estados es la cantidad de cadenas de 1 bit más la cantidad de cadenas de 2 bits, más la cantidad de cadenas de 3 bits, ..., más la cantidad de cadenas de n bits.

Por regla del producto sabemos que esto es:

$$2^1 + 2^2 + ... + 2^n \tag{11.1}$$

Lo cual, por la ecuación 4.1 sabemos que es:

$$2^{n+1} - 1 \tag{11.2}$$

De esta manera hay un total de $2^{n+1} - 1$ estados, y el programa tarda un tiempo constante en cambiar de un estado a otro, por lo tanto la complejidad es $O(2^n)$. □

```
1  void dfs(int a){
2          int i;
```

```
3           if(a<0  ||  a>=N)
4                   return;
5           V[a]=true;
6           for(i=0;i<n;i++){
7                   if(Mat[a][i] && !V[i]){
8                           dfs(Mat[a][i]);
9                   }
10          }
11  }
```

Ejemplo 11.1.2. Analiza la complejidad de *dfs* en el código anterior.

Solución Podemos darnos cuenta que si $a < 0$ ó $a \geq n$ entonces el programa terminaría en un número constante de pasos. Retomando nuestra "actitud pesimista" vamos a asumir que a solamente puede tener valores entre 0 y $n-1$, y además, vamos a utilizar el conjunto de posibles valores de a como los estados.

Debido a que $dfs(x)$ es llamado a lo mas una vez para cada x, podemos alegremente ignorar la recursión y solo contar el tiempo utilizado por el programa durante la ejecución de $dfs(x)$(sin contar las llamadas recursivas) para cada x.

El tiempo que tarda en ejecutarse $dfs(x)$ para alguna x y sin contar las llamadas recursivas es de complejidad $O(n)$; y como x puede tener hasta n valores diferentes la complejidad es $O(n^2)$.

11.2. Cortes

Muchas veces, al hacer llamadas recursivas un problema se divide en problemas mas pequeños, tal como se vió en el capítulo de "Divide y Vencerás", cuando esto sucede, frecuentemente un estado *mas pequeño* se procesa más rápido que un *estado mas grande* y las cosas se complican bastante.

Considere, por ejemplo, el siguiente código.

```
1   int suma_dummy(int a, int b){
2           int i, r=0;
3           if(a<=b)
4                   return 0;
5           for(i=a;i<b;i++){
6                   r++;
7           }
```

```
 8            r+=suma_dummy(a,  (a+b)/2));
 9            r+=suma_dummy((a+b)/2,  b));
10            return r;
11    }
```

Ejemplo 11.2.1. Analiza la complejidad de *suma_dummy*

Solución Vamos a denotar $suma\_dummy(a,b)$ como el tiempo en que tarda en ejecutarse dicha función.

Es obvio que $O(suma\_dummy(a,b)) = O(b-a+suma_dummy(a,\frac{a+b}{2}) + suma\_dummy(\frac{a+b}{2},b))$ cuando $a > b$ y que $O(suma\_dummy(a,b)) = O(1)$ cuando $a <= b$, sin embargo, expresar la complejidad de manera recursiva no parece ser una buena idea.

Para resolver esta problemática hay que tomar en cuenta el hecho de $O(suma\_dummy(a,b)) = O(suma\_dummy(a+c,b+c))$ para cualesquiera a, b y c.

Esto no es muy fácil de admitir, sin embargo, se justifica al ver que el número de iteraciones del ciclo for solamente depende de la diferencia entre a y b, lo mismo la condición de retorno.

Sin embargo, hay un par de líneas donde si parece afectar:

```
1            r+=suma_dummy(a,  (a+b)/2));
2            r+=suma_dummy((a+b)/2,  b));
```

Pero, esta problematica se resuelve rápidamente:

$$\lfloor\frac{a+b+2c}{2}\rfloor - (a+c) = \lfloor\frac{a+b}{2}\rfloor + c - a - c = \lfloor\frac{a+b}{2}\rfloor - a$$

Analogamente $b - \lfloor a+b\rfloor = b + c - \lfloor a+b+2c\rfloor$.

Una vez admitido que $O(suma\_dummy(a,b)) = O(suma\_dummy(a+c, b+c))$, vamos a utilizar la expresión $T(n)$ como el tiempo que tarda en ejecutarse $suma\_dummy(0,n)$, de esa manera $O(T(b-a)) = O(suma\_dummy(a,b))$.

Lo interesante aquí es que $O(T(n)) = O(n + 2T(\lceil\frac{n}{2}\rceil))$.

Para un análisis superificial, tomemos las potencias de 2 y veamos que sucede con $O(T(n))$ cuando n es una potencia de 2.

Además utilizaremos la letra k para designar una constante de tiempo *grande*.

- $T(1) \leq k$
- $T(2) \leq 2k + 2k = 2(2k) = 4k$

- $T(4) \leq 4k + 2(2k + 2k) = 3(4k) = 12k$
- $T(8) \leq 8k + 2(12k) = 4(8k) = 32k$
- $T(16) \leq 16k + 2(32k) = 5(16k) = 80k$

Utilizando inducción es fácil confirmar que $T(2^n) \leq k(2^n)(n+1)$. Puesto que $T(1) \leq k$ y $T(2^n) \leq k(2^n)(n+1)$ implica que $T(2^{n+1}) \leq 2^{n+1}k + 2k(2^n)(n+1) = k(2^{n+1} + 2^{n+1}(n+1) = k(2^{n+1})(n+2)$

Ahora, recordando que $T(n) \leq T(n+1)$ para todo n, tenemos que:

$$\begin{aligned} T(n) &\leq T(2^{\lceil log_2 n \rceil}) && (11.3) \\ &\leq K(2^{\lceil log_2 n \rceil})(\lceil log_2 n \rceil + 1) && (11.4) \\ &\leq K(n+1)(\lceil log_2 n \rceil + 1) && (11.5) \end{aligned}$$

Por lo tanto, la complejidad de *suma_dummy* es $O(nlog_2 n)$.

Hay que hacer notar que en el análisis de complejidad no importa la base del logaritmo, ya que $log_a b = \frac{lnb}{lna}$. Por lo que decir que un algoritmo es de complejidad $O(nlog_2 n)$ es lo mismo que decir que es de complejidad $O(\frac{1}{ln2}(nlog_2 n))$. Y como $ln2$ es una constante entonces $O(nlog_2 n) = O(n\ ln\ n)$.

□

Nuevamente nos encontramos con una complejidad $O(n\ log\ n)$. Como se había dicho anteriormente, esta complejidad aparece mucho en los algoritmos de ordenamiento. Pero no solamente aparece en esos algoritmos, sino que aparece por lo general en algoritmos que requieren realizar *cortes*.

Generalizando el ejemplo anterior, si K y M son constantes, un algoritmo basado en la estrategia "divide y vencerás" requiere un tiempo $O(1)$ procesar una entrada de tamaño $\leq K$ y requiere un tiempo $O(n)$ dividir una entrada tamaño $n > K$ en M entradas de tamaño $\frac{n}{M}$, la complejidad del algoritmo es $O(n\ log_M\ n)$.

La demostración de esta propiedad es análoga al análisis de complejidad de la función *suma_dummy*. Se deja como ejercicio para el lector.

Parte III

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son quizá los ejemplos mas citados de algoritmos donde la eficiencia juega un papel muy importante y también sirven para realizar una de las tareas mas comunes de una computadora, que es la de ordenar datos. Incluso un sinónimo de la palabra "'computadora"' es "'ordenador"'.

Podemos ver con frecuencia que aplicaciones como hojas de cálculo o navegadores de archivos tienen la opción de ordenar los datos para mostrarlos de una manera mas presentable. Sin embargo el ordenamiento tiene un propósito mucho mas importante que el de mostrar datos organizados.

Para entender la importancia del ordenamiento recordemos que en la Parte I se mostró un algoritmo llamado "'Busqueda Binaria"' el cual servía para encontrar valores en secuencias ordenadas de manera no descendente y parecía ser bastante rápido. Y en la Parte III se demostró que la complejidad en tiempo de la Búsqueda Binaria es $O(log_2 N)$. Es decir, encontrar datos en una secuencia ordenada de forma no descendente es mucho mas rápido que encontrar datos en una secuencia cualquiera.

Por este y otros motivos los algoritmos de ordenamiento juegan un papel escencial en las ciencias de la computación.

Capítulo 12

Ordenamiento

Antes de comenzar a tratar los algoritmos de ordenamiento es necesario saber cual es exactamente el objetivo de los algoritmos de ordenamiento y qué reglas deben de seguir. El sentido común nos dice que dada una secuencia A de números enteros positivos, queremos encontrar una secuencia $B = (b_1, b_2, b_3, ..., b_n)$ tal que B es una permutación de los elementos de A y además:

$$b_1 \leq b_2 \leq b_3 \leq ... \leq b_n$$

Pero los algoritmos de ordenamiento buscan algo mas general, lo cual examinaremos a continuación.

12.1. Función de comparación

Muy frecuentemente usamos expresiones tales como $a < b$, $a > b$ ó $a \leq b$. Cuando se trata de números reales sabemos que $a < b$ si y solo sí $a - b \notin \mathbb{R}_0^+$ o dicho de otra manera, nuestra **función de comparación** es:

$$\begin{aligned} f(a,b) &= 1 \text{ si } a - b \notin \mathbb{R}_0^+ \\ &\quad\ 0 \text{ en otro caso} \end{aligned}$$

Es decir, una función de comparación es una función que indica si un elemento es menor que otro.

Muchos lenguajes de programación traen implementados previamente algoritmos de ordenamiento y permiten que el programador defina sus propias funciones de comparación.

A continuación se define de una manera formal lo que es una función de comparación.

Definición 12.1.1 (Función de Comparación)**.** Una función de comparación es una función $f : A \times A \to \{0, 1\}$ para algun conjunto A, tal que se cumplen las siguientes propiedades:

- $f(a, a) = 0$ (antisimetría)
- Si $f(a, b) = 1$ y $f(b, c) = 1$ entonces $f(a, c) = 1$ (transitividad)
- Si $f(a, b) = 1$ entonces $f(b, a) = 0$ (totalidad o *completitud*)

La definición anterior se puede interpretar como que f es una función que hace comparaciones cumpliendo las siguientes reglas:

- Todos los pares ordenadados (a, b) tales que $a \in A$ y $b \in A$ se pueden comparar haciendo uso de f.
- Ningún elemento es menor que si mismo.
- Si a es menor que b y b es menor que c entonces a es menor que c.
- Si a es menor que b entonces b no es menor que a.

12.2. Conjunto Totalmente Ordenable

Un conjunto totalmente ordenable, como su nombre lo indica, es aquel que se puede ordenar de manera no descendente con una función de comparación dada.

Podrímos vernos tentados a definir un conjunto A como totalmente ordenable en base a la existencia de una permutación P de A tal que $P = (p_1, ..., p_n)$ y $f(p_i, p_{i-1}) = 0$ para todo $0 < i \leq n$. Pero eso no es necesario

Definición 12.2.1 (Conjunto Totalmente Ordenable)**.** Un conjunto totalmente ordenable es un par ordenado (A, f) donde A es un conjunto y f es una función de comparación con dominio en $A \times A$. Además se aceptan las siguientes definiciones:

- $a < b$ si y solo si $f(a, b) = 1$ y se lee "'a menor que b"'
- $a > b$ si y solo si $f(b, a) = 1$ y se lee "'a mayor que b"'
- $a = b$ si y solo si $f(a, b) = 0$ y $f(b, a) = 0$ y se lee "'a equivale a b"'

- $a \leq b$ si y solo si $a < b$ ó $a = b$ y se lee "'a menor o igual que b"'
- $a \geq b$ si y solo si $a > b$ ó $a = b$ y se lee "'a mayor o igual que b"'

Es un buen momento para hacer una aclaración, las matemáticas actuales en lugar de definir una función de comparación, definen algo muy parecido llamado relación de órden total. Así que la definición que vamos a manejar de Conjunto Totalmente Ordenable es ligeramente diferente a la mas usada.

Si algún lector se siente ofendido por esto, puede consultar lo que es una relación de orden total y convencerse de que todas las propiedades que analizaremos en los conjuntos totalmente ordenados también se cumplirán con la definición tradicional. Estamos usando funciones de comparación ya que en este contexto nos resultará mas familiar abordar los algoritmos de ordenamiento.

Al principio del capítulo se dijo que no se iba a definir un conjunto totalmente ordenable en base a la existencia de una permutación P de A tal que $P = (p_1, ..., p_n)$ y $p_{i-1} \leq p_i$ para todo $0 < i \leq n$; y el motivo de esto es que dada las propiedades de una función de comparación es posible comprobar que dicha permutación existe; de hecho luego de estudiar el siguiente algoritmo será fácil de convencerse de que es capáz de ordenar cualquier conjunto totalmente ordenable.

Es imporante mencionar que las implementaciones que aparecen aquí de algoritmos de ordenamiento son para aplicarse en el caso mas común: donde se tienen todos los elementos en un arreglo. Pero como se verá mas adelante, las cosas no siempre son tan simples.

12.3. Algoritmo de Selección

El ordenamiento por selección es quizá el primer algoritmo de ordenamiento que a la mayoría de la gente se le puede ocurrir y se basa en un hecho bastante simple:

Teorema 15. *Para todo conjunto totalmente ordenable $C = (A, f)$ tal que A es un conjunto finito, existe al menos un elemento $n \in A$ tal que $n \leq a$ para todo $a \in A$, al cual llamaremos* ***mínimo*** *y se denota por $min(C)$.*

Demostración. Supongamos lo contrario que no exista un elemento n en A tal que $n \leq a$ para todo $a \in A$.Esto implicaría que para todo elemento $m \in A$ existe un elemento $m_1 \in A$ tal que $m_1 < A$.

Como $m_1 \in A$ entonces existiría otro elemento $m_2 \in A$ tal que $m_2 < m_1$, y existiría otro elemento $m_3 \in A$ tal que $m_3 < m_2$, por inducción podemos

concluir que para cualquier elemento $m \in A$ y cualquier entero k existirían k elementos $m_1, ..., m_k \in A$ tales que $m_k < m_{k-1} < ... < m_2 < m_1 < m$.

Ahora, usando la conclusión del párrafo anterior, sabemos que para cualquier elemento $m \in A$ deberían existir tambien $|A|$ elementos $m_1, ..., m_{|A|}$ tales que $m_{|A|} < ... < m_1 < m$. Por la propiedad de totalidad, tenemos que, todos los elementos $m_{|A|}, ..., m_1, m$ son distintos, por lo que el conjunto A debería tener mas de $|A|$ elementos, lo cual es una contradicción.

□

La idea del algoritmo de selección es, dada una sucesión S, encontrar el mínimo elemento de $(s_1, s_2, ..., s_n)$ e intercambiarlo con s_1, después encontrar el mínimo elemento de $(s_2, s_3, ..., s_n)$ e intercambiarlo con s_2.

No es dificil darse cuenta que luego de hacer k intercambios, los primeros k elementos de la sucesión estarán ordenados. Por lo tanto, luego de hacer n intercambios se tendrá toda la sucesión S ordenada.

El siguiente código muestra una implementación del algoritmo de selección con enteros.

Código 12.1: Seleccion con Enteros

```
1  void seleccion(int S[], int n){
2          int i, k, minimo;
3          for(i=0;i<n;i++){
4                  minimo=i;
5                  for(k=i;k<n;k++){
6                          if(S[k]<S[minimo])
7                                  minimo=k;
8                  }
9                  intercambia(S[i], S[minimo]);
10         }
11 }
```

La implementación en C con enteros resulta bastante clara; C++ tiene la capacidad de definir funciones de comparación que se llamen al usar los operadores <y >, pero se muestra una implementación en C para ilustrar el funcionamiento del algoritmo de manera general:

Código 12.2: Seleccion en General

```
1  void seleccion(tipoDatos S[], int n){
2          int i, k, minimo;
3          for(i=0;i<n;i++){
4                  minimo=i;
5                  for(k=i;k<n;k++){
```

```
6                       if(f(S[k], S[minimo])==1)
7                           minimo=k;
8                   }
9                   intercambia(S[i], S[minimo]);
10              }
11  }
```

Como se puede ver, la implementación general cambió muy poco, y eso es lo que hace útiles a las funciones de comparación y les da mucha utilidad a los algoritmos de ordenamiento. En general un algoritmo de ordenamiento solamente puede manipular los datos de dos formas: dados dos datos a y b, compararlos ó intercambiarlos.

12.4. Algoritmo de Inserción

Este algoritmo es otro que a la gente se le ocurre de manera natural, se trata del algoritmo usado para ordenar las cartas de una baraja: A lo largo del ordenamiento, en la mano izquierda se tiene un conjunto de cartas ordenado, y luego, para cada carta de la mano derecha, se mueve la carta hacia el lugar que le corresponda en la mano izquierda.

Por ejemplo, si en la mano izquierda se tienen 4 cartas numeradas como $1, 3, 8$ y 10 y en la mano derecha se tiene una carta con el número 5, esta carta se deberá colocar entre las cartas 3 y 8, para asi obtener el conjunto ordenado de cartas $1, 3, 5, 8$ y 10.

Inicialmente en la mano izquierda se coloca una sola carta, y es trivial que ese conjunto ya está ordenado, posteriormente se van insertando las cartas de una por una con el procedimiento antes descrito hasta tener todas las cartas en la mano izquierda.

Una manera mas general de describir este procedimiento es inicialmente tener una sucesión vacía de elementos ordenados A y un conjunto B con n elementos no ordenados, después insertar cada elemento de B en A de manera que la sucesión A se mantenga ordenada luego de cada inserción, y continuar de esa manera hasta haber insertado todo los elementos.

En seguida se muestra una implementación del algoritmo de inserción:

Código 12.3: Insercion

```
1  void seleccion(tipoDatos S[], int n){
2          int i, k;
3          for(i=0;i<n;i++){
4                  for(k=i;k>0 && f(S[k], S[k-1])==1;k--){
5                          intercambia(S[k], S[k-1]);
```

```
6                       }
7               }
8   }
```

12.5. Análisis de los Algoritmos de Selección e Inserción

Recordando lo que hace cada algoritmo, el algoritmo de selección busca repetidas veces el mínimo de un conjunto de datos no ordenados y cada que encuentra un mínimo, lo "quita" del conjunto no ordenado y lo "pone" al final del conjunto ordenado. Si hay n elementos, este procedimiento se repite n veces.

Mientras tanto, el algoritmo de selección mantiene una sucesión ordenada de datos y realiza n inserciones en esa sucesión.

Teorema 16. *Con una sucesión de n datos, el algoritmo de selección realiza n consultas y el algoritmo de inserción realiza n inserciones.*

En las implementaciones que se mostraron, una consulta requiere tiempo $O(n)$ y una inserción requiere tiempo $O(n)$, por lo que ambos algoritmos funcionan en tiempo $O(n^2)$. Sin embargo, existen algunas situaciones e implementaciones donde tanto las consultas por el mínimo como las inserciones son de complejidad $O(logn)$, haciendo que estos algoritmos tengan una complejidad $O(nlogn)$.

Pero por el momento consideraremos la complejidad de estos algoritmos como $O(n^2)$.

12.6. Ordenamiento en $O(n\ log\ n)$

Los dos algoritmos que acabamos de ver son de complejidad $O(n^2)$ y generalmente los algoritmos que a la gente se le ocurren de manera natural son de esa complejidad.

Existen taambien muchos algoritmos de ordenamiento de complejidad $O(n\ log\ n)$(de hecho es posible comprobar que no hay algoritmo basado en una función de comparación que sea mas rápido), por el momento solo nos concentraremos en uno de ellos llamado "mezcla".

Para poder conocer el algoritmo de mezcla primero hay que resolver el siguiente ejemplo:

Ejemplo 12.6.1. Encuentra un algoritmo que dadas dos sucesiones ordenadas de manera no descendente A y B, encuentre una sucesión ordenada de manera no descendente C tal que conste de los elementos de A y los elementos de B, el algoritmo deberá funcionar en tiempo $O(n)$ donde n es el tamaño de la sucesión C.

Solución Siguiendo la idea del algoritmo de selección, es posible que nos venga a la mente buscar en ambas sucesiones cual es el elemento mas pequeño.

Pero recordando que $a_1 \leq a_2 \leq a_3 \leq a_4...$ y $b_1 \leq b_2 \leq b_3 \leq b_4...$, podemos concluir que $min(a_1, a_2, a_3, ..., b_1, b_2, b_3, ...) = min(a_1, a_2)$.

Por lo cual, podemos encontrar el mínimo de ambas sucesiones en tiempo constante. La idea del algoritmo es encontrar el minimo de ambas sucesiones, "quitarlo" de la sucesión original e insertarlo al final de C, y luego repetir los pasos anteriores hasta que ambas sucesiones esten vacías.

Por cada inserción en C hay que encontrar el menor elemento de ambas sucesiones, lo cual toma tiempo $O(1)$ y posteriormente insertarlo al final de la sucesión, lo cual toma también tiempo $O(1)$.

Asi que en total hay n inserciones y n búsquedas del mínimo, lo cual nos da como resultado un algoritmo que funciona en $O(n)$.

A continuación se incluye la implementación del algoritmo para clarificar las cosas. Los prámetros na y nb indican el número de elementos de A y el número de elementos de B respectivamente.

Código 12.4: Función que mezcla dos sucesiones ordenadas en otra sucesion ordenada

```
1  void mezclar(tipoDatos A[], tipoDatos B[], tipoDatos C
       [], int na, int nb){
2          int a=0, b=0, c=0;
3          while(a<na || b<nb){//Mientras A tenga
               elementos ó B tenga elementos
4                  if(a<na && b<nb){ //Si tanto A como B
                       tienen elementos
5                          if(A[a]<B[b]){
6                                  C[c++]=A[a++];
7                          }else{
8                                  C[c++]=B[b++];
9                          }
10                 }else if(a<na){//Si solo A tiene
                       elementos
11                         C[c++]=A[a++];
12                 }else{//Si solo B tiene elementos
```

```
13                              C[c++]=B[b++];
14                      }
15              }
16      }
```

□

La idea del algoritmo de mezcla es ordenar los primeros $\lfloor \frac{n}{2} \rfloor$ elementos de la sucesión y los últimos $\lceil \frac{n}{2} \rceil$ elementos de la sucesión por separado. Y posteriormente *mezclar* ambas *mitades* de la sucesión para obtener la sucesión ordenada.

Y obviamente, para ordenar cada *mitad* de la sucesión por separado se procede recursivamente.

De manera mas específica, se implementará una función llamada *ordenMezcla*, que recibirá como argumentos dos arreglos, S y $Dest$ y dos enteros $inicio$ y fin. La función ordenará todos los elementos de $S[inicio..fin]$ y los colocará en $Dest[inicio..fin]$.

A continuación se muestra una implementación de la función *ordenMezcla*.

Código 12.5: Ordenamiento por Mezcla

```
1  void ordenMezcla(tipoDatos S[], tipoDatos Dest[], int
       inicio, int fin){
2          int piv, i;
3          if(inicio>=fin)
4                  return;
5          piv=(inicio+fin)/2;
6          for(i=inicio;i<=fin;i++)
7                  Dest[i]=S[i];
8          ordenMezcla(Dest, S, inicio, piv);
9          ordenMezcla(Dest, S, piv+1, fin);
10         mezclar(&S[inicio], &S[piv+1], &Dest[inicio],
               piv-inicio+1, fin-piv);
11 }
```

Utilizando inducción es muy fácil darse cuenta de que este algoritmo funciona, pero el análisis de complejidad de este algoritmo no luce tan sencillo.

Vamos a llamarle T a una función tal que $T(n)$ es una conta superior para el tiempo que puede tardar en ejecutarse $ordenMezcla(S, Dest, a, a+n)$ para $a, n \in \mathbb{N}$.

Tenemos que T es de complejidad $O(O(T(\lceil \frac{n}{2} \rceil)) + O(T(\lfloor \frac{n}{2} \rfloor)) + n)$. Por regla de la suma esto se puede simplificar a $O(T(\lceil \frac{n}{2} \rceil) + n)$, podriamos vernos tentados a decir que cuando $n = 2$ la complejidad es $O(n)$ y cuando $n > 2$ la complejidad sigue siendo $O(n)$; pero eso no sería el camino correcto ya que

"$\lceil \frac{n}{2} \rceil$" en esta notación representa una función y no solamente un argumento.

Al llegar a este punto parece difícil proceder con las reglas conocidas para calcular complejidad, así que habrá que volver a definir el valor de T en base a una constante H:

$$\begin{aligned} T(n) &= H \text{ cuando n=1} \\ & \quad H(2T(\frac{n}{2}) + n + 1) \end{aligned}$$

El siguiente arreglo muestra algunos valores de T evaluado en potencias de 2:

$$\begin{aligned} T(1) &= H(1) \\ T(2) &= H(3+n) \\ T(4) &= H(7+3n) \\ T(8) &= H(15+7n) \\ T(16) &= H(31+15n) \\ &... \end{aligned}$$

Estos valores sugieren que:

$$T(2^x) = H(2^{x+1} + 2^x - 1)$$

La ecuación anterior se puede comprobar fácilmente con inducción. Ahora, vamos a definir una variable n tal que $n = 2^x$, la ecuación anterior se expresaría de la siguiente manera:

$$\begin{aligned} T(n) &= H(n(x+1) + n - 1) \\ & \quad H(n(log_2(n) + 1) + n - 1) \\ & \quad H(nlog_2 n + 2n - 1) \end{aligned}$$

Y ahora, como H es una constante y por regla de la suma es fácil de ver que T es de complejidad $O(nlogn)$.

Cabe mencionar que la complejidad solamente fué comprobada para potencias de 2, sin embargo, es fácil darse cuenta que si $2^a \leq n \leq 2^{a+1}$ entonces $T(2^a) \leq T(n) \leq T(2^{a+1})$, lo que prueba que T es de complejidad $O(nlogn)$ para cualquier entero n.

Existe también un algoritmo muy popular de ordenamiento llamado "Quicksort" que en la mayoría de los casos suele ser tan rápido como "mezcla" y

requiere menos memoria. Pero no se hablará de ese algoritmo en este libro debido a la dificultad de analizar su tiempo de ejecución, si el lector quiere aprender a usarlo puede consultar otro libro y usarlo bajo su propio riesgo.

12.7. Problemas

12.7.1. Mediana

Tiempo Límite: 1 segundo

Un nuevo experimento espacial involucra a N objetos etiquetados desde 1 hasta N

Se sabe de antemano que N es impar.

Cada objeto tiene una fuerza distinta(aunque desconocida) expresada por un número natural.

El objeto con la fuerza mediana es el objeto X tal que hay exactamente tantos objetos con una fuerza mas pequeña que X como objetos con una fuerza mayor que X .

Desafortunadamente, la unica manera de comparar las fuerzas es por un dispositivo que, dados 3 objetos distintos determina el objeto con fuerza mediana tomando en cuenta solamente esos 3 objetos.

Problema

Escribe un programa que determine el objeto con la fuerza mediana.

Libreria

Dispones de una biblioteca llamada **device** con estas tres operaciones:

- **GetN**, deberá ser llamado una vez al principio sin parámetros; regresa el valor de N
- **Med3**, deberá ser llamado con tres etiquetas de objetos como parámetros; regresa la etiqueta del objeto con la fuerza media.
- **Answer**, deberá ser llamado una sola vez al final, con una etiqueta de objeto como argumento; reporta la etiqueta del objeto X

Puedes experimentar con una biblioteca de diseño propio

Ejemplo

Secuencia

2 5 4 3 1

Interacción

```
1. GetN() regresa 5.
2. Med3(1, 2, 3) regresa 3.
3. Med3(3, 4, 1) regresa 4.
4. Med3(4, 2, 5) regresa 4.
5. Answer(4)
```

Consideraciones

$0 < N < 1000$
No esta permitido hacer mas de 7777 llamadas a la funcion *Med3* por cada ejecución de tu programa.

Referencias

Este problema apareció por primera vez en la IOI del 2000, fue traducido por Luis Enrique Vargas Azcona durante el 2008 con propósitos de entrenamiento.

El valor máximo de n fué modificado debido a que la solución oficial no es determinista pero hay soluciones deterministas que funcionan con $N < 1000$(actualmente las soluciones no deterministas estan vetadas de la IOI).

Parte IV

Estructuras de Datos

En lo que se refiere a la resolución de problemas, muchas veces para plantear el problema imaginamos objetos y acciones que se relacionan entre si.

Cualquier lenguaje de programación tiene ya implementados tipos de datos básicos como lo son enteros cortos, enteros largos, punto flotante, caractéres, arreglos y matrices.

Sin embargo, a medida que nos adentramos más y más en la programación, esos tipos de datos dejan de ser suficientes, podemos plantear problemas que traten de cosas mas allá de los números y arreglos.

No debemos de desanimarnos y pensar que la computadora solo nos servirá para problemas de números y arreglos, ya que con un poco de creatividad podremos manejar una infinidad de objetos distintos.

Si bien una computadora solamente cuenta con herramientas para manejar números, caractéres y arreglos, es posible hacer una analogía *representando* ciertos objetos abstractos con arreglos y números e implementando funciones que *simulen* las acciones de estos objetos.

Las representaciones de estos *objetos abstractos* constan de una serie de datos y funciones para manipular esos datos, a estas dos cosas juntas se les llama estructuras de datos. A continuación conoceremos las estructuras de uso mas frecuente.

Capítulo 13

Pilas, Colas, Listas

Iniciaremos nuestro estudio de las estructuras de datos con estas tres estructuras que son por mucho, las mas sencillas de todas: pilas, colas y listas.

13.1. Pilas

Imaginemos este sencillo escenario: un mesero tiene platos de colores apilados; de vez en cuando el que lava los platos coloca un plato recién lavado sobre la pila de platos; y en otras ocaciones el mesero toma el plato que esta hasta arriba y sirve ahí la comida que ha sido preparada por el cocinero para posteriormente llevarla a su destino.

Si sabemos de qué color es el primer plato de la pila, en qué momentos el que lava los platos colocó platos sobre la pila(tambien sabemos el color de los que se van añadiendo), y en qué momentos el mesero retiró cada plato que se encontraba hasta arriba; podemos saber de qué color será el plato que le toca a cada cliente.

Una manera de saberlo podría ser, hacer una representación dramatica de los hechos; pero esto no es necesario, ya que tambien podríamos tomar un lapiz y un papel, y escribir una lista de los colores de los platos, posteriormente, ir escribiendo los colores de los platos que se pusieron en la pila al final de la lista, y borrar el ultimo color de la lista cada que un plato se retire.

No se necesita ser un gran matemático para pensar en hacer eso, sin embargo, en el momento de querer implementar un programa en C que lo reprodusca, nos encontramos con que no tenemos ninguna lista donde se coloquen y se quiten cosas del final, tenemos solamente arreglos, variables,

estructuras, apuntadores, etc.

Claro que podemos simular esta lista con las herramientas que nos proporciona C++ o incluso C, asi pues, este es un ejemplo de objetos(como la pila de platos) ligados a operaciones(como poner un nuevo plato o quitar un plato) que modifican al objeto, es decir, una estructura de datos.

Una pila, es la estructura de datos mencionada en este ejemplo, es decir, un altero de objetos. O mas objetivamente:

Definición 13.1.1 (Pila)**.** Estructura de datos que simula una lista en la cual solo se pueden realizar 2 operaciones: colocar un elemento al final, o quitar un elemento del final.

Lo unico que se puede hacer en una pila es colocar un objeto hasta arriba, o quitar el objeto que esta arriba, en el ejemplo anterior si se quita un objeto de abajo o del centro(lo mismo que si se intenta añadir uno), la pila colapsaría.

Si queremos programar algo similar, lo mas obvio es guardar la información de la pila en un arreglo, además en este ejemplo usaremos números para denotar los colores.

Imaginemos que el restaurant tiene en total 10 platos(un restaurant bastante pobre), ello nos indicaría que un arreglo de tamaño 10 podría guardar todos los platos sin temor a que el tamaño del arreglo no alcance.

Suponiendo que inicialmente hay 2 platos, uno de color 1, y otro de color 2, el arreglo debería lucir algo asi:

```
2 1 0 0 0 0 0 0 0 0
```

Si repentinamente el que lava los platos pone un plato hasta arriba de color 2, luego de ello el arreglo debería de lucir asi:

```
2 1 2 0 0 0 0 0 0 0
```

Si luego pone hasta arriba un plato de color 3, entonces el arreglo debería de quedar asi:

```
2 1 2 3 0 0 0 0 0 0
```

Pero si el mesero toma un plato de arriba, el arreglo estará de nuevo de esta manera:

```
2 1 2 0 0 0 0 0 0 0
```

Si el mesero vuelve a tomar un plato de arriba, el arreglo quedará de esta manera:

```
2 1 0 0 0 0 0 0 0 0
```

Para lograr esto, basta con declarar un arreglo y una variable de tal manera que el arreglo diga especificamente qué platos hay en la pila, y la variable cuántos platos hay.

Entonces, podemos representar los datos de una pila de la siguiente manera:

```
1 int pila[tamaño maximo];
2 int p=0;
```

En esta implementación supusimos que la pila consta únicamente de números enteros, sin embargo en la práctica cualquier tipo de datos es válido, incluso el funcionamiento es el mismo si se declaran varios arreglos para guardar elementos en la pila que consten de varias variables, por ejemplo, pares ordenados de números.

Cada que queramos añadir un elemento en la parte superior de la pila, es suficiente con esta línea de código:

```
1 pila[p++]=objeto;
```

Y cuando queramos retirar el elemento que este en la parte superior.

```
1 pila[--p]=0;
```

Hay que hacer notar casi siempre es suficiente decrementar al variable p para retirar un elemento de la parte superior, pero todo depende del problema.

Por último, como cultura general, en ingles a la pila se le llama *stack*, a la operación de poner un elemento en la parte superior se le llama *push*, y la de quitar un elemento se le llama *pop*.

Asi mismo, si la pila sobrepasa su tamaño máximo, el error devuelto es "stack overflow" o "desbordamiento de pila"(ahora ya sabemos qué quieren decir algunos errores comunes en aplicaciones inestables).

Vale la pena recordar que cada que el sistema operativo maneja también una pila para cada programa que se está ejecutando, en dicha pila se añaden las variables locales cada que se llama a una función y se retiran cuando se regresa de la función.

La implementación de dicha pila es muy parecida a la que se muestra aquí, esto es un indicio de que a pesar de ser una implementación muy sencilla tiene sus ventajas.

13.2. Colas

Imagina una conversación de chat entre 2 personas, aunque los conversantes no se den cuenta, existe algo llamado *lag*, es decir, el tiempo que tardan las 2 computadoras en mandarse y recibir los mensajes.

Dependiendo de la conexión, el *lag* puede variar entre menos de un segundo o incluso mas de un minuto.

Si por un momento, por falla del servidor, una de las 2 computadoras pierde la conexión, y en ese momento un usuario está intentando mandar varios mensajes, el programa de chat guardará los mensajes que el usuario está tratando de mandar, y cuando se recupere la conexión, el programa de chat mandará los mensajes en el mismo orden que el usuario los escribió.

Obviamente, el programa de chat no usa una pila para eso, ya que si usara una pila, el receptor leería los mensajes en el orden inverso que el emisor los escribió.

Es decir, en una pila el ultimo que entra es el primero que sale. De ahí que a las pilas se les conozca como estructuras LIFO(Last In, First Out).

Existen otras estructuras llamadas colas, en una cola el primero que entra es el primero que sale. Su nombre deriva de las filas que se hacen en los supermercados, cines, bancos, etc. Donde el primero que llega, es el primero en ser atendido, y el último que llega es el último en ser atendido(suponiendo que no haya preferencias burocráticas en dicho establecimiento).

Las colas, son conocidas como estructuras FIFO(First In, First Out).

Definición 13.2.1 (Cola)**.** Una cola es una estructura de datos que simula una lista, en la cual sólo se pueden aplicar estas dos operaciones: colocar un elemento al final, o quitar un elemento del principio.

Para representar una cola obviamente necesitamos tambien un arreglo. Nuevamente para ilustrar cómo utilizar el espacio del arreglo imaginaremos una situación divertida.

Supongamos que hay varios participantes en la cola para el registro de la OMI(Olimpiada Mexicana de Informática). Cada participante tiene un nombre que consiste de un número entero mayor o igual que 1(Son bastante populares esos nombres en nuestros días).

Ccomo faltan muchos estados a la OMI, solo habrá 10 participantes.

Entonces el arreglo podría lucir algo asi...

```
0 0 0 0 0 0 0 0 0 0
```

En ese momento llega 3 y se forma

```
3 0 0 0 0 0 0 0 0 0
```

Luego llega 5 y se forma

```
3 5 0 0 0 0 0 0 0
```

Despues llega 4 y se forma

```
3 4 5 0 0 0 0 0 0
```

Luego llega 9, y despues de eso llega 2

```
3 4 5 9 2 0 0 0 0
```

Entonces al encargado del registro se le ocurre comenzar a atender, y atiende a 3.

En la vida real, los participantes darían un paso hacia delante, pero en una computadora, para simular eso, sería necesario recorrer todo el arreglo, lo cual es muy lento; por ello, es mas practico dejar el primer espacio de la cola en blanco.

```
0 4 5 9 2 0 0 0 0
```

Luego se atiende a 4

```
0 0 5 9 2 0 0 0 0
```

En ese momento llega 1 corriendo y se forma

```
0 0 5 9 2 1 0 0 0
```

Y así continúa...

Ya para este momento, te debes de estar imaginando que para implementar una cola, unicamente se requiere un arreglo y dos variables, donde las variables indican donde inicia y donde termina la cola.

La mayoría de las veces podremos saber por adelantado cuantos elementos "se formarán" en la cola. Sin embargo, suponiendo que se forma alguien e inmediatamente es atendido, se vuelve a formar y vuelve a ser atendido inmediatamente, y asi 1 000 veces, entonces se requeriría un arreglo de tamaño 1 000, cuando nunca hay mas de un elemento dentro de la cola.

Para evitar eso, podemos añadir elementos al principio de la cola si ya no hay espacio al final.

Por ejemplo, si luego de que se atendió a muchos participantes, la cola está de esta forma:

```
0 0 0 0 0 0 0 0 7 3
```

Y para rectificar algo 5 vuelve a formarse, podemos colocar a 5 al principio

```
5 0 0 0 0 0 0 0 7 3
```

Luego si 4 vuelve a formarse

```
5 4 0 0 0 0 0 0 7 3
```

Puede parecer extraño esto, pero el programa sabrá que la cola empieza donde está 7 y termina donde está 4. Asi que si el organizador atiende al siguiente, atenderá a 7, y la cola quedará de esta manera

```
5 4 0 0 0 0 0 0 0 3
```

Luego atenderá a 3

```
5 4 0 0 0 0 0 0 0 0
```

Despues atenderá a 5

```
0 4 0 0 0 0 0 0 0 0
```

Y asi sucesivamente...

Implementar la operación de meter un elemento a la cola es muy sencillo:

```
1  cola[fin++]=elemento;
2  if(fin>=tamaño de la cola)
3          fin=0;
```

Y casí lo mismo es sacar un elemento de la cola

```
1   inicio++;
2  if(inicio>=tamaño de la cola)
3          inicio=0;
```

Figura 13.1: Representación gráfica de una lista enlazada

13.3. Listas

Frecuentemente necesitamos tener almacenadas unas k listas de datos en memoria, sabiendo que el número total de datos en memoria no sobrepasa n.

Si disponemos de suficiente memoria, podemos guardar en memoria n arreglos de tamaño k o una matriz de tamaño nk, pero no siempre dispondremos de tanta memoria.

Tambien hay veces que se requieren tener listas de números y agregar números a dichas listas pero no al final ni al principio, sino en medio.

Para solucionar estos y otros problemas, existen las listas enlazadas.

Las listas enlazadas son estructuras de datos compuestas por una sucesión de elementos llamados nodos; en la que cada nodo contiene un dato y la dirección del proximo nodo, en caso de que haya próximo.

La figura 13.3 muestra una representación gráfica de una lista enlazada.

Definición 13.3.1 (Nodo)**.** Se considera un nodo a cualquiera de estas dos cosas:

- Una estructura vacía ó
- Un elemento de información y un enlace a otro nodo.

La tarea de implementar una enlazada puede hacerse eficazmente con 2 arreglos: uno para guardar los datos y otro para guardar los enlaces, además se requiere una variable que diga el tamaño de la lista de la siguiente manera...

```
1 Tipo dato[tamaño maximo de la lista];
2 int proximo[tamaño maximo de la lista];
3 int tam_lista=1; //Tamaño de la lista
```

Lo único que falta definir es el elemento vacío, para ello, podemos definir que el dato 0 es el elemento vacío, y en el momento que nos encontremos con él, sabemos que la lista ya habrá terminado.

La ventaja de usar el 0 para representar el elemento vacío radica en que muchos arreglos se inicializan automáticamente en 0 y no es necesario inicializarlos después.

Si se mira con atención las tres líneas sugeridas para declarar la lista enlazada, es posible darse cuenta que el tamaño inicial de la lista es 1, a

pesar de que inicialmente la lista no tiene elementos. Esto se debe a que esa implementación está pensada para reservar el nodo 0 como el nodo vacío, por lo que inicialmente la lista consiste únicamente del nodo vacío.

Como toda estructura de datos, las listas tienen operaciones para manipular los datos, aquí contemplaremos un par de operaciones: insertar y recorrer.

Como el lector puede imaginar, usaremos una estrategia similar a la de la pila para determinar qué partes de los arreglos ir utilizando conforme se vayan insertando nuevos elementos.

Mas precisamente, una vez insertados n elementos, las primeras $n+1$ posiciones de los arreglos habrán sido usadas.

Una vez considerado todo esto, insertar un nodo con un dato x, justo despues de otro nodo k, se puede hacer facilmente en tiempo constante:

```
1  void insertar(Tipo x, int k){
2          dato[tam_lista]=x;
3          proximo[tam_lista]=proximo[k];
4          proximo[k]=tam_lista++;
5  }
```

Lo que hace este código es colocar x en el siguiente espacio en blanco dentro del arreglo `datos`, luego colocar un enlace al sucesor de k en el siguiente espacio en blanco dentro del arreglo `proximo`, y hacer que k apunte al nodo que se acaba de crear.

De esa forma k apuntará al nuevo nodo, y el nuevo nodo apuntará al nodo que apuntaba k.

Si observamos bien, esta implementación solo funciona cuando ya se tiene un nodo inicial en la lista, por lo cual es necesario crear el primer nodo mediante otro. La siguiente función crea un primer nodo de una lista asignándole el valor x y devuelve el índice del nodo:

```
1  int primer_nodo(Tipo x){
2      dato[tam_lista]=x;
3      proximo[tam_lista]=0;
4      return tam_lista++;
5  }
```

Una vez que tenemos creada una lista necesitamos consultar los datos de alguna manera, en una lista lo único que podemos hacer es consultar sus datos en órden, desafortunadamente tenemos que pagar el precio de no poder acceder directamente a los datos, pero por los motivos antes expresados, a veces vale la pena.

Al recorrer la lista se pueden hacer muchas cosas, aquí mostraremos simplemente cómo imprimir los elementos de la lista. El siguiente código imprime todos los datos contenidos en una lista, asumiendo que 1 es el primer elemento de dicha lista.

```
1        for(i=1;i!=0;i=proximo[i])
2                printf("%d", dato[i]);
```

Estas implementaciones estan algo límitadas, ya que se le da un trato especial al primer elemento de la listas y no se pueden borrar nodos entre otras cosas.

Muchos lenguajes de programación(por ejemplo C++) ya tienen previamente implementadas listas enlazadas, con implementaciones bastante mas completas que las mostradas aquí. El objetivo de mostrar las listas y las implementaciones es simplemente mostrar la idea de las listas enlazadas y una implementación sencilla, ya que juega un papel muy importante en el estudio de los algoritmos y sirven de base para entender otras estructuras de datos.

También hay que hacer notar que tanto las pilas como las colas pueden ser implementadas con listas enlazadas, pero su implementación es mas tediosa, su rendimiento es mas lento y además los enlaces hacia el siguiente elemento requieren cierta memoria adicional; la elección sobre qué implementaciones usar depende del problema en específico.

13.4. Problemas

13.4.1. Equilibrando Símbolos

Jorge está estudiando para ser un desarrollador de software y como tarea le han encargado elaborar un compilador.

Para hacerlo requiere de un algoritmo que verifique que la sintaxis del código fuente sea la correcta.

Una de las cosas que se requieren para que un código tenga una sintaxis correcta es que todos los paréntesis, corchetes y llaves estén correctamente emparejados, es decir, que por cada paréntesis, corchete o llave que abra exista una pareja correspondiente que cierre, y además que no se traslapen.

Jorge requiere implementar un algoritmo que reciba un programa como entrada y devuelva la posición del primer error sintáctico, considerando únicamente las posiciones de paréntesis ("(", ")"), corchetes ("[", "]") y llaves ("{", "}").

El código fuente consiste de una sola línea con N caracteres (incluyendo espacios en blanco) donde el primer carácter está en la posición uno, el segundo en la posición dos y así sucesivamente.

Algunos de los caracteres pueden ser; "(", ")", "[", "]", "{" o "}". Dos símbolos son correspondientes si son del mismo tipo y uno de ellos *abre* mientras el otro *cierra* (ej. "(" y ")" son símbolos correspondientes).

Un código contiene un error sintáctico si se cumple al menos uno de tres casos:

- Para alguno de los símbolos que abren no existe un símbolo correspondiente que cierre. (ej. $(a + b) + (c * [)$).
- Cuando se encuentre un símbolo que cierre sin estar antes su correspondiente que abra. (ej. ([a+b]) + c] * (2 + 2)).
- Cuando Entre dos símbolos correspondientes (uno que abra y otro que cierre del mismo tipo) exista un tercer símbolo de tipo diferente que abra y que no tenga su correspondiente en ese rango, es decir, cuando los símbolos se traslapen. (ej. $a(b + c * \{d/h)\}$).

Problema

Escribir un programa que dados N (el número de caracteres del código fuente a revisar incluyendo espacios en blanco), y el código en sí, determine si éste es correcto sintácticamente o no.

Entrada

- Linea 1: Un solo entero N
- N caracteres que representan el código.

Salida

- La palabra "SI" si el código es sintácticamente correcto, la palabra "NO" en caso contrario.

Ejemplo de Entrada

```
20
a(b + c * { d / h )}
```

Ejemplo de Salida

```
NO
```

Referencias

El problema fue escrito por Nephtali Garrido y utilizado en uno de los primeros examenes preselectivos de la Olimpiada Mexicana de Informática a finales del 2007.

13.4.2. Pirámides Relevantes

Hay N pirámides alineadas en línea recta.

Las pirámides estan numeradas de 1 a N, de izquierda a derecha.

Cada pirámide i tiene un nivel de relevancia R_i y tambien tiene una altura H_i.

Desde la parte mas alta de cualquiera de las pirámides, es posible ver todas las pirámides hacia la izquierda o hacia la derecha, siempre y cuando cuando no exista una piramide mas alta que obstruya la vista.

Es decir, una pirámide j se puede ver desde la parte mas alta de una pirámide i, sí y solo sí **NO** existe alguna k tal que $i < k < j$ ó $j < k < i$ para $H_k \geq H_i$.

Por ejemplo, si hay 6 pirámides con alturas 1, 3, 2, 1, 5, 2(de izquierda a derecha en ese orden) desde la parte mas alta de la pirámide 3(que tiene altura 2), se pueden ver las pirámides 2, 3, y 4 y 5 y las pirámides 1 y 6 no pueden ser vistas desde ahí ya que las pirámides 2(de altura 3) y 5(de altura 5) obstruyen la vista. En cambio, desde la parte mas alta de la pirámide 5 se pueden ver todas las pirámides, mientras que desde la parte mas alta de la pirámide 6, solamente se puede ver la pirámide 5.

Un guía de turistas ha pedido tu ayuda.

Problema

Escribe un programa que dadas las características de las pirámides, determine, para la parte mas alta de cada pirámide, cual es la pirámide mas importante que puede ser vista desde ahí.

Entrada

- Línea 1: Un solo número entero N.
- Siguientes N líneas: La línea $i + 1$ contiene los dos enteros H_i y R_i separados por un espacio.

Salida

- N números enteros, donde el $i-$ésimo entero representa la relevancia de la pirámide mas relevante que se pueda ver desde la parte mas alta de la pirámide i.

Entrada de Ejemplo

```
6
1 10
3 5
2 4
1 3
5 1
2 2
```

Salida de Ejemplo

```
10 10 5 4 10 2
```

Consideraciones

Puedes asumir que $2 \leq N \leq 1000000$.

Referencias

El problema fue escrito por Luis Enrique Vargas Azcona y usado en uno de los primeros exámenes preselectivos de la Olimpiada Mexicana de Informática a finales del 2007.

Capítulo 14

Árboles Binarios

Seguramente despues de haber visto las listas enlazadas te llegó a la mente la idea de colocar mas de un enlace en cada nodo.

No fuiste el primero en tener esa idea, pues los árboles binarios son estructuras de datos parecidas a las listas enlazadas, con la diferencia de que cada nodo puede tener hasta 2 enlaces(de ahí el nombre de binario).

Estas estructuras de datos tienen muchas aplicaciones, las mas frecuentes involucran ordenamiento.

Al igual que a las listas enlazadas, es posible ver a los árboles binarios como un conjunto de *nodos* iterconectados y a cada nodo se le puede asignar información.

La figura 14 muestra una representación gráfica de un árbol binario, como se puede ver, es igual que una lista enlazada excepto porque cada nodo puede tener hasta dos enlaces y no solamente uno.

Antes de seguir entrando en materia, será conveniente dar unas cuantas definiciones.

Por ahora nos reservaremos la definición general de árbol, pues requiere de teoría de grafos para comprenderla, sin embargo, un árbol binario se puede definir de la siguiente manera, sin recurrir a teoría de grafos:

Definición 14.0.1 (Árbol Binario)**.** Un árbol binario es un conjunto V de elementos llamados **nodos** o vértices, tal que:

Para todo $v \in V$, $v = (A, B)$ donde:

- A es el conjunto vacío o un enlace a otro vértice $a \in V$(en este caso decimos que v apunta a a), A recibe el nombre de **hijo izquierdo** de v
- B es el conjunto vacío o un enlace a otro vértice $b \in V$(en este caso decimos que v apunta a b), B recibe el nombre de **hijo derecho** de v.

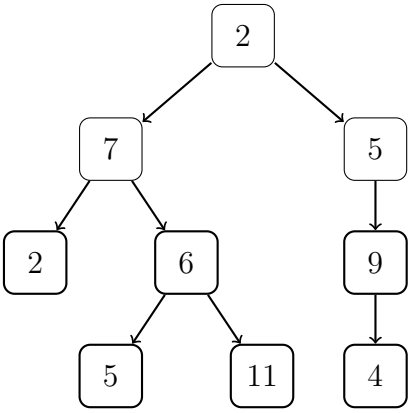


Figura 14.1: Representación gráfica de un árbol binario

- Hay un único elemento $r \in V$ tal que ningún otro vértice apunta a él; decimos que r es la **raíz** de V.

- Para todo $v \in V$ tal que $v \neq r$ existe un único $w \in V$ tal que w apunta a v al cual llamaremos **padre de v**.

- Para todo $v \in V$ tal que $v \neq r$ existe un camino desde la raiz hasta v(mas objetivamente, existen $w_1, ..., w_k \in V$ tales que $w_1 = r$, $w_k = v$ y w_i apunta a w_{i+1} para todo entero $1 \leq i < k$).

La definición anterior no es la que se utiliza en teoría de grafos para los árboles binarios y en cierto sentido tiene algunas propiedades distintas, sin embargo, los "árboles binarios" utilizados en programación son exactamente los que se acaban de definir.

El lector puede considerar esto una ambigüedad, y realmente lo es, pero puede estar tranquilo de que en este libro se hablará solo de los árboles binarios que se usan en programación evitando así toda ambigüedad.

Esto no quiere decir que nos vayamos a limitar en decir cómo programar los árboles binarios, sino que también exploraremos sus propiedades matemáticas.

Por lo general a cada nodo se le suele asociar un contenido llamado *clave*, sin embargo, esto no forma parte de la escencia de un árbol binario, ya que puede carecer de claves y aún así sigue siendo árbol binario.

Existen también otras definiciones que es conveniente conocer:

- Los enlaces que unen los nodos reciben el nombre de aristas.

- Se dice que un nodo B es hijo de un nodo A, si existe alguna arista que va desde A hasta B. Por ejemplo, en la figura, 7 es hijo de 2, 4 es hijo de 9, 11 es hijo de 6, etc.
- Se dice que un nodo A es padre de un nodo B si existe una arista que va desde A hasta B. Ej. 9 es padre de 4, 6 es padre de 5 y de 11, etc.
- Se dice que un nodo es hoja, si no tiene hijos. Ej. 11 y 4 son hojas, pero 6, 7, y 9 no lo son.
- La rama izquierda de un nodo es el árbol que tiene como raíz el hijo izquierdo de tal nodo, por ejemplo 7, 2, 6, 5, 11 son los nodos de la rama izquierda de 2.
- La rama derecha de un nodo es el árbol que tiene como raíz el hijo derecho de tal nodo.
- La altura de un árbol es la distancia(en áristas) de la hoja mas lejana a la raíz.

14.1. Implementación

Los árboles binarios tienen muchas formas de implementarse, cada una con sus ventajas y desventajas, por ahora solo trataremos una.

Pueden ser implementados utilizando 3 arreglos: clave(los datos que guarda cada nodo), hijo izquierdo e hijo derecho.

Y por supuesto otra variable entera: el número de nodos en el árbol.

```
1 Tipo clave[maximo de nodos]; //Dato que guarda el nodo
2 int izq[maximo de nodos]; //Hijo izquierdo
3 int der[maximo de nodos]; //Hijo derecho
4 int nodos=0;
```

En seguida se muestran los datos que podrían contener los arreglos para obtener el árbol que se muestra el la figura 14 :

nodo	0	1	2	3	4	5	6	7	8
clave	2	7	2	6	5	11	5	9	4
izq	1	2	-1	6	-1	-1	-1	8	-1
der	4	3	-1	5	7	-1	-1	-1	-1

Como se observa en la tabla, se está utillizando el nodo 0 como raíz, y el -1 para representar el vacío.

Se puede crear un nuevo nodo facilmente con la siguiente función:

```
1        int crear_nodo(Tipo dato){
2                clave[nodos]=dato;
3                izq[nodos]=-1;
4                der[nodos]=-1;
5                return nodos++;
6        }
```

Como se puede ver, la función recibe como parámetro el dato que va a tener el nuevo nodo y regresa el lugar en el arreglo donde está ubicado el nuevo nodo, depende del programador saber dónde poner el enlace para accesar a dicho nodo.

14.2. Propiedades Básicas

Los árboles binarios tienen muchas propiedades que pueden resultar útiles. Exploraremos varias de ellas.

Teorema 17. *Todo árbol binario con n nodos contiene exactamente $n-1$ aristas.*

Demostración. Sea A un árbol binario con n nodos.

Toda arista une a un nodo padre con un nodo hijo, es decir, el número de aristas es igual al número de nodos que tienen un padre.

Para todo $v \in A$ tal que v no es la raíz, existe un único nodo w tal que v es hijo de w. Por lo tanto, hay tantas aristas como nodos distintos de la raíz, es decir, hay $n-1$ aristas. □

Teorema 18. *Un árbol binario de altura n tiene a lo mas $2^{n+1}-1$ nodos.*

Demostración. Existe un solo nodo a distancia 0 de la raíz, existen a lo más 2 nodos a distancia 1 de la raíz, existen a lo más 4 nodos a distancia 3 de la raíz.

Si para algun k existen a lo más 2^k nodos a distancia k de la raíz, como cada uno de esos nodos puede tener a lo más 2 hijos, entonces existen a lo más 2^{k+1} nodos a distancia $k+1$ de la raiz.

Por inducción, existen a lo más 2^k nodos a distancia k de la raíz para todo $k \in \mathbb{N}$.

Por lo tanto, el número de nodos es $2^0 + ... + 2^n$, lo que por la ecuación 4.1 es igual a $2^{n+1}-1$.

□

Un árbol estrictamente binario es aquel en el que todo nodo tiene 2 hijos o bien no tiene hijos.

Teorema 19. *Un árbol estrictamente binario A tiene exactamente $\frac{|A|+1}{2}$ hojas.*

Demostración. Sea h el número de hojas que hay en el árbol, y sea m el número de nodos que no son hojas, es decir $m = |A| - h$.

Por cada nodo $v \in A$ tal que v no es una hoja, existen 2 aristas, por lo tanto el número de aristas es igual a $2m$.

Por el teorema 17, el árbol tiene exactamente $|A| - 1$ aristas. Es decir $|A| - 1 = 2m = 2|A| - 2h$ y así se concluye que:

$$h = \frac{|A| + 1}{2} \tag{14.1}$$

□

14.3. Recorridos

Aunque nos hemos dedicado a demostrar propiedades de los árboles binarios e incluso ya vimos cómo representarlos en un programa, aún no hemos visto ni una sola cosa útil que se pueda realizar con ellos.

Desafortunadamente el lector tendrá que esperar si quiere encontrar cosas *útiles*, pues antes tenemos que estudiar los recorridos en árboles binarios.

El tema de los recorridos en los árboles binarios suele parecer muy aburrido en un inicio, ya que su explicación carece de motivación. Por el momento el lector deberá de creer que dichos recorridos nos servirán muy pronto. Aún así se explicarán superficialmente ejemplos de dónde se usa cada recorrido.

Existen 3 formas recursivas distintas de visitar los nodos de un árbol binario:

- **Recorrido en Preorden**. Tambien llamado Búsqueda en Profundidad en la teoría de grafos, consiste en visitar primero el nodo actual, luego el hijo izquierdo y despues el hijo derecho. Puede ser implementado de la siguiente manera:

```
1        void preorden(int nodo){
2            if(nodo==-1)
3                return;
4            visita(nodo);
5            preorden(izq[nodo]);
6            preorden(der[nodo]);
7        }
```

Entre otras cosas, este recorrido es útil para saber cómo recorrería una persona, un robot, o culquier cosa que pueda trasladarse pero no teletransportarse una estructura similar a un árbol binario de búsqueda; ya que para llegar a los nodos hijos hay que pasar primero por el nodo padre. Dicho de otra manera, estos recorridos son *caminables*.

- **Recorrido en Orden**. En algunos lugares se le menciona como inorden, consiste en visitar primero el hijo izquierdo, luego el nodo actual, y finalmente el hijo derecho. El siguiente código lo ilustra:

```
1       void orden(int nodo){
2               if(nodo==-1)
3                       return;
4               orden(izq[nodo]);
5               visita(nodo);
6               orden(der[nodo]);
7       }
```

La principal aplicación de este recorrido es en el ordenamiento, dicha aplicación se tratará en la siguiente sección.

- **Recorrido en Postorden**. Como ya te imaginarás, consiste en visitar primero los nodos hijos y después el nodo actual.

```
1       void postorden(int nodo){
2               if(nodo==-1)
3                       return;
4               postorden(izq[nodo]);
5               postorden(der[nodo]);
6               visita(nodo);
7       }
```

A veces es necesario borrar los nodos al visitarlos, en caso de que fuera así, este recorrido tendría la característica de que solamente desaparecerían hojas del árbol y en ningun momento habría vértices aislados.

Es decir, este recorrido es indispensable cuando se requiere liberar la memoria de un árbol binario(aunque estas implementaciones no reserven y liberen dinámicamente la memoria, es muy probable que el lector lo necesite realizar en algun momento de su vida).

14.4. Árboles Binarios de Búsqueda

Los árboles binarios son bellos, pero no demuestran todo su potencial hasta que se utilizan como árboles binarios de búsqueda; los árboles binarios de búsqueda aparecieron con el propósito de mantener conjuntos de datos ordenados.

Ya hemos visto como ordenar conjuntos de datos, pero si en algun momento dado quicieramos agregar mas datos al conjunto, con lo que sabemos hasta ahora tendríamos que repetir todo el ordenamiento, o en el mejor de los casos usar inserción, como ya hemos visto, si una inserción toma tiempo lineal entonces N inserciones tomarían tiempo cuadrático.

El principal objetivo de los árboles binarios de búsqueda es poder hacer inserciones en $O(logN)$; lamentablemente los árboles que estudiaremos no alcanzan ese objetivo, pero al menos se *acercan* de cierta manera.

Definición 14.4.1 (Arbol Binario de Búsqueda)**.** Un árbol binario de búsqueda es un árbol binario A con una función de costo $w : A \to C$ donde C es un conjunto totalmente ordenable.

Además, para todo $v \in A$:

- Si v tiene hijo izquierdo(llamémosle i) entonces $w(v) \geq w(i)$
- Si v tiene hijo derecho(llamémosle d) entonces $w(v) \leq w(d)$

Al número $w(v)$ le vamos a llamar el **peso** de v o la **clave** de v.

El siguiente teorema revelará por qué el libro no deja de pregonar que los árboles binarios de búsqueda son útiles para el ordenamiento.

Teorema 20. *Un recorrido en órden en un árbol binario de búsqueda procesa los nodos en órden no descendente con respecto a sus claves.*

Demostración. Utilizando inducción, en un árbol binario de búsqueda con un solo nodo, solamente se visita un nodo, por lo cual el órden en que los nodos se visitan es no descendente.

Supongamos que para alguna k todos los árboles binarios de búsqueda con $k' < k$ nodos. Imaginemos un árbol binario de búsqueda que tiene exactamente k nodos.

Tanto su rama izquierda como su rama derecha son árboles binarios de búsqueda, y además, tienen menos de k nodos.

Al recorrer su rama izquierda, todos sus nodos se visitarán en orden no descendente, al visitar la raíz, por la definición de árbol binario de búsqueda, su clave será mayor o igual que todas las claves de los nodos ya visitados.

Finalmente, al recorrer la rama derecha, todos los nodos tendrán claves mayores o iguales que los nodos que ya se habían visitado, además, como la rama derecha es un árbol binario de búsqueda con menos de k nodos, también se visitarán dichos nodos en órden. □

Hasta este momento hemos visto cómo recorrer árboles binarios y cómo guardarlos en memoria, sin embargo, sin una manera de construirlos, esto resultaría inútil.

No vale la pena entrar en detalle respecto a cómo construir un árbol binario de búsqueda de un solo nodo. Una vez teniendo un árbol binario de búsqueda, es necesario poder agregarle mas nodos, ya que de otra manera de nada nos serviría que el recorrido en órden procesara las claves de un árbol binario de búsqueda en órden no descendente.

La inserción de un dato en un árbol binario de búsqueda, además de que implica crear un nodo, la parte mas *difícil* radica en saber dónde insertar el nuevo nodo.

Esa dificultad se supera sin mucho problema adoptando un procedimiento recursivo, llamaremos x a la clave que tendrá el nuevo nodo:

- Si x es mayor que la clave de la raíz y no hay rama derecha, entonces hay que crear un nodo con clave x que haga las veces de rama derecha.
- Si x es mayor que la clave de la raíz y hay rama derecha, entonces hay que insertar x en la rama derecha.
- Si x es menor que la clave de la raíz y no hay rama izquierda, entonces hay que crear un nodo con clave x que haga las veces de rama izquierda.
- Si x es menor que la clave de la raíz y hay rama izquierda, entonces hay que insertar x en la rama izquierda.

Si se diera el caso que x fuera igual a la clave de la raíz, x puede ser insertado en cualquiera de las dos ramas sin alterar la condición de árbol binario de búsqueda.

A continuación se muestra una implementación de dicho algoritmo:

```
1        void insertar(int nodo, Tipo dato){
2            if(clave[nodo]>dato){
3                if(izq[nodo]==-1){
4                    izq[nodo]=crear_nodo(
                         dato);
5                }else{
```

```
6                                   insertar(izq[nodo],dato
                                        );
7                           }
8                   }else{
9                           if(der[nodo]==-1){
10                                  der[nodo]=crear_nodo(
                                        dato);
11                          }else{
12                                  insertar(der[nodo],
                                        dato);
13                          }
14                  }
15          }
```

La función anterior, crea un nuevo nodo con clave *dato* y lo inserta en un árbol(o subárbol) que tiene raiz en nodo.

Asi que si queremos generar un árbol de búsqueda con las claves 5, 4, 8, 2, 1, lo unico que tenemos que hacer es:

```
1 raiz=crear_nodo(5);
2 insertar(raiz, 4);
3 insertar(raiz, 8);
4 insertar(raiz, 2);
5 insertar(raiz, 1);
```

Ahora podemos ver que un árbol binario de búsqueda permite mantener un conjunto de datos ordenado e insertar nuevos datos; pero el objetivo de los árboles binarios de búsqueda es poder hacer inserciones en tiempo logarítmico, lo cual aún no hemos logrado.

14.5. Encontrar un Elemento en el Árbol Binario de Búsqueda

Otra aplicación de los árboles binarios de búsqueda, es el hecho de saber si en el árbol existe un nodo con una determinada clave.

Nuevamente la recursión resuelve fácilmente este problema. Llamémosle x al valor que se está buscando:

- Si la clave de la raiz es x entonces x está en el árbol.
- Si la clave de la raíz es menor que x, entonces x está en el árbol si y solo sí x está en la rama derecha

- Si la clave de la raiz es mayor que x, entonces x está en el árbol si y solo sí x está en la rama izquierda

Estas 3 observaciones conducen instantanemente a la siguiente implementación recursiva:

```
1  bool esta_en_el_arbol(int nodo, Tipo x){
2          if(nodo==-1)
3                  return false;
4          if(clave[nodo]==x){
5                  return true;
6          }else if(clave[nodo]<x){
7                  return esta_en_el_arbol(der[nodo], x);
8          }else{
9                  return esta_en_el_arbol(izq[nodo], x);
10         }
11 }
```

Como se podrá notar, si el árbol tiene altura h, se visitarán a lo más h nodos para encontrar el nodo deseado, o bien, darse cuenta que no existe.

14.6. Complejidad

Respecto a la complejidad de la inserción en árboles binarios de búsqueda, hay mucho que decir. Ya que si se insertan los datos en órden ascendente o descendente, insertar un nuevo nodo requeriría $O(n)$; pero también es posible insertar los datos en otro órden de manera que cada inserción tome a lo más $\lceil log_2\ n \rceil$ llamadas recursivas.

Consideremos el ejemplo de insertar los números enteros desde 1 hasta 7.

Si se insertaran de la siguiente manera no obtendríamos algo no muy diferente a una lista enlazada:

```
1  raiz=crear_nodo(1);
2  for(i=2;i<=7;i++)
3          insertar(raiz, i);
```

De hecho, el árbol binario tendría una altura de 6 aristas.

Pero si se insertaran los nodos de la siguiente manera, obtendríamos algo muy diferente:

```
1  raiz=crear_nodo(4);
2  insertar(raiz, 2);
3  insertar(raiz, 1);
```

```
4   insertar(raiz, 3);
5   insertar(raiz, 6);
6   insertar(raiz, 5);
7   insertar(raiz, 7);
```

En este segundo caso, obtendríamos un árbol con altura de 2 aristas.

De esta manera, es posible que una inserción tome $O(log\ n)$ ó tome $O(n)$, dependiendo de cómo se haya construido el árbol, pero esa no es toda la dificultad, ya que aquí solamente presentamos 2 formas de construir un árbol, cuando en realidad hay $n!$ formas de construirlo, y además, puede que un mismo árbol se pueda construir de dos o mas maneras diferentes.

A estas alturas, obviamente nos gustaría saber, *en promedio*, ¿qué altura tiene un árbol binario de búsqueda?.

Prestando atención a lo que significa "en promedio", podemos definirlo como la suma de las alturas de los $n!$ árboles(dichos árboles se generan insertando los números de 1 a n en todos los órdenes posibles) entre $n!$.

Vamos a llamar S a una función $\mathbb{N} \to \mathbb{Q}$ tal que $S(n)$ sea el promedio de la altura al generar un árbol binario de búsqueda de n nodos.

Por definición, si $n > 1$:

$$S(n) = \frac{\sum_{i=1}^{n!} H(P(i))}{n!} \tag{14.2}$$

Donde $H(P(i))$ representa la altura del árbol generado por una permutación numerada como i(suponiendo que se numeraran todas las permutaciones de 1 a $n!$).

Es necesario tomar en cuenta que $F(n+1) \geq F(n)$, ya que todos los árboles con $n+1$ nodos se obtienen generando primero cada uno de los árboles con n nodos y añadiendoles otro nodo al final.

Con lo anterior, es fácil demostrar que la altura de un árbol con a nodos en la rama izquierda y b nodos en la rama derecha tiene en promedio altura de $S(max(a,b)) + 1$.

También hay que hacer la observación ahora de que se generan $(n-1)!$ árboles con cada una de las claves como raíz.

Por lo tanto, si $n > 1$ entonces:

$$S(n) = \frac{\sum_{i=0}^{n-1} S(max(i, n-i-1)) + 1}{n} \tag{14.3}$$

Si n es par:

$$S(n) = \frac{2\sum_{i=\frac{n}{2}}^{n-1} S(i)}{n} + 1 \tag{14.4}$$

Si n es impar

$$S(n) = \frac{2\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} S(i)}{n-1} + \frac{S(\lfloor \frac{n}{2} \rfloor)}{2} + 1 \tag{14.5}$$

Sabemos de antemano que $S(1) = 1$ y $S(0) = 0$.

Nuevamente buscaremos evaluaremos algunos valores de n e intentaremos adivinar:

- $S(2) = 2\frac{1}{2} + 1 = 2$
- $S(3) = 2\frac{2}{3} + \frac{1}{3} + 1 = 2 + \frac{2}{3}$
- $S(4) = 2\frac{4+\frac{2}{3}}{4} + 1 = 3 + \frac{1}{3}$
- $S(5) = 2\frac{6}{5} + \frac{2}{5} + 1 = 3 + \frac{4}{5}$
- $S(6) = 2\frac{9+\frac{4}{5}}{6} + 1 = 4 + \frac{4}{15}$
- $S(7) \approx 2\frac{11,4}{7} + \frac{2,67}{7} + 1 = 4,64$
- $S(8) \approx 5,00$
- $S(9) \approx 5,30$
- $S(10) \approx 5,60$
- $S(11) \approx 5,85$
- $S(12) \approx 6,11$
- $S(13) \approx 6,33$
- $S(14) \approx 6,55$
- $S(15) \approx 6,74$
- $S(16) \approx 6,94$

Aquí es mas difícil encontrar un patrón, el único patrón que parece aparecer es que S *crece cada vez mas lento*. Esto sugiere que la complejidad es logaritmica. Y al examinar los valores de las razones $\frac{S(16)}{S(8)}$, $\frac{S(8)}{S(4)}$ y $\frac{S(4)}{S(2)}$ parece haber una confirmación de dicha sospecha.

Por el momento no disponemos de suficientes herramientas para demostrar esto para cualquier n, sin embargo, con un programa es posible demostrarlo para todas las n menores o iguales 50 millones, lo cual debe ser suficiente para casi cualquier aplicación.

14.7. El Barrido, una Técnica Útil

El cálculo de los valores de S tampoco es algo trivial, ya que S está definida a través de una sumatoria, y si se calcula esa sumatoria cada vez que se quiera calcular el valor de $S(i)$ para alguna i, requeriría en total un tiempo cuadratico, ya que para calcular $S(i)$ es necesario haber calculado antes $S(1), S(2), ..., S(i-1)$.

Para poder calcular los valores de S en tiempo $O(n)$ haremos uso de una técnica que comunmente la denominan *barrido* o *ventana deslizante*.

Un barrido normalmente consiste en obtener el valor de una función evaluada en un intervalo a partir del valor de la misma función evaluada en un intervalo parecido.

No existe una regla general de cuándo es conveniente aplicar un barrido, pero en el cáluculo de S, como lo veremos a continuación, si es conveniente.

Otra forma de definir S es como:

$$S(i) = \frac{F(i)}{i} + 1 \tag{14.6}$$

Cuando i es par y

$$S(i) = \frac{F(i)}{i} + \frac{S(\lfloor \frac{i}{2} \rfloor)}{i} + 1 \tag{14.7}$$

Cuando i es impar.

En las dos ecuaciones anteriores se está utilizando $F(i)$ para denotar $\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} S(i)$.

La observación clave para lograr calcular S en tiempo lineal es que si $i \geq 1$ entonces $F(2i) = F(2i-1) + S(2i-1)$ y $F(2i+1) = F(2i) - S(i) + S(2i)$. Es decir, si se sabe de antemano cuales son los valores de $F(1), ..., F(i)$ y el valor de $S(i)$, calcular los valores de $S(i+1)$ y $F(i+1)$ se puede hacer en tiempo constante.

Por lo cual, el algoritmo consistiría en llenar un arreglo S con los valores que se vayan encontrando de la función S y mantener una variable *num* con el valor de $F(i)$.

Para que este algoritmo funcione es necesario primero calcular $S(1)$, luego calcular $S(2)$, después calcular $S(3)$, etc. Esta forma de calcular valores de funciónes recursivas de abajo hacia arriba es conocida como "Programación Dinámica".

En seguida se muestra una implementación del algoritmo:

```
1 double S[50000001];
2 double num=0.0;
```

```
3
4   int main(){
5           double i;
6           S[0]=0;
7           S[1]=1;
8           for(i=2;i<=50000000;i++){
9                   num+=S[int(i)-1];
10                  if(int(i)%2==0){
11                          S[int(i)]=(2.0*num)/i+1.0;
12                  }else{
13                          num-=S[int(i/2)];
14                          S[int(i)]=(2.0*num)/i+S[int(i
                                /2)]/i+1.0;
15                  }
16          }
17          return 0;
18  }
```

Utilizando la ascersión $assert(S[int(i)] <= 3{,}2 * log(i))$ en el programa anterior, es posible comprobar que $S(i) \leq 3{,}2ln\ i$ para toda i entre 1 y 50 millones. De ahí llegamos al siguiente teorema

Teorema 21 (Altura promedio de un árbol binario de búsqueda)**.** *El valor esperado de la altura de un árbol binario de búsqueda construido al azar es menor o igual a* $3{,}2log(n)$ *donde* n *es el número de nodos del árbol.*

Ahora hemos confirmado como los árboles binarios de búsqueda se acercan bastante a su objetivo inicial de hacer inserciones en tiempo logarítmico, ya que un árbol binario de búsqueda construido al azar con menos de 50 millones de vertices probablemente tendrá una altura *razonable*.

Como podrá intuir el lector, S es una función cuya complejidad es $O(logN)$, sin embargo la demostración de dicho resultado va mas allá de todo lo que se pueda ver en este libro, y para sentirnos bien con nuestra conciencia, no utilizaremos ese resultado sino el teorema 21.

14.8. Problemas

14.8.1. Ancho de un Arbol

Tiempo Límite: 1 segundo

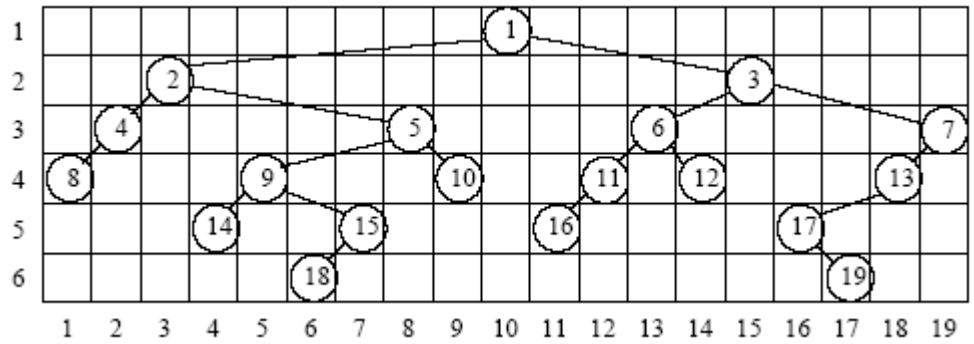
Descripcion

Supon que deseas dibujar un arbol binario en una cuadricula cuyas columnas estan numeradas de acuerdo a las siguientes reglas:

- Todos los nodos en un mismo nivel deberan estar en la misma fila
- Cada columna de la cuadricula puede tener solamente un nodo
- Los nodos en el subarbol izquierdo de un nodo deberan ser dibujados en una columna a la izquierda del mismo, al igual los nodos del subarbol derecho deberan ser dibujados en columnas a la derecha del mismo
- Al dibujar el arbol no debe quedar ninguna columna sin nodo entre la columna mas a la izquierda y mas a la derecha del dibujo.

El ancho de un nivel se puede obtener restando el numero de la columna derecha menos la columna izquierda del mismo mas uno. La raiz del arbol se considera el nivel 1.

La siguiente figura muestra un arbol binario dibujado de acuerdo a la siguiente regla. El ancho del primer nivel es uno mientras que el ancho del segundo es 13, el tercer, cuarto, quindo y sexto nivel tienen los anchos 18, 18, 13 y 12 respectivamente.



Escribe un programa que al dibujar un arbol de esta forma calcule cual es el nivel mas ancho del arbol, si dos niveles tienen el ancho maximo, como en el caso del ejemplo el nivel 3 y el 4, entonces debes tomar el nivel con menor numero.

Entrada

Tu programa debera leer del teclado los siguientes datos, la primera linea contendra un numero N entre 1 y 1,000 que indica el numero de nodos del arbol.

Cada una de las siguientes N lineas contiene 3 enteros, denotando 3 nodos, donde el primer numero indica un nodo, el segundo y tercer numeros de la linea indican el nodo izquierdo y derecho del nodo respectivamente.

Cada nodo esta numerado del 1 al N. Si hay un nodo que no tenga hijos, entonces su hijo tendra el numero -1. El nodo raiz tiene el numero 1.

Salida

Tu programa debera escribir a la pantalla dos numeros en una linea separados por un espacio, el primer numero indica el nivel con el ancho maximo, mientras que el segundo numero indica el ancho del nivel. Si hay mas de un nivel con el ancho maximo imprime el nivel de menor numero.

Ejemplo

Entrada

```
19
1 2 3
2 4 5
3 6 7
4 8 -1
5 9 10
6 11 12
7 13 -1
8 -1 -1
9 14 15
10 -1 -1
11 16 -1
12 -1 -1
13 17 -1
14 -1 -1
15 18 -1
16 -1 -1
17 -1 19
18 -1 -1
19 -1 -1
```

Salida

```
19
1 2 3
2 4 5
3 6 7
4 8 -1
5 9 10
6 11 12
7 13 -1
8 -1 -1
9 14 15
10 -1 -1
11 16 -1
12 -1 -1
13 17 -1
14 -1 -1
15 18 -1
16 -1 -1
17 -1 19
18 -1 -1
19 -1 -1
```

Limites

N siempre estara entre 1 y 1,000

Referencias

Este problema apareció en el concurso "VU atrankinis turas 2005 m. ACM ICPC var¸yboms"

Posteriormente fue utilizado en los preselectivos 2004-2005, que fueron organizados donde Cesar Arturo Cepeda García y Francisco Javier Zaragoza Martínez se encargaron de elegir y traducir los problemas.

14.8.2. Cuenta Árboles

Tiempo Límite: 1 segundo

Problema

Escribe un programa que dado N, determine cuantos árboles binarios se pueden formar con exactamente N nodos.

Entrada

Una línea con un único número: N.

Salida

La primera y única línea deberá contener un solo entero: el número de árboles que se pueden formar mod 1000000.

Entrada de Ejemplo

3

Salida de Ejemplo

5

Consideraciones

$1 \leq N \leq 1000$

Referencias

Este es un problema clásico de combinatoria de conteo, esta redacción fue utilizada por primera vez para un examen preselectivo de la Olimpiada Mexicana de Informática a finales del 2005.

Capítulo 15

Grafos

Muchos problemas, nos presentan algun conjunto, en el cual algunos pares de elementos de dicho conjunto se relacionan entre sí. Ese tipo de problemas son muy estudiados en la teoría de gráficas o teoría de *grafos*.

Estrictamente hablando, estas estructuras se llaman gráficas, sin embargo la palabra puede generar confusión con la gráfica de una función. Por tal motivo en este libro nos referiremos a estas estructuras como *grafos*.

El primer ejemplo de grafos se utilizará en algo no muy relacionado con la programación, esto se hace para evitar que el lector se vea tentado a confundir el concepto de grafo con el de la implementación de un grafo; pues hay problemas donde es necesario utilizar grafos en el razonamiento y no en la implementación.

Ejemplo 15.0.1. En una reunión hay n invitados, se asume que si un invitado a conoce a un invitado b, b tambien conoce a a, demuestra que en la reunión hay al menos 2 invitados que conocen al mismo número de invitados. Asúmase tambien que todos conocen por lo menos a alguien de la reunión.

Solución En el ejemplo anterior, tenemos un conjunto de n personas, y como vemos, algunos pares de personas se conocen entre sí.

Para resolver este tipo de problemas es muy útil usar grafos, tambien llamados "graficas" por algunos, sin embargo, aquí se utilizará el término "grafo", para evitar que se confunda con el concepto de la gráfica de una función.

Una forma un tanto pobre de definir un grafo(poco después trataremos la definición formal) es:

Definición 15.0.1 (Definición Provisional de Grafo)**.** Conjunto de puntos llamados vértices, en el cual algunos vertices pueden estar unidos por líneas

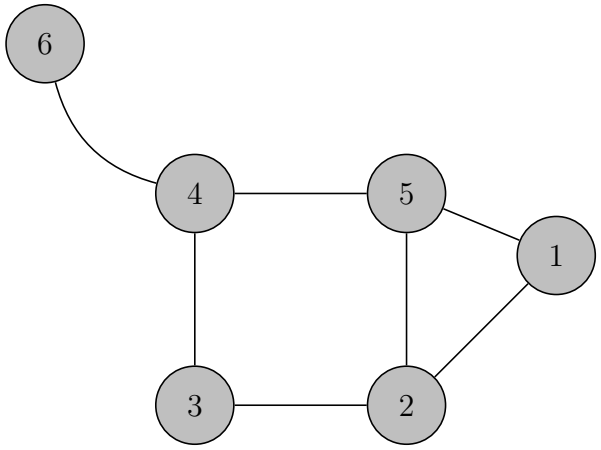


Figura 15.1: Dibujo de un grafo

llamadas aristas

Volviendo al ejemplo, es posible dibujar un punto por cada persona de la reunión, y entre cada par de personas que se conozca entre sí, colocar una línea.

Por ejemplo si 6 conoce a 4, 4 conoce a 5, 5 conoce a 1, 5 conoce a 2, 2 conoce a 1, 3 conoce a 2 y 4 conoce a 3, se podría dibujar un grafo similar al de la figura 15

Ahora, una vez que tenemos visualizada de cierta manera la solución, podemos ver que cada nodo puede estar conectado con al menos otro nodo(cada invitado conoce por lo menos a otro invitado) y a lo mas $n-1$ nodos.

Es decir, el número de conocidos de cada persona va desde 1 hasta $n-1$. Como cada persona puede tener $n-1$ números distintos de conocidos y hay n personas, por principio de las casillas al menos dos personas deberán tener el mismo número de conocidos.

□

15.1. ¿Qué son los grafos?

Por si alguien no leyó la solución del ejemplo. Hay que mencionar que un grafo puede imaginarse como un conjunto de puntos llamados vértices, en el cual algunos vertices pueden estar unidos por líneas llamadas aristas.

Los grafos son muy útiles en problemas que involucran estructuras tales como carreteras, circuitos electricos, tuberías de agua y redes entre otras cosas, y como lo acabamos de ver, con los grafos incluso se pueden representar las relaciones en una sociedad.

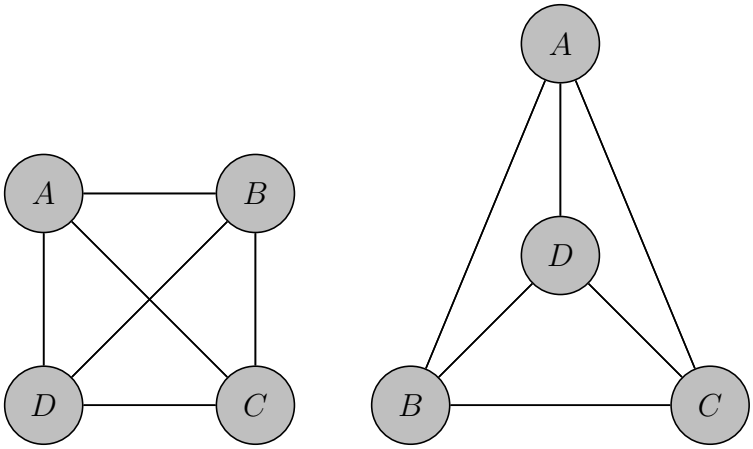


Figura 15.2: Dos dibujos de un mismo grafo conocido como K_4

Si eres observador, tal vez ya te diste cuenta que tanto las listas enlazadas como los árboles son grafos.

Hay que hacer notar que no importa cómo se dibujen los grafos, si las aristas estan curvas o rectas, o si los vertices estan en diferente posición no es relevante, lo unico que importa qué pares de vertices estan unidos.

Por ejemplo, la figura 15.1 muestra dos maneras distintas de dibujar un mismo grafo.

Es decir, un grafo, mas que un conjunto de puntos unidos por líneas, es un conjunto de objetos en el cual algunos pares de estos se relacionan entre sí, mas formalmente:

Definición 15.1.1 (Grafo)**.** Par ordenado $G = (V, A)$, donde los elementos de V son llamados vertices, y A es un conjunto de pares no ordenados de vértices llamadas aristas.

Aunque hay quienes se dedican a investigar grafos infinitos, en este libro nos dedicaremos exclusivamente a grafos finitos.

De esa forma, el grafo que se dibuja arriba se puede expresar como:

$$G = (\{A, B, C, D\}, \{\{A, B\}, \{A, D\}, \{D, B\}, \{D, C\}, \{B, C\}, \{A, C\}\})$$

Esto no quiere decir que para manejar grafos hay que olvidarse completamente de dibujarlos como puntos unidos por líneas, ya que en un dibujo a simple vista se pueden distinguir detalles del grafo que no serían tan facil distinguirlos sin el dibujo.

Antes de proseguir es conveniente ver algunas definiciones relacionadas con los grafos.

La primera que veremos será la de *grado*, o *valencia*, al hablar del grado de un vértice, nos referimos al número de "extremos de arista" conectados con él, es decir:

Definición 15.1.2 (Grado)**.** Para un grafo $G = (V, A)$ el grado de un vertice $v \in V$ es el número de aristas de la forma $\{v, w\} \in A$ tales que $v \neq w$ más dos veces el número de aristas de la forma $\{v, v\} \in A$.

El grado de v se denota como $\delta(v)$.

Por ejemplo, en la figura 15 el nodo etiquetado como 1 tiene grado 2, y el nodo etiquetado como 4 tiene grado 3, mientras que en la figura 15.1 todos los nodos tienen grado 3.

Un grafo dirigido(o digrafo) informalmente hablando es un grafo en el cual las aristas tienen dirección, es decir una arista entre v_i y v_j no implica una arista entre v_j y v_i.

La única diferencia es que en lugar de que las aristas sean conjuntos de uno o dos elementos, las aristas serán pares ordenados.

Definición 15.1.3 (Digrafo)**.** Par ordenado $G = (V, A)$ donde $A \subseteq V \times V$.

Nuevamente cabe aclarar que en este libro asumiremos también que los digrafos son finitos.

15.2. Propiedades Elementales de los Grafos

Los grafos también tienen muchas propiedades interesantes, la primera de ellas se demostró en el ejemplo 15.0.1, por lo cual solamente la vamos a enunciar con propósitos de referencia:

Teorema 22. *Sea $G = (V, A)$ un grafo tal que $\{a, a\} \notin A$ para toda $a \in V$(no hay aristas que unan a un vértice consigo mismo), entonces existen dos vértices con el mismo grado.*

La siguiente propiedad también tiene que ver con los grados de los vértices un grafo:

Teorema 23. *Sea $G = (V, A)$ un grafo se tiene que la suma de los grados de todos los vértices en V es exactamente $2|A|$.*

Demostración. El grado de un vértice es el número de aristas incidentes a él, donde las aristas cuyo origen y destino son el mismo vértice se cuentan doble.

Como cada arista está conectada a lo más con dos vértices entonces, si está conectada con dos vértices aporta la cantidad de 1 al grado de cada vértice, aportando en total 2 a la suma de todos los grados; si está conectada con un solo vértice, entonces aporta 2 al grado de ese vértice y por tanto 2 a la suma de todos los grados.

Por lo tanto la suma de todos los grados es el doble del número de aristas($2|A|$). □

Teorema 24. *Sea $G = (V, A)$ un grafo, el número de vértices de grado impar es par.*

Demostración. Usando el teorema anterior, sea z la suma de los grados de G entonces $z = 2|A|$, además notamos que $z = \delta(v_1) + \delta(v_2) + ... + \delta(v_n)$ donde $V = \{v_1, ..., v_n\}$.

Como z es par entonces solamente puede ser obtenida como suma de una cantidad par de impares y 0 o más pares, por lo tanto, el número de vértices de grado impar es par. □

15.3. Implementación

Luego de haber visto la naturaleza de los grafos, seguramente estas pensando en cómo implementar una estructura de datos similar a un grafo.

Hay muchas maneras de implementar grafos, pero solo se tratarán las 2 mas usadas: matriz de adyacencia y listas de adyacencia.

- **Matriz de Adyacencia**

  Esta es quizá la forma mas común de representar un grafo en un lenguaje de programación debido a su sencillez, sin embargo, requiere $|V|^2$ de memoria, y en la mayoría de los casos es un poco lenta.

  Un grafo $G = (V, A)$ puede ser representado como una matriz M de $|V| \times |V|$, donde $M_{i,j} \neq 0$ si existe una arista en A que une a los nodos v_i y v_j.

  Por ejemplo, el grafo de la Figura G1 puede ser representado como:

$$
\begin{aligned}
M = \{ & \\
& 010010 \\
& 101000 \\
& 010100 \\
& 001010 \\
& 110100 \\
& 000100 \\
\} &
\end{aligned}
$$

La implementacion de un grafo por matriz de adyacencia es bastante simple:

```
1   int m[número_de_vertices][número_de_vertices];
```

- Listas De Adyacencia

  Consisten en $|V|$ listas enlazadas, es decir, una lista enlazada para cada vertice $v \in V$, dicha lista consta de los nodos que estan unidos a v por medio de alguna arista. La ventaja fundamental de las listas de adyacencia sobre la matriz de adyacencia es que las listas de adyacencia solo requieren $|A| + |V|$ de memoria.

  Sin embargo, la representación de un grafo por listas de adyacencia es un poco mas compleja y muchos se ven tentados a usar memoria dinamica para estos casos. Nuevamente se presentará una implementación con memoria estática por los motivos que se mencionan en el prefacio.

  Si tenemos un grafo de 100 000 nodos, y a lo mas un millón de aristas, es obvio que no podemos crear 100 mil listas reservando espacio para 100 mil nodos en cada una, ya que requeriría una gran cantidad de memoria y es muy probable que carezca de ella la computadora donde se vaya a ejecutar el programa.

  La solución para esto, es guardar los datos de todas las listas en un solo arreglo, y guardar cual es el primer elemento de cada lista en otro arreglo.

  Además, será necesario tener una variable que indique el número de aristas de la lista.

  En el siguiente código ilustra cómo implementar un grafo con a lo mas 100 000 nodos y a lo mas un millón de aristas:

```
1  int dato[1000001]; //El nodo hacia el cual apunta
       la arista
2  int proximo[1000001]; //El siguiente elemento de
       la lista
3  int primero[100001]; //El primer elemento de la
       lista de cada nodo
4  int aristas=1; //El número de aristas
```

Aquí se declaró como una arista inicialmente para dejar libre el 0 para expresar el elemento vacío que indica el final de toda lista.

El procedimiento de insertar una arista entre 2 nodos también es algo tedioso con listas de adyacencia comparandolo a lo simple que es con matriz de adyacencia, la siguiente función inserta una arista entre los nodos v y w en un grafo dirigido.

```
1          void insertar_arista(int v, int w){
2                  dato[aristas]=w;
3                  proximo[aristas]=primero[v];
4                  primero[v]=aristas++;
5          }
```

Si se quiere que el grafo sea no dirigido simplemente hay que llamar a la función anterior 2 veces

Como se ve en el código, la arista siempre se inserta al principio de la lista de adyacencia, para que de esa forma, el conjunto de todas las aristas se inserte en tiempo lineal; ya que si se recorriera la lista de adyacencia cada que se quisiera insertar una nueva arista, insertar todo el conjunto de aristas se realizaría en tiempo cuadratico, queda como tarea para el lector demostrar por qué.

15.4. Recorridos en Grafos

Ahora que ya conocemos los grafos y cómo representarlos en una computadra comenzaremos con una de sus aplicaciones mas básicas: los recorridos en grafos o búsquedas.

La idea de “recorrer un grafo” se desprende de imaginar los nodos como cuartos y las aristas como pasillos que conectan cuartos, y el “recorrido” es visitar caminando todos los cuartos.

A estos recorridos les llamaremos caminos, los cuales se formalizan mediante la siguiente definición:

Definición 15.4.1 (Camino)**.** Un camino C es una sucesión de vertices $C = (v_1, v_2, v_3, ..., v_n)$, tal que para toda $0 < i \leq n$ existe una arista que conecta v_i con $v_i + 1$.

Si es posible iniciar en un cuarto y visitar todos los demás entonces se dice que grafo es conexo. O dicho de una manera mas precisa:

Definición 15.4.2 (Grafo conexo)**.** Un grafo $G = (V, A)$ es conexo si para cada par de vertices $v, w \in V$ existe un camino que inicia en v y termina en w.

Básicamente las búsquedas sirven para visitar todos los vertices de un grafo conexo, o bien de un subgrafo conexo de manera sistemática.

Es decir, iniciamos en el vertice A, y queremos saber desde ahí, a cuáles vertices se puede llegar, y la manera de hacerlo es... ¡Buscando!.

El primer recorrido que veremos será la busqueda en profundidad.

La busqueda en profundidad es un recorrido de grafos de naturaleza recursiva(aunque se puede implementar de manera no recursiva).

¿Recuerdas la antigua leyenda del minotauro? En caso de que no la recuerdes aquí va la parte que interesa:

Había un minotauro(un hombre con cabeza de toro) dentro de un laberinto, un valiente joven, llamado Teseo se dispuso a matar al minotauro(el minotauro mataba gente, asi que no tengas lastima del minotauro), pero para no perderse en el laberinto, tomó una cuerda para ir marcando por donde ya había pasado, y saber qué recorrido había seguido, para así poder regresar. Luego cuando mató al minotauro regresó siguiendo su rastro, como lo había planeado, y el resto ya no tiene que ver con el tema.

¿Qué moraleja nos deja esta historia?

Basicamente que si nos encontramos en un laberinto(o en un grafo), es necesario ir marcando el camino por el que ya pasamos y saber por dónde veníamos.

La idea de la búsqueda en profundidad es dejar una marca en el nodo que se esta visitando; y despues visitar recursivamente los vecinos de dicho nodo que no hayan sido visitados.

El siguiente codigo es una implementacion de la búsqueda en profundidad en un grafo con N nodos y con matríz de adyacencia g:

```
1  void visitar(int nodo){
2          visitado[nodo]=1;
3          for(i=1;i<=N;i++)
4                  if(g[nodo][i]!=0 && visitado[i]==0){
5                          visitar(i);
```

```
6                       }
7   }
```

Hay que mencionar que en un grafo $G = (V, A)$, la búsqueda en profundidad funciona con matriz de adyacencia en $O(|V|^2)$ mientras que con listas de adyacencia funciona en $O(|V| + |A|)$.

El otro recorrido que se va a tratar aquí es la búsqueda en amplitud(algunos lo llaman búsqueda a lo ancho).

Lamentablemente la búsqueda en amplitud no fue idea de Teseo, y como te imaginarás, es un poco mas difícil que a alguien se le ocurra hacerla si no la conoce de antemano.

Para comenzar a adentrarnos en la búsqueda en amplitud es conveniente que imagines la siguiente situación:

Eres miembro de una banda de bándalos(valga la redundancia), y la banda tiene como centro de reunión una esquina de la ciudad. Los miembros de la banda quieren que toda la ciudad sepa a cuántas cuadras se encuentran en cada momento de "su terrirtorio", asi que deciden dejar pintados mensajes en la paredes de cada esquina de la ciudad diciendo a cuántas cuadras de encuentran del "territorio prohibido".

Ahora, el jefe de la banda te ha encargado a ti determinar qué número se debe de pintar en cada esquina de la ciudad.

La tarea no es facil, ya que si haces rodeos, podrías llegar a creer que estas mas lejos de lo que realmente te encuentras, y la ciudad no es muy rectangular que digamos.

Conviene al lector detenerse unos momentos a reflexionar sobre este problema y pensar en una posible solución antes de leerla.

Lo que deberías hacer en ese caso, es pintar una advertencia "Alejate, este es nuestro territorio" en la esquina donde la banda tiene su centro de reunión, luego caminar una cuadra en cualquier dirección y pintar algo asi como "estas a una cuadra de territorio prohibido", después regresar al punto de encuentro y caminar una cuadra en otra dirección y otra vez pintar lo mismo, repitiendo esa operación hasta que ya se haya caminado una cuadra en todas las direcciones.

Después, para cada esquina donde hayas escrito "estas a una cuadra de terrirtorio prohibido", caminas una cuadra en todas las direcciones y escribes "estas a dos cuadras de territorio prohibido"(claro, si no habías pintado nada ahí anteriormente).

Y continúas así hasta haber pintado toda la ciudad o hasta ser atrapado por la policía.

Es claro que el divertido ejemplo anterior solo pretende enseñar la idea de la búsqueda en amplitud y no se espera que el lector realize tales actos.

La idea de la búsqueda en amplitud es visitar un nodo inicial v, y luego visitar los nodos que se encuentren a una arista de v, después los que se encuentren a 2 aristas de v y asi sucesivamente.

De esa forma, la búsqueda en amplitud visita todos los nodos que pueden ser alcanzados desde v, y para cada uno de ellos, calcula su distancia(mínimo número de aristas) con v.

En la situación que imaginamos del bándalo pintando paredes, el bándalo tenía que caminar mucho despues de pintar cada esquina, sin embargo, como nosotros usaremos una computadora para realizar un proceso similar, no necesitamos estar caminando, o recorriendo todo el grafo en busca del siguiente vertice que hay que marcar.

Una forma mas rápida de obtener los mismos resultados es teniendo una cola, y metiendo a v en la cola, luego. mientras la cola no este vacía, sacar un vertice de la cola, visitarlo, y meter a los vecinos no visitados a la cola.

Una manera de hacerlo(asumiendo que la cola nunca se llenará) es la siguiente:

```
1           cola[fin++]=v; //Meter v a la cola
2           visitado[v]=1; //Inicializar
3           distancia[v]=0;
4           while(inicio!=fin){ //Mientras la cola no este
                vacía
5                   a=cola[inicio++]; //Sacar de la cola
6                   for(i=1;i<=N;i++) //Meter vecinos a la
                        cola
7                           if(g[a][i] && visitado[i]==0){
8                                   visitado[i]=1;
9                                   distancia[i]=distancia[
                                        a]+1;
10                                  cola[fin++]=i;
11                          }
12          }
```

El código anterior usa una matriz de adyacencia g, con N nodos.

Como nodo se visita una vez y cada arista se visita 1 o 2 veces dependiendo de si el grafo es dirigido o no dirigido, este algoritmo funciona en tiempo $O(|V|^2)$ con matriz de adyacencia y con listas de adyacencia funciona en tiempo $O(|V| + |A|)$.

15.5. Conectividad

Al inicio de la sección anterior se definió que era un grafo conexo y se vió como recorrer un grafo conexo. Sin embargo no hemos analizado muy a fondo todo lo que esto implica.

Definición 15.5.1 (Conectividad)**.** Decimos que dos vértices v, w estan conectados si existe un camino que los una. Y lo denotaremos como $v \rightsquigarrow w$

Por facilidad vamos a pensar que todo vértice está conectado consigo mismo, hay algunos problemas donde no es prudente pensar así, pero en general es mucho mas manejable la conectividad si pensamos de esta manera. Esta primera propiedad la llamaremos **reflexiva**.

Hay que notar que para todo par de vértices v y w tales que $v \rightsquigarrow w$ también $w \rightsquigarrow v$; eso es fácil de ver ya que solamente hay que recorrer el camino que une a v y w al revés para encontrar el camino que une a w con v. Esta propiedad la llamaremos **simétrica**.

Otra propiedad interesante de la conectividad es que para 3 vértices u, v y w, si $u \rightsquigarrow v$ y $v \rightsquigarrow w$ entonces $u \rightsquigarrow w$. Esta propiedad la llamaremos **transitiva**.

Una vez observadas estas 3 propiedades vamos a definir algo que llamaremos compontente conexa.

Definición 15.5.2 (Componente Conexa)**.** Sea $G = (V, A)$ un grafo y $v \in V$, la componente conexa de v es el conjunto de todos los vértices $u \in V$ tales que $v \rightsquigarrow u$ y se denota como $[v]$.

Esta definición tiene una íntima relación con las 3 propiedades que acabamos de enunciar. Como veremos a continuación con este teorema:

Teorema 25. *Sea $G = (V, A)$ un grafo y sean $u, v \in V$ dos vértices cualesquiera. Tenemos que $u \rightsquigarrow v$ se cumple sí y solo sí $[u] = [v]$.*

Demostración. Supongamos que $u \rightsquigarrow v$, sea $x \in [u]$ y sea $y \in [v]$.

Sabemos por definición que $v \rightsquigarrow y$ y que $u \rightsquigarrow x$.

Por la propiedad transitiva $u \rightsquigarrow y$, por lo tanto $y \in [u]$. Por la propiedad simétrica $v \rightsquigarrow u$ y por la transitiva $v \rightsquigarrow x$ por lo tanto $x \in [v]$.

Es decir todo elemento de $[u]$ también está en $[v]$ y todo elemento de $[v]$ también está en $[u]$ por lo tanto $[u] = [v]$.

Ahora nos vamos a olvidar de que u y v están conectados y vamos a suponer solamente que $[u] = [v]$.

Por la propiedad reflexiva $u \in [u]$ pero como $[u] = [v]$ entonces $u \in [v]$ es decir $v \rightsquigarrow u$ y por la propiedad simétrica $u \rightsquigarrow v$.

□

Teorema 26. *Sea $G = (V, A)$ y sean $[u], [v]$ dos componentes conexas, tenemos que se cumple una de estas dos cosas:*

- $[u] = [v]$ *o bien*
- $[u] \cap [v] = \emptyset$

Demostración. Sean $u, v \in V$ vértices tales que $u \in [u]$ y $v \in [v]$, tenemos que si $u \rightsquigarrow v$ entonces $[u] = [v]$. En caso contrario, para todo $x \in [v]$ tenemos que u no está conectado con x dado que $x \rightsquigarrow v$, y análogamente para todo $y \in [u]$ tenemos que v no está conectado con y. Es decir, $[u]$ y $[v]$ no tienen elementos en común. □

Una vez vistos estos dos teoremas podemos apreciar como el problema de determinar la conectividad entre todos los pares de vértices se reduce a particionar el grafo en varias componentes conexas.

Una vez que se tiene las componentes conexas, para saber si dos vértices estan conectados solamente hay que ver si pertenecen a la misma componente conexa.

Además, para encontrar todas las componentes conexas lo único que hay que hacer es recorrer cada una de ellas una sola vez. ¡Eso es $O(|V| + |A|)$!.

En general esta forma de particionar un conjunto con respecto a una relación tiene el nombre de **clases de equivalencia** y se puede hacer con cualquier relación que sea reflexiva, simétrica y transitiva (por ejemplo, la relación del paralelismo entre rectas en el plano).

Observa nuevamente las demostraciones de los dos teoremas y te darás cuenta que solamente se usaron las 3 propiedades enunciadas y no se usó en absoluto el hecho de que se estuviera hablando de un grafo.

15.6. Árboles

Ya anteriormente se mencionaron los árboles binarios como estructura de datos, sin embargo, la definición que se dió se aleja mucho de lo que *realmente* es un árbol.

Antes de poder entender los árboles necesitamos definir los ciclos, un ciclo es un camino que inicia y termina en el mismo vértice, es decir:

Definición 15.6.1 (Ciclo)**.** Un ciclo es un camino $C = (v_1, v_2, v_3, ..., v_n)$ donde $v_1 = v_n$.

La idea empírica de los árboles se basa en tener un grafo compuesto por *ramificaciones*, y en la cual cada *ramificación* se puede dividirse en otras

ramificaciones y así sucesivamente, tal como si se tuviera un tronco del cual se desprenden ramas, y luego cada rama puede subdividirse a la vez en otras ramas.

Esta idea de los árboles se formaliza con la siguiente definición:

Definición 15.6.2 (Árbol)**.** Se dice que un grafo $G = (V, A)$ es un árbol si cumple con las siguientes dos condiciones:

- Es conexo
- No contiene ciclos

Los árboles además aparecen de manera natural en muchas situaciones, como se demuestra con el siguiente par de teoremas:

Teorema 27. *Sea $G = (V, A)$ un grafo conexo, las siguientes tres proposiciones son equivalentes:*

- *G es un árbol.*
- *Entre cada par de vértices existe un solo camino que no repite vértices que los une.*
- $|A| = |V| - 1$

Demostración. Primero supongamos que G es un árbol, al ser un árbol(y por tanto conexo) sabemos que entre cada par de vértices v y w existe al menos un camino que los une, vamos a denotarlo como $A = (a_1, ..., a_n)$ donde $a_1 = v$ y $a_n = w$.

Ahora, para probar que este camino es único vamos a suponer que existe otro, el cual denotaremos como $B = (b_1, ..., b_m)$ con $b_1 = v$ y $b_m = w$.

Como A y B no son el mismo camino entonces existe al menos un entero k tal que $a_k \neq b_k$, también hay que notar que $a_1 = b_1$ y $a_n = b_m$.

A lo que se intenta llegar con esto es que los dos caminos tienen un *prefijo común*, el cual sea tal vez de un solo nodo, y luego se vuelven a juntar en otro nodo.

Dicho de otra manera podemos hablar del mayor entero i tal que $a_1 = b_1, a_2 = b_2, ...a_i = b_i$, en este caso decimos que $(a_1, ..., a_i) = (b_1, ..., b_i)$ es el prefijo que tienen en común A y B.

Tomando en cuenta que $a_n = b_m$ podemos estar seguros que al menos un elemento de $\{a_{i+1}, a_n\}$ es igual a un elemento de $\{b_{i+1}, b_m\}$, ahora, de entre todos esos elementos, vamos a elegir aquel que aparezca antes en A, y vamos a denominarlo a_x, además, como por definición aparece en ambos, vamos a elegir alguna y tal que $a_x = b_y$.

Consideramos ahora el camino $(a_i, a_{i+1}, ..., a_x, b_{y-1}, b_{y-2}, ..., b_i)$ este camino es un ciclo, por lo tanto un árbol no puede tener mas de un camino uniendo al mismo par de vértices. Es decir, si G es un árbol, entre cualquier par de vértices existe un solo camino.

Para seguir probano que las tres proposiciones son equivalentes, ahora nos vamosa olvidar de que $G = (V, A)$ es un árbol y solo vamos a suponer que entre cada par de nodos existe un único camino, e intentaremos demostrar que $|A| = |V| - 1$.

Usaremos inducción, es tentador usar como caso base un grafo con un solo vértice(en el cual es evidente que se cumple); sin embargo, resulta mas cómodo usar como caso base un grafo con dos vértices.

Si solamente hay dos vértices, entonces solo puede haber una arista(existen estructuras llamadas multigrafos que soportan varias aristas entre los mismos dos vértices, pero no estamos hablando de ellas), y entre ese único par de vértices hay un solo camino.

Ahora, si se tiene un grafo con n vértices(para $n \geq 2$), $n - 1$ aristas, y entre cada par de vértices hay un solo camino, entonces, si se añade una nueva arista entre dos vértices v y w, entonces habría dos caminos que conectan a v con w, uno sería el que ya se tenía antes, y el otro sería un camino de una sola arista (v, w).

Aquí acabamos de demostrar una propiedad interesante, y es que si se añade una arista a un grafo donde entre cada par de vértices hay un solo camino, entonces, luego de ello, existirá al menos un par de vértices conectados por mas de un camino.

Continuado con la inducción, si le añadimos un vértice al grafo, llamémosle v' y lo conectamos con otro vértice w, tenemos que para cualquier otro vértice $u \in V$ existe un único camino que une a u con w, llamémosle C, y por tanto existe un único camino entre u y v, el cual consiste de C seguido de v.

Por lo tanto, por inducción, si entre cada par de vértices existe un solo camino que los une, entonces $|A| = |V| - 1$.

El tercer paso de nuestra prueba es demostrar que si un grafo conexo $G = (V, A)$ tiene exactamente $|V| - 1|$ aristas, entonces el grafo es un árbol.

Para demostrar eso vamos a recurrir a un concepto parecido al de árbol y es el de bosque, un bosque es un grafo sin ciclos, el cual puede o no ser conexo. Como se puede ver, un árbol es un bosque pero un bosque no necesariamente es un árbol.

Un grafo con n vértices y ningún arista es un bosque y tiene n componentes conexas, además es el único grafo de n vértices que tiene exactamente n componentes conexas. ¡Esto es un buen caso base para inducción!

Supongamos que para un $k > 0$ todo grafo con n vértices, k aristas y

$n-k$ componentes conexas es un bosque. Si a cualquiera de esos grafos le añadiéramos una arista entre dos vértices de la misma componente conexa, obtendríamos un grafo con n vértices, $k+1$ aristas y $n-k$ componentes conexas.

Sin embargo, si añadimos una arista entre dos vértices de dos componentes conexas distintas obtenemos un grafo con n vértices, $k+1$ aristas y $n-k-1$ componentes conexas. Este grafo será un bosque, puesto que conectar dos componentes conexas distintas con una sola arista nunca crea un ciclo. Es decir, todo grafo con n vértices, $k+1$ aristas y $n-k-1$ componentes conexas será un bosque.

Por inducción un grafo con n vértices, $n-1$ aristas será un bosque con $n-(n-1)=1$ componentes conexas, es decir, un árbol.

Observamos que la primera proposición implica la segunda, la segunda implica la tercera y la tercera implica la primera, de esta manera queda demostrado que las tres proposiciones son equivalentes.

□

Un elemento muy interesante que poseen los árboles son las *hojas*, en los árboles binarios las hojas son aquellos vértices que no tienen hijos, aquí son algo parecido, son aquellos vértices que estan unidos solamente con vértice, podemos pensar en ese vértice como si fuera el *padre* de la hoja.

Por lo tanto nuestra definición de hoja queda de la siguiente manera:

Definición 15.6.3 (Hoja)**.** Una hoja es un vértice con grado 1.

Las hojas son un elemento interesante ya que son el punto de partida para las soluciones de algunos problemas y además siempre están presentes en los árboles, o mas objetivamente:

Teorema 28. *Sea $G=(V,A)$ un árbol, si $|V|\geq 2$ entonces G tiene al menos 2 hojas.*

Demostración. Es evidente que no hay vértices con grado 0, puesto que el grafo no sería conexo y por tanto no sería un árbol.

Como la suma de los grados de un grafo $2|A|$ entonces la suma de los grados de G es $2|V|-2$, si todos los vértices tuvieran grado mayor o igual a 2, entonces la suma de los grados sería al menos $2|V|$, pero como esto no es cierto, entonces existe un nodo de grado 1.

Si un solo nodo tuviera grado 1 y el resto de los nodos tuviera al menos grado 2, entonces la suma de los grados sería al menos $2|V|-1$, como la suma es $2|V|-2$ entonces al menos 2 nodos tienen grado 1, es decir, el árbol tiene al menos 2 hojas.

□

Ya vimos que todo grafo $G = (V, A)$ con $|V| - 1$ aristas es un árbol, si pensamos en un grafo conexo que tenga tantos vértices como aristas inmediatamente se puede imaginar a un árbol con una arista adicional. Lo cual hace pensar que es un grafo con un solo ciclo.

En este caso el sentido común no nos traiciona, ya que realmente el grafo es un árbol con una arista adicional que posee un solo ciclo, sin embargo, aún debemos de demostrarlo, pues debemos de considerar la posibilidad de que haya mas de un ciclo o de que el grafo deje de ser conexo si se quita alguna arista.

Teorema 29. *Sea $G = (V, A)$ un grafo conexo tal que $|V| = |A|$ entonces G contiene un solo ciclo.*

Demostración. Si G no contuviera ciclos entonces sería un árbol y eso implicaría $|A| = |V| - 1$, lo cual es falso.

Ya demostramos que hay al menos un ciclo, ahora demostraremos que hay un solo ciclo, sea $C = (v_1, ..., v_n)$ un ciclo en G, tenemos que para v_1, v_2 existen al menos dos caminos que los unen por estar en un mismo ciclo.

Pero ya sabemos que $\{v_1, v_2\} \in A$ por tanto (v_1, v_2) es un camino que une a v_1 con v_2, ahora veremos que sucede si le *quitamos* la arista al grafo, es decir, consideraremos el grafo $H = (V, A - \{v_1, v_2\})$.

Tenemos que H es conexo, ya que aún existe un camino que une a v_1 con v_2, ya que para cada par de vértices $u, w \in V$ existe un camino que los une en G, si la arista $\{v_1, v_2\}$ no está presente en el camino, entonces dicho camino también existe en H, y si está presente, entonces es posible reemplazarla por este camino $(v_2, ..., v_n, v_1)$.

Como el número de aristas en H es $|A| - 1$ entonces H es un árbol y no tiene ciclos. Es decir, todos los ciclos en G tienen la arista $\{v_1, v_2\}$.

Ahora, tomemos dos ciclos cualesquiera en H, si *quitamos* a $\{v_1, v_2\}$, luego de eso habrá un único camino entre v_1 y v_2, el cual es $(v_2, ..., v_n, v_1)$, y ambos ciclos unen a v_1 y a v_2 con 2 caminos incluyendo el camino de una sola arista, por lo tanto ambos ciclos pasan por ese camino y por la arista $\{v_1, v_2\}$, es decir, son el mismo ciclo. □

Los árboles estan íntimamente ligados a la conectividad, como se ve en la demostración del teorema 27. A continuación observaremos algunas propiedades de los árboles que los convierten en un objeto importante de estudio dentro de la conectividad.

Como ya habíamos visto, un árbol se define como un grafo conexo sin ciclos, el hecho de que no tenga ciclos hace suponer que tiene pocas aristas, por ello, para encontrar un grafo conexo con pocas aristas, suena razonable buscar un árbol.

La propiedad que estudiaremos ahora nos dice que no solo es razonable buscar un árbol, sino que es lo mejor. Es decir, no hay un grafo conexo que tenga menos aristas que un árbol, dicha propiedad se resume en el siguiente teorema:

Teorema 30. *Sea $G = (V, A)$ un grafo tal que $|A| < |V| - 1$ entonces, el grafo no es conexo.*

Demostración. Tomemos un grafo $H_0 = (V, \emptyset)$, es decir un grafo con los mismos vértices que V pero sin aristas, H_0 tiene exactamente $|V|$ componentes conexas.

Vamos a definir H_n como un subgrafo de G con exactamente n aristas.

Supongamos que para cualquier entero k, H_k tiene al menos $|V| - k$ componentes conexas. Si se añade una arista entre 2 vértices v, w, si $[v] \neq [w]$ el número de componentes conexas disminuye en 1, y si $[v] = [w]$ el número de componentes conexas se mantiene, por lo cual H_{k+1} tendrá al menos $|V| - (k + 1)$ componentes conexas.

Por inducción H_n tiene al menos $|V| - n$ componentes conexas. Notamos que $G = H_{|A|}$, por lo tanto, el grafo tiene al menos $|V| - |A|$ componentes conexas, pero $|V| - |A| > 1$, por lo tanto G tiene mas de una componente conexa, es decir, no es conexo. □

A veces lo que se necesita hacer es encontrar un grafo conexo, y ya vimos que si se quieren tener pocas aristas lo mejor es tener un árbol. Sin embargo, si se quiere trabajar con un subgrafo conexo de un grafo conexo, hasta el momento nada nos dice qué subgrafo tomar si queremos simplificar las cosas usando pocas aristas.

El lector puede estar pensando en tomar un árbol, este árbol es llamado árbol de expansión, ya vimos en la demostración del teorema 29 que este arbol siempre existe con un grafo conexo que tiene tantos vértices como aristas, veamos un caso mas general:

Teorema 31. *Sea $G = (V, A)$ un grafo conexo, entonces existe un árbol $T = (V, A')$ tal que $A' \subseteq A$, el cual es llamado árbol de expansión de G.*

Demostración. Como G es conexo $|A| \geq |V| - 1$.

Si $|A| = |V| - 1$ entonces G es un árbol, en este caso existe un árbol de expansión de G el cual es G.

Ahora, si $|A| > |V| - 1$ entonces G al ser conexo y no ser un árbol tiene al menos un ciclo, si quitamos cualquier arista del ciclo, el grafo seguirá siendo conexo, si se repite esta operación $|A| - (|V| - 1)$ veces, el grafo seguirá siendo conexo y será un árbol.

Dicho árbol está contenido en el grafo original, y es el mencionado árbol de expansión. □

15.7. Unión y Pertenencia

Ya vimos en la sección anterior como los árboles se relacionan de una manera natural con la conectividad, ahora vamos a aprovechar todo esto para resolver el problema de unión-pertenencia.

Este problema es un problema de diseño de estructura de datos, es decir, ocupamos mantener un conjunto de datos y estarlo continuamente actualizando. En este problema hay que diseñar una estructura de datos con la cual podamos representar estas dos operaciones en un grafo:

- **Unión**. Añadir una arista entre dos vértices.
- **Pertenencia**. Saber si un par de vértices estan conectados.

Pueden sonar extraños los nombres de las operaciones. Se llaman así porque se puede pensar en cada componente conexa como un conjunto. Añadir una arista entre dos vértices es equivalente a unir dos conjuntos. Saber si dos vértices se encuentran en la misma componente conexa se puede interpretar como saber que dos vértices se encuentran en el mismo conjunto.

A pesar de esta notación sugestiva, aquí seguiremos tratando este problema como un problema de conectividad.

Hasta el momento, con lo que hemos visto la única manera de atacar este problema sería realizando múltiples búsquedas.

Podríamos realizar la unión añadiendo una arista en tiempo constante si representamos el grafo con listas de adyacencia, pero si usamos una búsqueda para la operación de pertenencia nos tomaría tiempo lineal; y en este problema hay que estar preparados para poder ejecutar muchas veces ambas operaciones.

Otra cosa que podríamos hacer es numerar las componentes conexas y mantener un arreglo diciendo a qué componente pertenece cada uno de los vértices. Con esto podríamos lograr la pertenencia en tiempo constante, pero la unión en el peor de los casos seguiría funcionando en tiempo lineal ya que habría que reasignar la componente conexa a varios vertices.

En estos momentos parece claro que si seguimos pensando en optimizaciones menores para las búsquedas no llegaremos a mucho, necesitamos profundizar mas al respecto.

Si recordamos la sección anterior, todo grafo conexo tiene un árbol de expansión, por lo tanto, toda componente conexa también tiene un árbol de expansión. Hay que observar que si dos árboles son unidos por una arista, entonces el resultado será otro árbol sin importar qué vertice se elija en cada árbol.

Tambien hay que recordar que solo nos interesan las operaciones de unión y pertenencia. Con todo esto ya podemos estar seguros que no es necesario guardar todas las aristas, es suficiente con guardar un árbol de expansión de cada componente conexa.

Pero esta propiedad no basta para diseñar una estructura de datos eficiente. Recordemos que en un árbol entre todo par de vértices hay un único camino que los une(claro que descartando los caminos que repiten vértices).

Consideremos un árbol $T = (V, A)$, vamos a elegir arbitrariamente un vértice r, al cual llamaremos raíz. Está claro que para cualquier vértice $v \in V$ existe un único camino que lo une con r, vamos a llamar $P[r]$ al segundo vértice de dicho camino(el primer vértice del camino claramente es v), si $v = r$ entonces el camino consta solamente de un vértice y por conveniencia diremos que $P[r] = r$.

Así que para llegar v a la raiz simplemente hay que seguir este algoritmo recursivo:

```
1 int encuentra_raiz(int v){
2         if(P[v]==v){
3                 return v;
4         }else{
5                 return encuentra_raiz(P[v]);
6         }
7 }
```

Volviendo al problema original, si en lugar de guardar todo el grafo simplemente guardamos un árbol de expansión del grafo original entonces podemos a cada componente conexa asignarle una *raíz*, si hiciéramos esto ya no necesitaríamos usar listas de adyacencia ni búsquedas, simplemente guardaríamos para cada vértice v el valor de $P[v]$, y de esta manera desde cualquier vértice podríamos llegar a la raíz de su componente conexa.

Si dos vértices estan conectados, entonces estan en la misma componente conexa. Si estan en la misma componente conexa entonces estan conectados con la misma raíz. Análogamente si estan conectados con la misma raíz entonces están en la misma componente conexa y por tanto estan conectados.

Bajo este esquema debemos de notar también que solamente necesitamos tener un árbol para cada componente conexa y cada árbol debe de contener a los vértices de su respectiva componente. Pero no nos importa la estructura del árbol, solamente qué vertices tiene, por lo tanto no es necesario que sea un árbol de expansión.

Si queremos unir dos componentes conexas A y B, basta con hacer que la raíz de A apunte a la raíz de B(o viceversa), de esa manera todos los nodos A estarán conectados con la misma raíz que todos los nodos de B.

Nuestro procedimiento de unión queda de la siguiente manera:

```
1  void union(int a, int b){
2          int ra, rb;
3          ra=encuentra_raiz(a);
4          rb=encuentra_raiz(b);
5          P[ra]=rb;
6  }
```

Y como dos vértices estan conectados sí y solo sí estan conectados con la misma raíz entonces nuestro procedimiento de pertenencia queda de la siguiente manera:

```
1  bool pertenencia(int a, int b){
2          if(encuentra_raiz(a)==encuentra_raiz(b)){
3                  return true;
4          }else{
5                  return false;
6          }
7  }
```

No hay que olvidarse de que también el grafo necesita ser inicializado. Para este problema normalmente se considera que al inicio el grafo contiene solo vértices y no tiene aristas, por lo cual debe de haber $|V|$ componentes conexas y cada vértice es la raíz de su propia componente conexa. El siguiente código inicializa un grafo con n vértices, como ya lo debes imaginar, este código debe de ser llamado antes de usar unión o pertenencia.

```
1  void inicializa(int n){
2          int i;
3          for(i=1;i<=n;i++){
4                  P[i]=i;
5          }
6  }
```

Es claro que este algoritmo es bastante superior a hacer búsquedas, sin embargo, aún no lo analizamos adecuadamente.

15.8. Mejorando el Rendimiento de Unión-Pertenencia

Cómo recordará el lector, el algoritmo para resolver el problema de unión-pertenencia se basa en tener varios árboles con raíces asignadas, donde cada vértice apunta a su *padre*, y para unir dos árboles hacemos que la raíz de uno apunte a la raíz del otro.

Ya debemos de acordarnos de los árboles binarios de búsqueda y cómo a veces los árboles que se crean pueden tener alturas terriblemente grandes, de manera que son similares a listas enlazadas.

En estos árboles tambien puede pasar lo mismo. Vamos a definir la altura de un árbol de la siguiente manera:

Definición 15.8.1 (Altura)**.** La altura de un árbol $T = (V, A)$ con raíz $r \in V$, es la distancia entre v y r donde $v \in V$ es el vértice mas lejano a r.

Si dos árboles de alturas h_1 y h_2 se unen con el procedimiento de la sección anterior, el árbol resultante puede tener altura igual a $h_1 + 1$ o bien altura igual a h_2. Lo primero sucedería en el caso de que $h_1 \geq h_2$ y lo segundo sucedería en el caso de que $h_1 < h_2$.

Esto es una fuerte sugerencia de que al realizar uniones entre árboles es mejor hacer que la raíz de un árbol con altura mínima apunte al otro árbol, si hacemos esto para obtener un árbol de altura 1 se necesitarían 2 de altura 0, para obtener uno de altura 2 se necesitarían 2 de altura 1, para obtener un árbol de altura 3 se necesitarían 2 de altura 2, o de manera mas general: para obtener un árbol de altura $n > 0$ serían necesarios al menos dos árboles de altura $n - 1$.

Si definimos $h(n)$ como el número mínimo de vértices que se pueden usar para construir un árbol de altura n, tenemos que $h(0) = 1$, y que $h(n) = 2h(n-1)$, por lo tanto si tenemos 2^n vértices el árbol mas alto que se puede construir tendría altura n.

Por lo tanto, de esta manera tanto la unión como la pertenencia tendrían un tiempo de ejecución de $O(log\ N)$, donde N es el número de vértices. Esto es bastante razonable.

Sin embargo todo esto se puede simplificar aún más si cada vez que se encuentra una raíz se borran todas las aristas que se recorrieron y se colocan aristas que apunten directamente a la raíz. Si hacemos esto, cada arista sería recorrida a lo más una vez, salvo que apunte directamente a la raíz.

Implementar esto es batante sencillo, basta con modificar el procedimiento para encontrar raíz de la siguiente manera:

```
1  int encuentra_raiz(int v){
2          if(P[v]==v){
3                  return v;
4          }else{
5                  P[v]=encuentra_raiz(P[v]);
6                  return P[v];
7          }
8  }
```

Como cada arista que no apunta directamente a la raíz se recorre a lo más una vez y todas las aristas son creadas mediante una operación de unión, entonces el algoritmo toma un tiempo de $O(u + p)$ donde u es el número de llamadas a la operación unión, y p es el número de llamadas a la operación pertenencia.

Esto no quiere decir que el procedimiento unión y el procedimiento pertenencia tomen tiempo constante, ya que el tiempo de ejecución del algoritmo no se distribuye uniformemente entre todas las llamadas a estos procedimientos.

15.9. Problemas

15.9.1. Una Ciudad Unida

Tiempo Límite: 1 segundo

Unos ambiciosos ingenieros tienen pensado construir una gran ciudad a partir de unos planos que elaboraron.
El plano de la ciudad consta de N esquinas y M calles bidireccionales.
Cada calle une a lo mas a dos equinas.
Se dice que existe un camino que une a dos esquinas v y w, si $v = w$, si hay una calle conectando directamente a las dos esquinas, o si existe una secuencia de esquinas $a_1, a_2, a_3, ..., a_k$, tal que $a_1 = v$, $a_k = w$, y toda a_i(menos a_k) está unida directamente con a_{i+1} por una calle.
Los ingenieros se preguntan si habrán planificado mal algo, como por ejemplo, que si una persona vive en una esquina A y quiere ir a visitar a un amigo que vive en una esquina B, no exista un camino que le permita llegar a la esquina B.

Problema

Debes hacer un programa que dado un plano de la ciudad, determine cuantos pares de esquinas hay tales que no exista un camino que las una.

Entrada

Descripción

Línea 1: 2 números enteros N y M separados por un espacio.
Siguientes M líneas: Cada línea representa una calle y contiene 2 números enteros representando los números de las esquinas que une la calle.

Ejemplo

```
5 3
1 2
2 4
3 5
```

Salida

Descripción

Línea 1: Un solo número entero, el número de pares de esquinas para las cuales no existen

Ejemplo

6

Consideraciones

0<2000<N
0<100000<M

Referencias

El problema fue escrito por Luis Enrique Vargas Azcona y fué utilizado por primera vez en un examen preselectivo de la Olimpiada de Informática durante el 2007.

15.9.2. Abba

Tiempo Límite: 1 segundo

Descripción

Dada una cadena de caracteres `S`, la operación `reemplaza(a, b)` cambia cada una de las ocurrencias del caracter `a` por el caracter `b`. Por ejemplo, si $S = abracadabra$ entonces `reemplaza(b, c)` produce la cadena `acracadacra`.

Un palíndrome es una cadena de caracteres que se lee de la misma forma de izquierda a derecha que de derecha a izquierda. Por ejemplo, `abba` y `dad` son palíndromes.

Problema

Escribe un programa que lea una cadena de caracteres S y que encuentre el número mínimo r de aplicaciones de la operación `reemplaza` que transforman a S en un palíndrome.

Entrada

Un solo renglón que contiene una cadena S formada exclusivamente por letras minúsculas del alfabeto inglés.

Ejemplo

```
croacia
```

Salida

El valor de r.

Ejemplo

```
3
```

Consideraciones

La cadena S tendrá una longitud entre 1 y 1,000,000.

Referencias

La idea original del problema fue de Sergio Murguía, el problema fué redactado por Francisco Javier Zaragoza Marinez y fue utilizado por primera vez en un examen selectivo de la Olimpiada Mexicana de Informática durante Julio del 2007.

15.9.3. Códigos de Prüfer

Tiempo Límite:1 Segundo

El matemático Heinz Prüfer inventó una forma ingeniosa de nombrar árboles cuyos vertices esten etiquetados de manera única con los números enteros de 1 a n

La forma de nombrar el árbol consiste de un código con $n-2$ números llamado código de Prüfer. Dicho código se obtiene mediante el siguiente algoritmo:

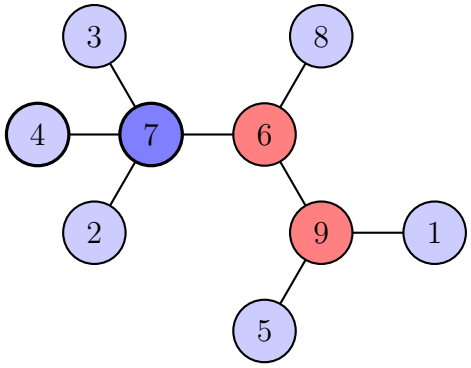
1. Localiza la hoja h que esté etiquetada con el número mas pequeño.

2. Agrega al código la etiqueta del vértice al cual está unido h

3. Borra h

4. Si quedan 3 vértices o más, regresa al paso 1

Por ejemplo, el código de Prüfer asociado al árbol de la siguiente figura es 9 7 7 7 9 6 6



Problema

Dado un código de Prüfer encuentra el árbol que genera ese código.

Entrada

Línea 1: Un entero n
Línea 2: $n-2$ enteros separados por espacios representando el código de Prüfer.

Ejemplo de Entrada

```
9
9 7 7 7 9 6 6
```

Salida

La salida deberá constar de N líneas, donde para cada $1 \leq i \leq n$, la $i-$ésima línea deberá indicar los números de los vértices unidos al vértice i en orden ascendente y separados por espacios.

Si hay múltiples soluciones simplemente imprime una sola línea con la palabra `AMBIGUO`.

Si no hay solución imprime una sola línea con la palabra `IMPOSIBLE`

Ejemplo de Salida

```
9
7
7
7
9
7 9
2 3 4 6
7 8 9
```

Límites

$2 \leq n \leq 100000$

Referencias

Este es un problema clásico ingeniado por el mismo Prüfer para demostrar que el número de árboles con exactamente n nodos es n^{n-2}.

El problema fue redactado por Luis Enrique Vargas Azcona en Junio de 2009 para entrenar a la preselección nacional en la Olimpiada Mexicana de Informática.

15.10. Sugerencias

Una Ciudad Unida

Considera hayar las componentes conexas en lugar de hacer una búsqueda independiente con cada vértice.

Abba

Piensa en un grafo $G = (V, A)$ donde V consta de los 26 caracteres del alfabeto inglés y $\{v, w\} \in A$ sí y solo sí en algún momento hay que cambiar alguna letra v ó alguna letra w(nótese que al decir letra v y letra w esta sugerencia se refiere a cualquier letra en V, no a los caracteres "v" y "w").

¿Qué sucede con ese grafo si se reemplazan todas las coincidencias de una letra por otra?

C+ódigos de Prüfer

¿Qué sucede con las hojas de un árbol en su código de Prüfer correspondiente?. ¿Realmente es posible que la salida sea `AMBIGÜO` o `IMPOSIBLE`?

Parte V

Optimización Combinatoria

Capítulo 16

Estructura de la Solución y Espacio de Búsqueda

Gran parte de los problemas tratados en la algoritmia son de optimización combinatoria; es decir, dentro de un conjunto finito, elegir el mejor elemento o encontrar aquellos elementos que cumplan con una característica especial.

Esto suena como si fuera algo realmente sencillo, pues todo lo que hay que hacer es evaluar uno por uno y quedarse con el mejor. Lo interesante de estos problemas radica en encontrar el mejor mas rapidamente o en reducir el número de posibles soluciones.

Al procedimiento de evaluar *candidato* que pudiera ser una solución le llamamos búsqueda(refiriéndonos a que se está *buscando* una solución), y al conjunto dentro del cual tenemos que encontrar la solución le llamaremos espacio de búsqueda.

Aquí hay que aclarar que no se está abusando de la palabra *búsqueda*, la cual es usada para referirse a los recorridos en grafos, ya que un grafo es en escencia un espacio de búsqueda.

Por lo general la definición de espacio de búsqueda es simplemente el conjunto de todos los posibles *candidatos* a ser una solución de un problema de optimización. Sin embargo, para nuestros propósitos usaremos una definición algo diferente, la cual requiere de un poco mas de texto para entender tanto su significado como su motivación.

Por el momento nos quedaremos con la siguiente definición provicional:

Definición 16.0.1 (Espacio de Búsqueda(definición temporal))**.** Un conjunto que contiene a todos los *candidatos* para ser una solución de un problema de optimización.

Es decir es un conjunto al cual pertenece la solución del problema, pudiendo tener otros elementos que no sean la solución.

Lo primero que se necesita para hacer uso eficaz de una búsqueda es definir un espacio de búsqueda, este paso es obvio, antes de buscar, requerimos saber en donde buscar. Pero aún cuando parezca obvio, es el paso más difícil de todos y el que menos programadores son capaces de tomar.

Como buen programador es indispensable que puedas definir un espacio de búsqueda para cualquier problema que se te presente, de otra forma tendrás serios problemas para poder encontrar soluciones a problemas que no sean triviales.

Para poder definir el espacio de búsqueda, es necesario conocer la estructura de la solución, es decir, ¿Qué es la solución que me piden? ¿Qué estructura tiene? ¿Cómo se puede representar?

16.1. Estructura de la Solución

Normalmente la estructura de la solución de un problema de optimización combinatoria se puede simplificar a una sucesión de decisiones, cada decición representada mediante un número.

Por ejemplo en el siguiente problema:

Ejemplo 16.1.1. En una tienda venden paquetes de crayones, el paquete de 11 crayones cuesta $15, el paquete de 20 cuesta $25, el paquete de 2 cuesta $5.

Se cuenta con exactamente $30, ¿cuál es la mayor cantidad de crayones que se pueden comprar?

Solución Este problema se puede reducir a tomar estas 3 deciciones:

- ¿Cuántos paquetes de 11 crayones comprar?
- ¿Cuántos paquetes de 15 crayones comprar?
- ¿Cuántos paquetes de 2 crayones comprar?

Se pueden comprar a lo mas 2 paquetes de 11, a lo más un paquete de 15 y a lo más 6 paquetes de 2. Nuestra estructura de la solución puede ser una terna ordenada de enteros indicando cuántos paquetes se van a comprar de cada tipo.

Por lo tanto nuestro espacio de búsqueda es el siguiente:

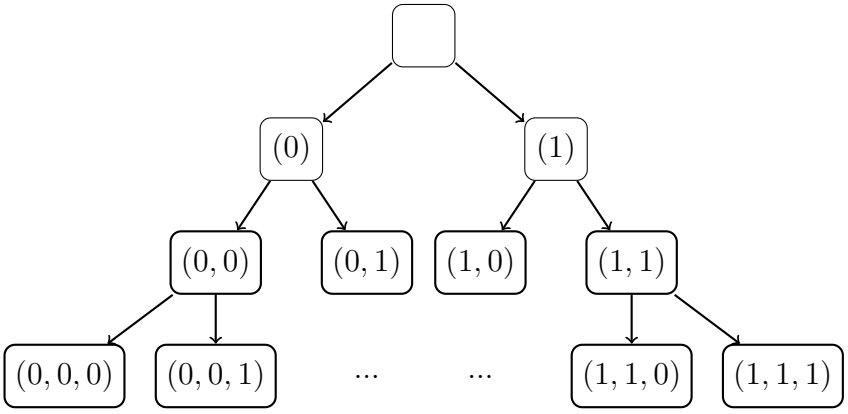


Figura 16.1: Árbol de Decisiones

$$\{(0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(0,0,5),(0,0,6) \quad (16.1)$$
$$(0,1,0),(0,1,1),(0,1,2),(0,1,3),(0,1,4),(0,1,5),(0,1,6) \quad (16.2)$$
$$(1,0,0),(1,0,1),(1,0,2),(1,0,3),(1,0,4),(1,0,5),(1,0,6) \quad (16.3)$$
$$(1,1,0),(1,1,1),(1,1,2),(1,1,3),(1,1,4),(1,1,5),(1,1,6) \quad (16.4)$$
$$(2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4),(2,0,5),(2,0,6) \quad (16.5)$$
$$(2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4),(2,1,5),(2,1,6)\} \quad (16.6)$$

Muchas veces resulta conveniente visualizar esta sucesión de decisiones como árboles enraizados(con raíz), donde cada nodo representa un prefijo de uno o mas *candidatos* a soluciones, como te podrás imaginar un nodo padre es prefijo de toda su descendencia y las hojas representan los elementos del espacio de búsqueda. Estos árboles a menudo son llamados *árboles de decisiones*.

Definición 16.1.1 (Árbol de Decisiones)**.** Si un espacio de búsqueda S posee como elementos solamente sucesiones finitas, entonces S posee un árbol de decisiones.

El árbol de decisiones de S es un árbol enraizado $T = (V, A)$ con raíz r en el cual se cumplen las siguientes condiciones:

- Para todo nodo $v \in V$ existe $p(v)$ tal que $p(v)$ es prefijo de algun elemento de S. Llamaremos $p(v)$ la **subsolución** asociada a v.
- Para todo prefijo x de todo elemento $s \in S$ existe un nodo $v \in V$ tal que $p(v) = x$.

- Si para dos nodos $u, v \in V$, u es padre de v entonces $p(u)$ es prefijo de $p(v)$.

La figura 16.1 muestra el árbol correspondiente al espacio de búsqueda que definimos. Obviamente solo muestra algunos nodos, ya que ocuparía demasiado espacio en la página. Notamos como la raíz está vacía, los hijos directos de la raíz constan de un solo número, los nodos a distancia 2 de la raíz son pares ordenados y las hojas, que estan a distancia 3 de la raíz, son ternas ordenadas.

Aquí debemos de notar que no todas las ternas del espacio de búsqueda son aceptables, puesto que algunas se exceden del presupuesto, por ejemplo, la terna $(2, 1, 6)$ no es aceptable, ya que comprar 2 paquetes de 11 crayones, 1 de 15 y 6 de 2 cuesta $30 + 15 + 30 = 75$ y en este problema solamente se dispone de la cantidad de 30.

El objetivo del espacio de búsqueda no es ser una solución al problema, ni ser un conjunto de soluciones aceptables, de hecho puede haber muchos espacios de búsqueda válidos para un mismo problema.

El objetivo real de un espacio de búsqueda es ser un conjunto finito donde seguramente se encuentre la solución, muchas veces es preferible que este conjunto sea pequeño, sin embargo a veces es mas sencillo trabajar con espacios mas grandes.

Volviendo al problema, si verificamos todas las ternas, una por una, llegaremos a la conclusión de que $(2, 0, 0)$ representa la solución al problema, es decir, la solución es comprar 2 paquetes de 11.

□

El ejemplo anterior fue algo burdo y muy probablemente el lector llegó a la conclusión de los 2 paquetes de 11 sin necesidad de verificar cada terna en el espacio de búsqueda. No hay que subestimar el ejemplo, se incluyó para ilustrar como la estructura de la solución de un problema de optimización combinatoria se puede reducir a tomar una sucesión de decisiones(en este caso 3 decisiones).

El identificar correctamente la estructura de la solución y visualizar el espacio de búsqueda como un árbol enraizado ya es un gran paso, y ahora podemos entender las búsquedas que ya se presentaron anteriormente de la siguiente manera:

- **Búsqueda en Profundidad**. Procedimiento recursivo que trata de llegar siempre lo más profundo que pueda, en cada paso, si aún no ha encontrado la solución, trata de bajar un nivel en el árbol, si no es posible bajar más, entonces regresa un nivel y trata de bajar por la siguiente rama que todavía no haya recorrido.

2	5	3
1	8	4
7	9	6

Figura 16.2: Estado inicial del tablero

- **Búsqueda en Amplitud**. Recorre el árbol nivel por nivel, es decir, en el primer paso busca la solución entre todos los nodos del primer nivel del árbol, si no encuentra la solución, entonces baja un nivel y la busca entre todos los nodos del segundo nivel, y de esa manera recorre cada uno de los niveles hasta encontrar la solución.

Aquí hay que hacer notar que las búsquedas en amplitud son especialmente útiles cuando lo que se desea es la solución que este más cercana a la raíz del árbol, ya que al ir recorriendo nivel por nivel, la primera solución que se encuentra es aquella que esta más cercana a la raíz.

También es agradable notar que no hemos dejado de ver las búsquedas como recorridos de grafos, ya que un árbol enraizado también es un grafo.

Dedicaremos las siguientes secciones completamente a ejemplificar el uso de búsquedas identificando los árboles de decisiones correspondientes así como a notar las deficiencias que tiene este modelo y motivar otra forma mas refinada de ver los espacios de búsqueda.

Es importante leer estas secciones ya que además de ilustrar como usar las búsquedas en problemas no triviales, nos llevarán a nuestra nueva concepción de espacio de búsqueda y en el camino se darán mas definiciones.

16.2. Juego de Números

Esta sección se dedica exclusivamente a un problema, el cual llamaremos "Juego de Números" no sobra decir que este fué uno de los primeros problemas utilizados para entrenar y elegir a la selección mexicana para la Olimpiada Internacional de Informática del 2005.

El contexto del problema es el siguiente:

Hay un juego, que se juega sobre un tablero de NxN casillas, en cada casilla hay un número distinto entre 1 y N^2, por lo que todas las casillas del tablero estan llenas.

El objetivo del juego consiste en llevar la configuración inicial del tablero a una configuración en la que los números del 1 al N^2 se encuentren ordenados comenzando de izquierda a derecha y de arriba a abajo.

1	2	3
4	5	6
7	8	9

Figura 16.3: Estado final del tablero

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

Figura 16.4: Rotación horaria en $(2, 2)$

Es decir, si el tablero inicialmente estuviera como se muestra en la figura 16.2 el objetivo del juego sería llevarlo a la configuración de la figura 16.2.

Para mover números de su lugar te puedes posicionar en cualquier casilla (i, j) tal que $1 \leq i < N$ y $1 \leq j < N$.

Una vez en la casilla (i, j) se puede hacer una rotacion, ya sea en sentido horario o antihorario utilizando las casillas $(i+1, j)$, $(i+1, j+1)$ y $(i, j+1)$.

Por ejemplo, para el de la figura 16.2 si nos posicionaramos en la casilla $(2, 2)$ podemos hacer una rotacion(horario o antihoraria) con las casillas $(3, 2)$, $(2, 3)$ y $(3, 3)$. Las figuras 16.2 y 16.2 muestran ambas rotaciones.

Ejemplo 16.2.1. Escribe un programa que dada la configuracion inicial de un tablero de 3 x 3 determine cual es el numero minimo de movimientos que se requieren para ordenarlo. Se considera un movimiento cualquier giro que se haga, ya sea en sentido horario o antihorario.

El programa deberá de funcionar en menos de un segundo.

Estructura de la Solución

Primero que nada identificaremos la estructura de la solución, y para eso debemos de responder esta pregunta ¿qué nos está pidiendo el problema?

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

Figura 16.5: Rotación antihoraria en $(2, 2)$

La respuesta es sencilla: Nos está solicitando el número mínimo de movimientos para llegar de una permutación inicial de las casillas a una permutación final.

Es conveniente ver el estado del tablero como una permutación de 9 números enteros, donde los primeros 3 corresponden a la primera fila, los siguientes 3 a la segunda fila y los últimos 3 a la última fila. Siendo así la permutación final es $(1, 2, 3, 4, 5, 6, 7, 8, 9)$.

Aunque este problema consiste en encontrar el número mínimo de movimientos para acomodar el tablero, si vemos la respuesta como un número es claro que no avanzaremos mucho. Vamos a cambiar un poco el problema con el fin de poder definir subsoluciones: en lugar de buscar únicamente el mínimo número de movimientos para acomodar el tablero por el momento supondremos que también necesitamos saber qué movimientos hacer para acomodar el tablero.

Ahora podemos ver el espacio de búsqueda como un árbol enraizado, donde cada nodo representa una sucesión de movimientos, la raíz representa la ausencia de movimientos, los 8 nodos a distancia 1 de la raiz representan las 8 rotaciones(2 rotaciones diferentes en cada uno de los 4 lugares diferentes) que se pueden realizar, los 64 nodos a distancia 2 de la raíz representan los 64 pares de rotaciones que se pueden realizar, etc.

Debido a que hay 8 rotaciones válidas en este juego, de cada estado surgen 8 nuevos estados, lo que ocasiona que el árbol de búsqueda crezca demasiado rápido. Hay un nodo a distancia 0 de la raíz, 8 nodos a distancia 1, 64 nodos a distancia 2, 512 nodos a distancia 3, etc.

Es decir, a distancia n hay $0+8+64+...+8^n$ nodos, lo cual es equivalente a $\frac{8^{n+1}-1}{7}$ nodos. Si buscaramos una solución de tamaño 10 tendríamos que revisar un total de ¡153391689 nodos!.

Esto pudiera parecer un callejón sin salida, ya que realmente no podemos revisar todas las subsoluciones (sucesiones de movimientos, en este caso) que se pueden realizar.

Sin embargo hay algo que nos puede sacar de este apuro: Hay solamente $9! = 362880$ acomodos del tablero posibles. También debemos de notar que para dos subsoluciones $A = (a_1, a_2, ..., a_n)$ y $B = (b_1, b_2, ..., b_m)$ con $n < m$, si A y B llegan al mismo acomodo del tablero, entonces A puede ser prefijo de una solución pero B no puede ser prefijo de una solución.

Recordamos que nuestra definición de solución es aquella sucesión finita de movimientos que lleva el tablero del estado inicial al final y que además dicha sucesión tiene longitud mínima.

Si B fuera prefijo de una solución $S = (s_1, s_2, ..., s'_m)$, significaría que $b_1 = s_1$, $b_2 = s_2$, ..., $b_m = s_m$, es decir, que una solución sería realizar los m movimientos de B llevando al tablero a un acomodo P y luego realizar

$m' - m$ movimientos llevando al tablero al acomodo final.

Pero si eso sucediera entonces sería posible llevar al tablero al al mismo acomodo P realizando $n < m$ movimientos y luego realizar los movimientos $s_{m+1}, s_{m+2}, ..., s'_m$ llevando al tablero al estado final.

Es decir la subsolución $T = (a_1, a_2, ..., a_n, s_{m+1}, s_{m+2}, ..., s'_m)$ llevaría al tablero del acomodo inicial al acomodo final, y además la longitud de T sería $n + m' - m$ lo cual es menor que $m + m' - m = m'$ y contradice la suposición de que S es una solución.

En resumen, lo que probamos en estos párrafos fue que **si hay varias formas de llegar a un mismo acomodo del tablero, solamente hay que tomar en cuenta la mas corta**.

Si varias de esas subsoluciones usan el mismo número de movimientos, basta con tomar cualquiera de ellas y no perderemos nuestra oportunidad de encontrar una solución ya que al llegar al mismo acomodo del tablero, lo que se puede hacer a partir de ahí es exactamente lo mismo y lleva a los mismos resultados.

Así que lo que podemos hacer para resolver el problema, sin necesidad de abandonar aún nuestro modelo del árbol de decisiones, es realizar una búsqueda en amplitud teniendo en cuenta qué acomodo de tablero genera cada subsolución; y si dicho acomodo de tablero ya fué visitado antes, entonces ignorar dicho nodo.

Podemos hacer eso ya que en la búsqueda en amplitud la primera vez que encontremos cada acomodo de tablero tendremos la certeza de que lo habremos encontrado con la menor cantidad de movimientos(todos los nodos que se recorran después estarán igual o mas lejos de la raíz).

Finalmente, para implementar esta solución lo que necesitamos es una cola(para la búsqueda en amplitud), una manera de representar cada nodo, y una manera de marcar los acomodos o permutaciones ya visitadas.

La información relevante de cada nodo viene dada por dos cosas: la subsolución(los movimientos realizados) y la permutación de casillas a la cual se llegó. Aunque la primera cosa implique la segunda, recordemos que el problema solamente nos pide el número de movimientos realizados, entonces basta con que guardemos la permutación de casillas y el número de movimientos realizados.

Es tentador representar cada permutación por un arreglo de 9 enteros, sin embargo, para marcar la permutación como visitada resultaría mucho mas práctico el *número de permutación*, es decir, si se ordenaran todas las permutaciones lexicográficamente, ¿cuántas permutaciones serían menores que una permutación en particular?.

Número de Permutación

Averigüar el número de permutación es un problema interesante, el cual sugiere el uso de diversas estructuras de datos para permutaciones de muchos elementos, sin embargo como tenemos solamente 9 elementos, difícilmente una estructura de datos mejoraría el tiempo de ejecución de dicho algoritmo.

¿cuál es el número de permutación de $P = (7, 3, 2, 6, 9, 5, 8, 1, 4)$? o mejor dicho ¿cuántas permutaciones hay menores que P?

Los siguientes dos párrafos describen algo verdaderamente sencillo de pensar pero difícil de redactar de manera precisa, la idea trata de dada una permutación p, tomar el menor prefijo de p que no sea prefijo de P y luego notar que p es menor que P sí y solo sí estos dos prefijos son distintos. El lector deberá leer con cuidado lo siguiente pero sin miedo ya que no es nada profundo.

Vamos a llamar $C(p)$ al mayor prefijo común entre p y P. Por ejemplo $C(7, 1, 2, 3, 4, 5, 6) = (7)$, $C(7, 3, 2, 1, 4, 5, 6) = (7, 3, 2)$. Y sea $C_1(p)$ el prefijo de p cuya longitud excede en 1 a la longitud de $C(p)$(notamos que no tiene sentido hablar de $C_1(P)$). Por ejemplo $C(7, 3, 2, 1, 4, 5, 6) = (7, 3, 2, 1)$

Ahora, consideremos que p es una permutación cualquiera de los números de $1, ..., 9$ tal que $p \neq P$. Vamos a llamar m a la longitud de $C_1(p)$. Tenemos que p es menor que P sí y solo sí. $C_1(p)$ es menor que el prefijo de tamaño m de P.

Notemos que si una permutación p empieza con 1, 2, 3, 4, 5 ó 6 entonces la permutación p es menor que P, y además hay 8! permutaciones que inician con cada uno de estos números.

Si una permutación empieza con $(7, 1)$ o con $(7, 2)$ también esa permutación es menor que P y además hay 7! permutaciones que inician con cada uno de estos prefijos.

Continuando con este razonamiento, la permutación también será menor si inicia con $(7, 3, 1)$ y hay 6! permutaciones que cumplen esto.

El siguiente número en la permutación es 6, parece que encontraremos 5 prefijos de tamaño 4 tales que sus primeros 3 elementos sean $7, 3$ y 2 y que sean prefijos de puras permutaciones menores que P. Pero la realidad es otra, ya que solamente existen 3 prefijos: $(7, 3, 2, 1)$, $(7, 3, 2, 4)$ y $(7, 3, 2, 5)$.

Lo que sucede ahora es que los prefijos no pueden terminar en 2 ni en 3 ya que repetirían números en la permutación, así que debemos cuidarnos también de no contar los números que se repiten.

Es decir, si una permutación p es menor que P y m es la longitud de $C_1(p)$, entonces, el último número de $C_1(p)$ no puede estar en $C(p)$ ni tampoco ser mayor o igual que el $m-$ésimo número de P, es decir, los *números aceptables* son aquellos que no estan en $C(p)$ y son menores que el $m-$ésimo número de

P.

Para contar aquellos números aceptables para una longitud de prefijo determinada, vamos a definir una función M donde par un entero n entre 1 y 9, $M(n)$ representa la cantidad de números que no aparencen en el prefijo de tamaño n de P y que son menores que el $n-$ésimo número de P.

Por ejemplo, ya vimos que $M(1) = 6$, $M(2) = 2$, $M(3) = 1$ y $M(4) = 3$.

Por lo tanto el número de permtuaciones menores que P es:

$$M(9)0! + M(8)1! + M(7)2! + M(6)3! + M(5)4! \\ +M(4)5! + M(3)6! + M(2)7! + M(1)8!$$

Es interesante notar que si bien nos motivamos en el valor de la permutación de P para obtener esta fórmula, la fórmula no depende realmente de los valores de la permutación, solamente estamos suponiendo que P es una permutación de los números enteros de 1 a 9.

Si aplicaramos un algoritmo análogo para una permutación de los enteros de 1 a N, el algoritmo sería $O(N^2)$, puesto que evaluar la función M sería lineal y habría que evaluarla N veces. Sin embargo, como aquí N siempre vale 9, esta particularización del algoritmo es $O(1)$.

Para representar una permutación usaremos un arreglo de 9 caractéres, ya que los caracteres gastan menos memoria que los enteros.

A continuación se muestra una implementación del algoritmo que encuentra el número de permutación en una función llamada `menores`:

```
1  int menores(char perm[]){
2          int i, M, k, r;
3          int factorial=1;
4          r=0;
5          for(i=0;i<=8;i++){
6                  if(i!=0)
7                          factorial*=i;
8                  M=perm[9-i-1];
9                  for(k=0;k<9-i;k++){
10                         if(perm[k]<=perm[9-i-1]){
11                                 M--;
12                         }
13                 }
14                 r+=M*factorial;
15         }
16         return r;
17 }
```

Implementación

Volviendo al problema original, nos disponemos a usar una búsqueda en amplitud, donde cada elemento de la cola constará de una permutación representando un acomodo del tablero y el tamaño de la subsolución mas corta que llega a dicho acomodo, también será necesario un arreglo con 9! elementos para marcar los tableros que ya se lograron formar.

En la búsqueda en amplitud se visitaran solo aquellos nodos del árbol de deciciones que formen un acomodo del tablero que no se haya podido formar anteriormente.

Dado que nunca habrá mas de 9! elementos en la cola, podemos implementar la cola con un tamaño máximo de 9!, cada elemento de la cola consta de un arreglo de 9 caracteres y un entero. Por lo tanto la cola se pude implementar de la siguiente manera:

```
1 char ColaPerm[9*8*7*6*5*4*3*2*1][9];
2 int ColaMov[9*8*7*6*5*4*3*2*1];
3 int ColaInicio=0;
4 int ColaFin=0;
```

El arreglo para marcar los tableros ya formados puede ser usado también para guardar cuántos movimientos se necesitaron para formarlo, puede ser declarado asi:

```
1 int MinMov[9*8*7*6*5*4*3*2*1];
```

`MinMov` valdrá 0 cuando la permutación no se haya visitado, y valdrá el número de movimientos más uno cuando la permutación se haya visitado. Esto de sumarle 1 se debe a que la permutación inicial se forma con 0 movimientos.

Resulta conveniente marcar el arreglo MinMov dentro de la función para encolar:

```
1 void encolar(char perm[], int movs){
2         int m=menores(actual);
3         if(MinMov[m]==0){
4                 memcpy(ColaPerm[ColaFin], perm, 9*
                      sizeof(int)); //Copiar a ColaPerm[
                      ColaFin] el contenido de perm
5                 ColaMov[ColaFin]=movs;
6                 ColaFin++;
7                 MinMov[m]=mov;
8         }
9 }
```

Y para desencolar así:

```
1  int desencolar(char perm[]){
2          memcpy(perm, ColaPerm[ColaInicio], 9*sizeof(int
               ));
3          return ColaMov[ColaInicio++];
4  }
```

La función para desencolar regresa el número de movimientos y copia la permutación a un arreglo dado.

Notamos también que una rotación en sentido antihorario es equivalente a 3 rotaciones en sentido horario, por lo tanto solamente es necesario implementar la rotación en sentido horario:

```
1  void rota(char perm[], int fila, int col){
2          char *a, *b, *c, *d; //Estas variables
               apuntaran a las posiciones de las 4 casillas
                que se van a rotar
3          char ta, tb, tc, td; //Estas variables
               guardarán los valores iniciales de las 4
               casillas
4          fila--; col--; //Cambia la fila y columna de
               indices basados en 1 a indices basados en 0
5          a=&perm[fila*3+col];
6          b=&perm[fila*3+col+1];
7          c=&perm[(fila+1)*3+col];
8          d=&perm[(fila+1)*3+col+1];
9          ta=*a; tb=*b; tc=*c; td=*d; //Guarda el estado
               inicial en variables temporales
10         *a=tc; *b=ta; *c=td; *d=tb; //Rota en sentido
               horario
11 }
```

Una vez implementadas todas estas funciones la búsqueda en amplitud es algo muy sencillo:

```
1  encolar(actual, 1);
2  while(MinMov[0]==0){
3          movs=desencolar(actual);
4          for(i=1;i<=2;i++){
5                  for(k=1;k<=2;k++){
6                          rota(actual, i, k);
7                          encolar(actual, movs+1); //
                               Rotacion horaria
```

```
8                               rota(actual, i, k);
9                               rota(actual, i, k);
10                              encolar(actual, movs+1); //
                                    Rotación antihoraria
11                              rota(actual, i, k); //Despues
                                    de 4 rotaciones la
                                    permutación inicial regresa
                                    a como estaba al inicio
12                              }
13                      }
14              }
```

El código completo de esta solución(incluyendo la cabecera, la lectura y la escritura) mide menos de 80 líneas.

16.3. Empujando Cajas

El siguiente problema se usó en el 2005 en la Competencia Iberoamericana de Informática por Correspondencia.

Seguramente ya conoces un videojuego clásico que trata de que hacer que un obrero empuje cajas para colocarlas sobre una marcas dibujadas en el suelo.

Considera un juego de similar en el cual el obrero debe empujar solamente una caja a traves de un cuarto hacia una ubicación determinada.

El cuarto está representado por una cuadricula de 5 x 5 rodeada de una pared perimetral, dentro de la cuadricula hay dos tipos de casillas: las casillas libres y las casillas con pared.

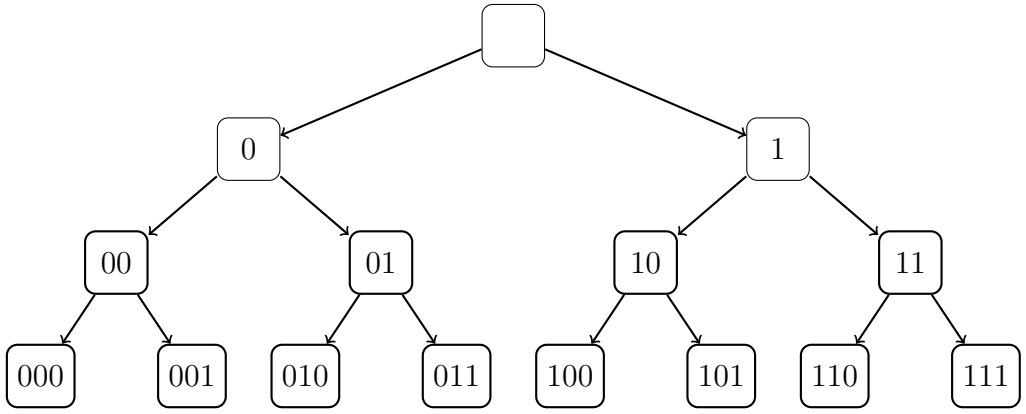
Además de esto, dentro del cuarto se encuentra el obrero y la caja, siempre en casillas diferentes.

El obrero puede moverse una casilla al norte, sur, este u oeste, siempre y cuando no ocupe la posición de una pared ni la posición de la caja.

En ciertas ocaciones el obrero puede empujar la caja pero no tiene suficientes fuerzas para tirar de ella.

El acto de empujar la caja se puede ejercer cuando el obrero está en una casilla vecina(horizontal o verticalmente pero NO en diagonal) a la casilla de la caja que empuja con la restricción de que la caja nunca puede ocupar una casilla con pared.

Luego de empujar la caja, el obrero pasa a ocupar la casilla que antes ocupaba la caja y la caja se mueve una casilla en la misma dirección que el obrero.



...

Figura 16.6: Los primeros 4 dígitos del árbol de decisiones de "Empujando Cajas"

Ejemplo 16.3.1. Escribe un programa que dado el mapa del cuarto, determine el número mínimo de pasos para que el obrero coloque la caja en la ubicación dada realizando el mínimo número de pasos. Donde un "paso" se define como la acción de moverse de una casilla a una casilla vecina.

El mapa del cuarto se representa por una cuadrícula de 7 por 7 caracteres donde:

- El caracter "X" indica pared
- El caracter "O"(no cero) indica la ubicación del obrero
- El caracter "C" la ubicación de la caja.
- El caracter "D" el lugar de destino de la caja.
- El caracter "."(ASCII 46) casilla libre.

El programa deberá funcionar en menos de un segundo.

Estructura de la Solución

Nuevamente nos debemos de preguntar: ¿qué es lo que nos está pidiendo el problema?. La respuesta es el número mínimo de pasos.

Procederemos como en la sección anterior, pensando en la solución como una secuencia finita de pasos, cada paso puede ser representado solamente por un entero del 0 al 3 indicando la dirección.

Así que diremos que las soluciones de este problema son aquellas secuencias finitas de pasos(representadas por enteros del 0 al 3) tales que hacen que el obrero empuje la caja hasta su posición de destino y que tienen longitud mínima.

En este problema cada subsolución será también una secuencia finita de pasos pero puede no llevar la caja a su lugar de destino.

Por lo tanto el árbol de decisiones lo podemos ver como un árbol enraizado en el cual la raíz es una cadena vacía y cada nodo representa una sucesión finita con los enteros del 0 al 3.

La figura 16.3 muestra los primeros 4 niveles(incluyendo la raíz) de este árbol de decisiones. Nuevamente podemos darnos cuenta de que el árbol de decisiones crece bastante rápido, con tan solo 10 niveles hay mas de un millón de nodos y en 15 niveles hay mas de mil millones de nodos.

Mejorando la Solución

En la sección anterior se resolvió el problema del juego de dígitos, en el cual salimos del apuro de tener un espacio de búsqueda demasiado grande dándonos cuenta que había varias manera de llegar a un mismo acomodo del tablero y una vez que se llega a un acomodo del tablero no importan qué movimientos se hicieron para llegar ahí sino cuántos movimientos se hicieron.

Trataremos de aplicar un razonamiento parecido aquí. Nuestro acomodo no es simplemente el lugar donde se encuentra el obrero, ya que el obrero puede mover la caja, pasar por su posición original y volver a mover la caja después. Dos subsoluciones pueden ser escencialmente distintas aún cuando el obrero se encuentre en la misma posición.

Sin embargo, si para dos casillas cualesquiera (a, b) hay varias formas de colocar la caja en a y que el obrero se mueva después a b, sin importar cual de todas esas formas usó el obrero para hacer eso. Las maneras de llevar la caja a su destino no cambian.

Es decir, para cualquier par de casillas (a, b) si dos subsoluciones x y y tales que $|x| < |y|$, que llevan al obrero a la casilla b y a la caja a la casilla a, se tiene que y no es prefijo de ninguna solución, puesto que si es posible llevar la caja a su destino, no la llevaría en el mínimo número de pasos.

Vamos a definir entonces un estado como un par de casillas (a, b) y vamos a decir que una subsolución pertenece a dicho estado si luego de ejecutar los pasos de la subsolución, la caja termina en la casilla a y el obrero termina en la casilla b.

Asi que haremos una búsqueda en amplitud en el árbol de decisiones, cada que se visite una subsolución, revisaremos si el estado al cual pertenece no había sido generado antes antes, y si había sido generado no visitaremos esa subsolución,de esta manera todos los estados se generarán unicamente en el nodo mas cercano a la raíz, es decir, para cada estado encontraremos una de sus subsoluciones mas cortas, y solamente una.

La solución podría ser cualquiera de las subsoluciones (a, D) donde D es la posición de destino de la caja.

Implementación

Como se había mencionado anteriormente, vamos a hacer una búsqueda en amplitud en el árbol de decisiones.

El problema solo nos pide la longitud de la subsolución y no la solución completa, entonces no es necesario guardar en la cola todo sino solamente el tamaño de la subsolución encontrada y el estado que genera dicha subsolución.

Como habíamos dicho antes, una subsolución es un par de casillas. Pero necesitamos representar las casillas de alguna manera. Es posible representarlas por un par de enteros indicando la fila y la columna, pero como sabemos que el mapa es de 7x7 entonces sería mas sencillo numerar las casillas de 0 a 48 y representar cada casilla por un solo entero.

Digamos que las casillas de la primera fila tienen números del 0 al 6, de la segunda fila del 7 al 13, etc. Si en cada fila a cada casilla le asignamos los números en forma creciente de izquierda a derecha tendremos que a una casilla ubicada en f-ésima fila y la c-ésima columa, le correspondería el número $7(f-1)+(c-1)$. Esto sugiere que es conveniente tanto en filas como columnas asignarles números de 0 a 6.

Por lo tanto, dado el número de casilla, obtener la fila y la columna es particularmente fácil:

```
1 int obtenFila(int casilla){
2         return casilla/7;
3 }
4 int obtenColumna(int casilla){
5         return casilla%7;
6 }
```

Y dada la fila y la columna, obtener el número de casilla también es fácil:

```
1 int obtenCasilla(int fila, int columna){
2         return 7*fila+columna;
3 }
```

Usaremos un arreglo llamado `Mejor` para guardar la longitud de la subsolución mas corta para generar cada estado. La cola, como ya se había mencionado, debe de guardar un estado y la longitud de la subsolución, es decir, tres enteros.

Para representar los movimientos al norte, sur, este y oeste, vamos a usar un par de *vectores de dirección*, es decir, dos arreglos Df y Dc tales que para alguna casilla en fila f y columna c, $f + Df[i]$ y $c + Dc[i]$ representen una casilla vecina a la casilla original, indicando i la dirección con un entero de 0 a 4.

Así que estas son las estructuras que se definirán en el problema:

```
1  int Mejor[49][49];
2
3  int ColaA[49*49];
4  int ColaB[49*49];
5  int ColaTam[49*49];
6  int ColaInicio;
7  int ColaFin;
8
9  int Df[]={0, 1, 0, -1};
10 int Dc[]={1, 0, -1, 0};
```

Para encolar un estado, en lugar de usar como parámetros las dos casillas y la longitud de la subsolución, resulta mas cómodo pasar la fila y la columna de cada una de las casillas.

```
1  void encolar(int f1, int c1, int f2, int c2, int len){
2          int a, b;
3          a=obtenCasilla(f1, c1);
4          b=obtenCasilla(f2, c2);
5          if(Mejor[a][b]==0){ //Si el estado no había
               sido encontrado antes
6                  Mejor[a][b]=len;
7                  ColaA[ColaFin]=a;
8                  ColaB[ColaFin]=b;
9                  ColaTam[ColaFin]=len+1;
10                 ColaFin++;
11         }
12 }
```

Al final la búsqueda en amplitud queda de esta manera:

```
1  void amplitud(int O, int C){
2          int f1, c1, f2, c2, len;
```

```
3           int f, c;
4           int a, i;
5           encolar(obtenFila(O), obtenColumna(O),
                obtenFila(C), obtenColumna(C), 0);
6           while(ColaInicio!=ColaFin){
7                   f1=obtenFila(ColaA[ColaInicio]);
8                   c1=obtenColumna(ColaA[ColaInicio]);
9                   f2=obtenFila(ColaB[ColaInicio]);
10                  c2=obtenColumna(ColaB[ColaInicio]);
11                  len=ColaTam[ColaInicio];
12                  ColaInicio++;
13                  for(i=0;i<4;i++){
14                          f=f1+Df[i];
15                          c=c1+Dc[i];
16                          if(Mapa[f][c]!='X'){ //Si el
                                obrero se mueve a una
                                posición donde no hay pared
17                                  if(f==f2 && c==c2){ //
                                        Si empuja la caja
18                                          if(Mapa[f2+Df[i
                                                ]][c2+Dc[i
                                                ]]!='X'){ //
                                                Si la caja
                                                se mueve a
                                                una posición
                                                 donde no
                                                hay pared
19                                                  encolar
                                                        (f,
                                                        c,
                                                        f2+
                                                        Df[i
                                                        ],
                                                        c2+
                                                        Dc[i
                                                        ],
                                                        len
                                                        +1);
20                                          }
21                                  }else{//Si se mueve a
                                        suelo libre
```

```
22                                                  encolar(f, c,
                                                        f2, c2, len
                                                        +1);
23                                              }
24                                          }
25                                      }
26                              }
27  }
```

El programa completo que resuelve este problema posee poco menos de 100 líneas.

16.4. Camino Escondido

Este problema apareció en un examen preselectivo para la IOI del año 2007, el contexto del problema es el siguiente:

Imagina que estas a punto de entrar en la pirámide que antecede a la sala donde se encuentran los legendarios tesoros de los mayas, y en la puerta esta escrito un extraño jeroglifico el cual se traduce de la siguiente manera:

Aqui no pueden entrar seres que no sean descendientes de nuestra gloriosa civilizacion, para demostrar que por tus venas corre sangre maya deberás demostrar tus habilidades para rastrear el camino siguiendo un patron que se repite hasta llegar a la salida

El mapa de la piramide tiene forma de cuadricula y mide N x N, donde $N \leq 50$, los movimientos pueden ser de manera vertical u horizontal, pero no estan permitidos movimientos en diagonal.

La entrada a la cámara esta representada por un "1" y se encuentra en la parte superior, la salida tambien esta representada con un "1" pero se encuentra en la parte inferior, el patron que debes de seguir debe de coincidir con los numeros que estan dibujados en el suelo

Por ejemplo considera el siguiente mapa:

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

Si el patron a seguir fuera $(3, 5, 8)$ el número mínimo de pasos para llegar de la entrada a la salida seria 8 (en la figura el camino está resaltado con negritas).

Ejemplo 16.4.1. Escribe un programa que dado un mapa de la pirámide encuentre el número mínimo de pasos para desde la entrada hasta llegar a la salida.

El programa deberá funcionar en menos de un segundo.

Estructura de la Solución

De igual manera que en los problemas anteriores, la pregunta que todo lector se debe hacer es: **¿qué estoy buscando?**.

Para contestar correctamente la pregunta es necesario haber leído con atención el problema y comprenderlo correctamente. En este caso específico, estamos buscando el camino válido más corto entre la celda A y la celda B.

Ya sabemos lo que estamos búscando. ¿Cuál es el siguiente paso? Por supuesto, algo que parece obvio y sin embargo poca gente hace; preguntarse ¿Cómo es lo que estoy buscando? Para cuestiones de problemas de informática esto se traduce: ¿Cómo se representa lo que estoy buscando?

¿Cómo se representa lo que estoy buscando? Para este problema, un camino se representa como una secuencia de celdas contiguas, ya sea de manera horizontal o vertical, de la matriz, que inicia en la celda A y termina en la celda B.

Una vez especificado el objeto que buscamos, surgen varias otras preguntas interesantes sobre las características del mismo. Estas preguntas se aplican a casos particulares y no es sencillo generalizarlas como las dos anteriores, sin embargo, basta decir que una vez especificado el objeto de la búsqueda deben intentar conocerse la mayor cantidad de características posible del mismo.

Un punto importante para el problema que estamos investigando es:

¿Cuál es el largo máximo que puede tener un camino? Inicialmente podríamos pensar que el largo máximo posible es el número de celdas en la matriz, es decir, N x N.

Pero leyendo el problema con atención podremos observar que nunca se dijo que el camino no podía pasar 2 veces por la misma celda. Eliminada esta limitante vemos entonces que el largo máximo de un camino puede ser mayor que el número total de celdas de la matriz. De hecho, el largo máximo de un camino puede ser MUUUY grande.

Muy bien, ya sabemos que estamos buscando, también sabemos como es. Sigamos adelante, aqui viene la pregunta más importante hasta el momento: **¿en donde debemos buscarlo?**

¿En donde debemos buscar? Esta es la pregunta clave en cualquier problema de búsqueda y su correcta respuesta es lo que nos definirá el espacio de búsqueda. Volvamos a nuestro problema, la primera opción que se ocurre es

construir todos los caminos posibles (válidos e inválidos) y entre ellos buscar el que queremos.

Dado que no definimos una cota superior para el largo máximo del camino, la cantidad de caminos que podemos construir es hasta el momento infinita, y aunque las computadoras son muy rápidas, eso es insuficiente para buscar entre un número infinito de caminos.

Equivalencia de Subsoluciones

Una vez definido un primer espacio de búsqueda, el siguiente paso es recortarlo lo más posible hasta llegar a una cardinalidad manejable. Para nuestro problema hay dos recortes inmediatos:

- En vez de construir todos los caminos, construyamos únicamente los que sean válidos. Este puede parecer un buen recorte, sin embargo, al no conocer los datos de entrada de antemano, no podemos estar seguros de que sirva de algo. Un caso de prueba planeado cuidadosamente puede hacer que el número de caminos válidos sea tan grande que para fines prácticos buscar entre ellos sea como buscar en un número infinito de opciones.

- Dado que queremos el camino más corto, hay que buscar entre todos los caminos válidos empezando por los de menor longitud. Este recorte, a menos que el único camino sea el más largo posible, obviamente nos ahorra algo. Pero de nuevo, al no saber que tan largo es el camino que buscamos, no podemos estar seguros de que lo encontraremos en un tiempo corto.

Hemos avanzado algo, sin embargo, aún no llegamos a un punto en donde podamos asegurar una correcta solución en tiempo para cualquier instancia posible del problema.

Es muy importante siempre calcular el tamaño del espacio de búsqueda y pensar que en el peor de los casos nuestro programa va a tener que buscar en la totalidad del espacio, piensen en el caso en el que no haya solución, para estar seguros tenemos que buscar entre todas las opciones, de modo que requerimos que el tamaño del espacio sea tal que aún examinándolo completamente nuestro programa termine en el tiempo establecido.

Para nuestro problema, dado que un camino puede ser una solución, una subsolución sería un prefijo del camino. En los dos problemas anteriores hemos logrado exitosamente encontrar cierta *redundancia* en las subsoluciones (es decir, hay pares de soluciones tales que no todo sufijo válido para alguna de ellas es sufijo válido para la otra).

A esta *redundancia* de subsoluciones la llamaremos equivalencia. La idea es que dos subsoluciones son equivalentes si la forma de llegar desde ellas a la solución búscada es exactamente la misma.

Para definir la equivalencia de subsoluciones de una manera mas precisa, vamos a cambiar nuestro concepto de solución. Diremos que una solución es aquella secuencia de decisiones que lleve al resultado deseado, sin importar el número de decisiones que se tomen, es decir, solamente descartamos que una solución es necesariamente el camino mas corto; en este problema vamos a aceptar como solución cualquier camino.

De esa manera lo que buscamos ahora es una solución tal que su longitud sea la mas corta. Una vez aclarado esto estamos listos para definir la equivalencia de subsoluciones:

Definición 16.4.1 (Equivalencia de Subsoluciones)**.** Sean a y b dos subsoluciones, se dice que a y b son equivalentes si y solo sí:

- Para cualquier solución S tal que S resulta de la concatenación de a con alguna c(es decir, $S = (a_1, a_2, ..., a_n, c_1, c_2, ..., c_m)$), se tiene que la concatenación de b con c también es una solución.

- Para cualquier solución S tal que S es la concatenación de b con alguna c, se tiene que la concatenación de a con c también es una solución.

¿De qué nos sirve tener sub-soluciones equivalentes? En realidad eso depende un poco del problema específico, sin embargo, en la mayoría de los casos el beneficio viene de que dos sub-soluciones se pueden comparar de manera directa y en base a eso decidir cual de las dos es óptima dados los criterios que buscamos.

Notamos que si a y b son dos subsoluciones equivalentes, también b y a también son equivalentes.

Si a es equivalente con b, y b es equivalente con c, entonces a es equivalente con c(la demostración surge directamente de la definición anterior).

Además, toda subsolución es equivalente consigo misma.

Estas 3 propiedades fueron las que utilizamos en la conectividad de los grafos para ver cómo un grafo se podía particionar en componentes conexas.

Nos vemos tentados a definir algo totalmente análogo a las componentes conexas pero sustituyendo las aristas con las equivalencias. Sin embargo esto puede complicar el problema, definiremos algo parecido, lo cual llamaremos estados.

Estados

La idea de los estados es poder particionar adecuadamente el espacio de búsqueda. Es decir, cada estado es un conjunto de subsoluciones y ninguna subsolución está en mas de un estado, todo esto de manera que cada decisión que se tome, o cada paso que se dé, o como se le quiera llamar al hecho de descender un nivel en el árbol de decisiones sea equivalente a cambiar de un estado a otro.

También se requiere que todas las subsoluciones que pertenezcan a un mismo estado sean equivalentes(obsérvese que puede que subsoluciones asociadas a estados distintos también sean equivalentes).

Hay que aclarar que existen muchas formas válidas de definir los estados, pero todas deben de cumplir con esas condiciones que se acaban de mencionar.

Aterrizando un poco, en el problema de camino escondido, ¿Cuáles serían dos sub-soluciones equivalentes?

Dijimos anteriormente que una subsolución es cualquier prefijo de un camino solución, como un camino es un sucesión de celdas de la matriz, entonces una sub-solución se definiría como una sucesión de celdas en la matriz que va de la celda A a la celda Q.

La solución debe de seguir los números especificados por la secuencia S, de modo que nuestra subsolución que termina en Q además se encuentra en una cierta posición de la secuencia S a la que llamaremos p.

Supongamos ahora que conocemos la solución desde (Q, p) hasta B, entonces todas las subsoluciones que terminen en (Q, p) son equivalentes ya que su camino hacia B es el mismo sin importar la manera en la que llegaron a (Q, p).

Para nuestro problema, lo mas natural es identificar los estados por el par ordenado (Q, p). Aunque haya estados para los cuales no hay subsoluciones asociadas(es decir, estados vacíos), eso no representa ninguna dificultad.

Queremos el camino más corto. Si tenemos dos subsoluciones que llevan al estado (Q, p), su camino más corto hacia la celda B es el mismo, por lo tanto, podemos decir que la subsolución cuya longitud sea menor entre las dos es mejor para nuestros propositos, si ambas son de la misma longitud, entonces podemos simplemente tomar cualquiera de las dos, ya que, a fin de cuentas, son equivalentes.

Esto nos sugiere hacer una búsqueda amplitud en los estados. Si el lector regresa a las secciones anteriores y mira los códigos de las búsquedas en amplitud, se dará cuenta que realmente lo que se hizo fue una búsqueda en amplitud con los estados.

Habiendo visto lo que son los estados y las subsoluciones equivalentes

estamos listos para llegar a nuestra definición final de espacio de búsqueda:

Definición 16.4.2 (Espacio de Búsqueda)**.** Es un par ordenado $E = (S, T)$, donde a los elementos de S los llamaremos estados y los elementos de T los llamaremos cambios o transiciones de estado. Además se deben cumplir las siguientes propiedades:

- Todo estado es un conjunto de subsoluciones(prefijos de *candidatos* a solución).
- Para todo par de estados a y b tales que $a \neq b$ se tiene que a y b son disjuntos.
- Para todo estado e, si tomamos cualesquiera $s_1, s_2 \in e$, se tiene que s_1 y s_2 son equivalentes.
- Los elementos de T son pares ordenados de los elementos de S, y $(e_1, e_2) \in T$ sí y solo sí existen $t_1 \in e_1$ y $t_2 \in e_2$ tales que t_1 es prefijo de t_2 y $|t_1| = |t_2| - 1$.

De las tres propiedades de esta definición, solo la cuarta propiedad parece perderse un poco en detalles sin mostrar una motivación adecuada. Esta propiedad se puede recordar simplemente como que dos estados estan conectados sí y solo sí es posible moverse de uno a otro en solamente un paso.

También es conveniente notar que los espacios de búsqueda son grafos dirigidos. Así que el proceso de resolver un problema de optimización combinatoria se reduce a identificar un espacio de búsqueda de tamaño razonable, identificar en ese espacio de búsqueda que es lo que se quiere optimizar(en este problema sería el número de transiciones) y dadas esas dos condiciones, elegir cómo recorrer el espacio de búsqueda.

Como lo que queremos minimizar es el número de transiciones de estado, la búsqueda en amplitud nuevamente nos va a servir.

No tenemos que buscar dos veces en un mismo estado, de modo que nuestra búsqueda se limita a la cantidad de estados que existan.

Para nuestro caso todas los pares (Q, p) posibles identifican los estados. Dado que Q es una celda de la matriz, existen N x N posibles valores para Q. Asi mismo, p representa una posición en la secuencia S así que la cantidad de valores posibles para p es igual al largo de la secuencia S.

Sustituyendo por los valores máximos posibles tenemos que en el peor caso nuestra búsqueda abarcara un total de $(300)(300)(50) = 4500000$ estados, lo cual funcionará fácilmente en un segundo.

Implementación

Aunque vamos a hacer una búsqueda en un grafo dirigido y anteriormente se habló de la implementación de grafos con matriz de adyacencia y listas de adyacencia, esta vez nos conviene representar los vértices como ternas ordenadas de enteros.

Segun este planteamiento, una terna (p, f, c) representará el estado que contiene los caminos que terminan en la casilla (f, c) y en la posición p del patrón.

Para marcar los estados ya visitados y al mismo tiempo guardar el tamaño del camino mas corto para llegar a cada estado usaremos un arreglo de 3 dimensiones llamado `Marca`, de manera que `Marca[p][f][c]` indique el camino mas corto para llegar al estado (p, f, c), o -1 si no se ha encontrado dicho camino.

Por lo tanto hay que declarar el siguiente arreglo:

```
1  int Marca[50][52][52];
```

El motivo para declarar el arreglo de tamaño 50 x 52 x 52 en lugar de 50 x 50 x 50 es para usar la columna 0, la fila 0, la columna $N+1$ y la fila $N+1$ como una *frontera*.

Si se utilizan esta noción de la frontera en el arreglo para marcar los estados, también hay que usarla en el arreglo que guarda el mapa:

```
1  int Mapa[50][52][52];
```

También hay que recalcar que en alguna parte del código hay que inicializar todas las posiciones del arreglo con -1.

Para implementar la búsqueda en amplitud es necesaria una cola, le asignaremos un tamaño máximo de 50^3.

```
1  int ColaInicio, ColaFin;
2  int ColaP[50*50*50];
3  int ColaF[50*50*50];
4  int ColaC[50*50*50];
```

También es necesario implementar una función para meter un estado a la cola; como nunca se deben de encolar estados ya visitados, suena razonable que la función de meter a la cola verifique que el estado no haya sido visitado y una vez metido a la cola lo marque como visitado:

```
1  void encolar(int p, int f, int c, int w){
2          if(Marca[p][f][c]==-1){
3                  ColaP[ColaFin]=p;
4                  ColaF[ColaFin]=f;
```

```
5                       ColaC[ColaFin]=c;
6                       Marca[p][f][c]=w;
7                       ColaFin++;
8               }
9       }
```

El patrón lo guardaremos en un arreglo llamado `Patron`:

```
1   int Patron[50];
```

Como se puede apreciar, se pretende guardar datos a partir de la posición 0 y no a partir de la posición 1. Los índices basados en 0 son adecuados para este momento ya que si el índice de una posición del patrón fuera p entonces el índice de la siguiente posición sería `(p+1)%P`.

Al igual que en el ejemplo anterior, este algoritmo habla de movimientos entre cuadros vecinos de una cuadrícula, por lo que es conveniente usar los vectores de dirección aquí también:

```
1   int Df[]={0, -1, 0, 1};
2   int Dc[]={1, 0, -1, 0};
```

En este problema en particular, el procedimiento de meter en la cola simplifca las cosas si se implementa en una función aislada, pero el procedimiento de sacar de la cola puede implementarse dentro de la búsqueda sin que esto haga el código difícil de manejar.

Como queremos que luego de dar el primer paso el estado esté ubicado en la primera posición del patrón, entonces antes de dar el primer paso vamos a considerar que el estado está ubicado en la última posición del patrón.

Así que la búsqueda en amplitud se puede implementar de la siguiente manera:

```
1               encolar(P-1, 1, inicioC, 0);
2               while(ColaInicio!=ColaFin){
3                       f=ColaF[ColaInicio];
4                       c=ColaC[ColaInicio];
5                       p=ColaP[ColaInicio];
6                       ColaInicio++;
7                       for(i=0;i<4;i++){
8                               k=Mapa[f+Df[i]][c+Dc[i]];
9                               if(k==1 || Patron[(p+1)%P]==k){
10                                      encolar((p+1)%P, f+Df[i
                                            ], c+Dc[i], Marca[p
                                            ][f][c]+1);
11                              }
```

```
12                      }
13              }
```

Las variables P e *inicioC* indican el tamaño del patrón y el número de columna inicial respectivamente.

Con esto la solución del problema está prácticamente completa, solo falta imprimir la salida y leer la entrada; esta parte, debido a que es poco interesante se deja como ejercicio al lector para que verifique si comprende bien esta implementación.

Bibliografía

[1] Cesar Arturo Cepeda García, **Espacio de Búsqueda** 2007

[2] Ian Parberry y William Gasarch, **Problems on Algorithms** 2002

[3] Arthur Engel, **Problem Solving Strategies** Springer 2000

[4] Maria Luisa Pérez Seguí **Combinatoria** 2005

[5] Bernard Kolman, Robert C. Busby y Sharon Cutler Ross, **Estructuras de Matemáticas Discretas para la Computación** Editorial Pearson Educación 1995

[6] Francisco Javier Zaragoza Martínez **Una Breve Introducción a la Recursión** 2004