Análisis de algoritmos

Notas de clase basadas en
Análisis de Algoritmos y Complejidad Computacional de Ian Parberry

Dr. Francisco Javier Zaragoza Martínez
franz@correo.azc.uam.mx

UAM Azcapotzalco
Departamento de Sistemas

Trimestre 2012 Otoño

Contenido

Contenido

Contenido

Complejidad computacional

- La teoría de la complejidad computacional es el estudio del costo involucrado en la solución de los problemas.
- Se desea medir la cantidad de recursos: tiempo, espacio, etc.
- Y se desean analizar al menos dos aspectos:
  - Cota superior: dar un buen algoritmo.
  - Cota inferior: probar que ningún algoritmo es mejor.
- El análisis de algoritmos es el análisis de los recursos usados por un algoritmo dado.

Clases de complejidad

- La cantidad de recursos se puede ver como una función.
- Las funciones se pueden catalogar en **clases de complejidad**:
  - Logarítmicas ($\log_2 n$).
  - Lineales ($3n + 1$).
  - Cuadráticas ($n^2 - n + 1$).
  - Polinomiales ($n^4 + n^2 + 1$).
  - Exponenciales (2^n).
  - Factoriales ($n!$).
- De polinomial hacia arriba es **bueno**. Hacia abajo es **malo**.

Motivación

- Los algoritmos eficientes llevan a programas eficientes.
- Los programas eficientes se venden mejor.
- Los programas eficientes hacen mejor uso del hardware.
- ¡Los programadores que escriben programas eficientes son mejores!

Factores que afectan la eficiencia

- El problema que se esté resolviendo.
- El lenguaje de programación.
- El compilador.
- El hardware.
- La habilidad del programador.
- La efectividad del programador.
- El algoritmo.

Contenido

Inducción matemática

- La **inducción matemática** es una técnica de demostración que se puede aplicar a muchos tipos de problemas.
- Para demostrar que una cierta propiedad se cumple para todo natural n se deben probar dos cosas:
  1. Que la propiedad se cumple para $n = 1$.
  2. Para toda $n \geq 1$, si la propiedad se cumple para n entonces también se cumple para $n + 1$.
- A esto se le llama el **caso base** y el **caso inductivo**.

Alternativas

- El caso base puede cambiar:
  1. La propiedad se cumple para $n = 3$.
  2. Para toda $n \geq 3$, si la propiedad se cumple para n entonces también se cumple para $n + 1$.
- Puede haber varios casos base:
  1. La propiedad se cumple para $n = 1$, $n = 2$ y $n = 3$.
  2. Para toda $n \geq 3$, si la propiedad se cumple para n entonces también se cumple para $n + 1$.
- El caso inductivo puede cambiar:
  1. La propiedad se cumple para $n = 1$.
  2. Para toda $n \geq 1$, si la propiedad se cumple para toda $1 \leq m \leq n$ entonces también se cumple para $n + 1$.

Primer ejemplo de inducción

- Demuestre que $1 + 2 + \cdots + n = n(n + 1)/2$ para toda $n \geq 1$.
- Primero para $n = 1$: el lado izquierdo es igual a 1, el lado derecho es igual a $1(1 + 1)/2$ y son iguales.
- Ahora suponga que $1 + 2 + \cdots + n = n(n + 1)/2$ para alguna $n \geq 1$. Entonces el lado izquierdo es:

$$\begin{aligned} 1 + 2 + \cdots + n + (n + 1) &= n(n + 1)/2 + (n + 1) \\ &= (n + 1)(n/2 + 1) \\ &= (n + 1)(n + 2)/2. \end{aligned}$$

- Que es igual al lado derecho correspondiente.

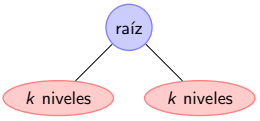
Segundo ejemplo de inducción

- Demuestre que si $1 + x > 0$ entonces $(1 + x)^n \geq 1 + nx$ para toda $n \geq 1$.
- Para $n = 1$ notamos que ambos lados son iguales.
- Ahora suponga que $(1 + x)^n \geq 1 + nx$ para alguna $n \geq 1$. Entonces:

$$\begin{aligned} (1 + x)^{n+1} &= (1 + x)(1 + x)^n \\ &\geq (1 + x)(1 + nx) \\ &= 1 + (n + 1)x + nx^2 \\ &\geq 1 + (n + 1)x. \end{aligned}$$

- Que es lo que queríamos demostrar.

Árboles binarios completos



- Demuestre que un **árbol binario completo** con k niveles tiene exactamente $2^k - 1$ nodos.
- Para empezar, observe que un árbol binario completo con 1 nivel tiene exactamente $1 = 2^1 - 1$ nodo.
- Para terminar, observe que un árbol binario completo con $k + 1$ niveles consiste de la raíz y dos árboles binarios completos con k niveles.
- Por lo tanto tiene $1 + 2(2^k - 1) = 2^{k+1} - 1$ nodos.

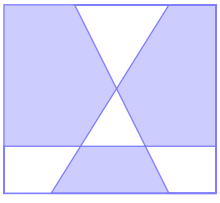
Un ejemplo de una desigualdad

- Demuestre que $\sum_{i=1}^{n} \frac{1}{2^i} < 1$ para toda $n \geq 1$.
- Para $n = 1$ el lado izquierdo vale $\frac{1}{2} < 1$.
- Ahora suponga que el enunciado es cierto para n, entonces:

$$\begin{aligned} \sum_{i=1}^{n+1} \frac{1}{2^i} &= \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n+1}} \\ &= \frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n}\right) \\ &= \frac{1}{2} + \frac{1}{2}\sum_{i=1}^{n} \frac{1}{2^i} \\ &< \frac{1}{2} + \frac{1}{2} \cdot 1 \\ &= 1. \end{aligned}$$

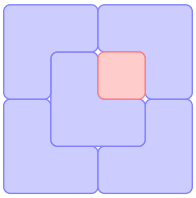
- Que es lo que queríamos demostrar.

Un ejemplo geométrico



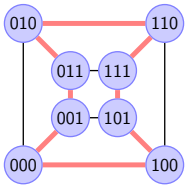
- Demuestre que cualquier conjunto de regiones definidas por n rectas en el plano se puede colorear con sólo dos colores de modo que no existan dos regiones del mismo color que compartan una arista.

Un ejemplo de rompecabezas



- Un **trominó** es una figura en forma de L formada por la yuxtaposición de tres cuadrados unitarios.
- Demuestre que se puede **tapizar** con trominós cualquier cuadrícula de $2^n \times 2^n$ a la que le falta un cuadrito en una posición arbitraria.
- ¿Qué puede decir de las cuadrículas de $n \times n$?

Un ejemplo combinatorio



- Un **código Gray** es una secuencia circular de los 2^n distintos números binarios de n bits de modo que cada pareja de números consecutivos difiere exactamente en un bit.
- Demuestre que existen códigos Gray para toda $n \geq 1$.

Contenido

Correctitud

- ¿Cómo sabemos que un algoritmo funciona?
  - Lo probamos con ejemplos.
  - Demostramos que funciona.
- Puede ser que las simples pruebas no encuentren errores, por lo que sería peligroso sólo hacer pruebas.
- ¡Las demostraciones también pueden contener errores!
- En la práctica se usa una combinación de las dos.

Correctitud de algoritmos recursivos

- Se demuestra por inducción en el **tamaño** del problema a resolver (por ejemplo, la cantidad de elementos en un arreglo o el número de bits en un entero).
- La base de la recursión es la base de la inducción.
- Se debe demostrar que las llamadas recursivas son en efecto subproblemas, es decir, que no se tiene una recursión infinita.
- El paso inductivo se hace suponiendo que las llamadas recursivas funcionan correctamente para demostrar que la llamada actual también funciona correctamente.

Números de Fibonacci recursivos

- $F_0 = 0$, $F_1 = 1$ y $F_n = F_{n-2} + F_{n-1}$ para $n \geq 2$.
- Considere la función fib(n) que regresa n si $n \leq 1$ y fib($n-2$) + fib($n-1$) en caso contrario.
- Demostremos por inducción que fib(n) regresa F_n.
- Para $n = 0$ la llamada fib(n) regresa $0 = F_0$.
- Para $n = 1$ la llamada fib(n) regresa $1 = F_1$.
- Suponga que $n \geq 2$ y que para toda $0 \leq m < n$ la llamada fib(m) regresa F_m.
- Entonces, la llamada fib(n) regresa

$$\text{fib}(n-2) + \text{fib}(n-1) = F_{n-2} + F_{n-1} = F_n$$

que es lo que queríamos demostrar.

Máximo recursivo

- Sea $A_1, \ldots, A_n$ un arreglo con n enteros y considere la función máximo(n) que regresa A_1 si $n \leq 1$ y máx(máximo($n-1$), A_n) en caso contrario.
- Demostremos por inducción que máximo(n) regresa máx($A_1, \ldots, A_n$).
- Para $n = 1$ la llamada máximo(n) regresa A_1.
- Suponga que $n \geq 1$ y que máximo(n) regresa máx($A_1, \ldots, A_n$).
- Entonces, la llamada máximo($n+1$) regresa

$$\text{máx}(\text{máximo}(n), A_{n+1})$$

es decir

$$\text{máx}(\text{máx}(A_1, \ldots, A_n), A_{n+1})$$

o bien

$$\text{máx}(A_1, \ldots, A_n, A_{n+1}).$$

Multiplicación recursiva

- Para toda x real definimos $\lfloor x \rfloor$ como el mayor entero que no excede x (la parte entera de x).

Función multiplica(y, z)

1. Si $z = 0$ entonces regresa 0.
2. Si z es impar
   1. entonces regresa multiplica($2y, \lfloor z/2 \rfloor$) + y,
   2. si no regresa multiplica($2y, \lfloor z/2 \rfloor$).

- Demostremos por inducción en z que para toda $y, z \geq 0$ la llamada multiplica(y, z) regresa yz.

Demostración

- Para $z = 0$ la llamada multiplica(y, z) regresa 0.
- Ahora suponga que $z \geq 0$ y que para toda $0 \leq q \leq z$ la llamada multiplica(y, q) regresa yq.
- ¿Qué es lo que regresa multiplica$(y, z+1)$?
- Si $z+1$ es impar entonces multiplica$(y, z+1)$ regresa

$$\begin{aligned} \text{multiplica}(2y, \lfloor (z+1)/2 \rfloor) + y &= 2y\lfloor (z+1)/2 \rfloor + y \\ &= 2y(z/2) + y \\ &= y(z+1). \end{aligned}$$

- Si $z+1$ es par entonces multiplica$(y, z+1)$ regresa

$$\begin{aligned} \text{multiplica}(2y, \lfloor (z+1)/2 \rfloor) &= 2y\lfloor (z+1)/2 \rfloor \\ &= 2y(z+1)/2 \\ &= y(z+1). \end{aligned}$$

Correctitud de algoritmos iterativos

- Analice el algoritmo un ciclo a la vez, comenzando en el ciclo más interno en el caso de ciclos anidados.
- Proponga un **invariante** para cada ciclo, es decir, un enunciado que permanezca verdadero a lo largo de la ejecución del ciclo y que capture el progreso hecho por el ciclo.
- Demuestre que el invariante se cumple.
- Use el invariante para demostrar que el ciclo termina.
- Use los invariantes para demostrar que el algoritmo calcula el valor correcto.

Notación

- Nos concentraremos en algoritmos con sólo un ciclo.
- Denotaremos por x_i al valor de la variable x inmediatamente después de que termine la i-ésima iteración del ciclo.
- Esto significa que x_0 es el valor de x justo antes de comenzar el ciclo.
- Usaremos esta notación para indicar el funcionamiento del algoritmo.

Números de Fibonacci iterativos

Función fib(n)

1. Si $n = 0$ entonces regresa 0.
2. $a \leftarrow 0$, $b \leftarrow 1$, $i \leftarrow 2$.
3. Mientras $i \leq n$ haz:
   1. $c \leftarrow a + b$,
   2. $a \leftarrow b$,
   3. $b \leftarrow c$,
   4. $i \leftarrow i + 1$.
4. Regresa b.

- Demostraremos que fib(n) regresa F_n.

Valores de las variables

- La variable i cumple que $i_0 = 2$ e $i_{j+1} = i_j + 1$.
- La variable a cumple que $a_0 = 0$ y $a_{j+1} = b_j$.
- La variable b cumple que $b_0 = 1$ y $b_{j+1} = c_{j+1}$.
- La variable c cumple que $c_{j+1} = a_j + b_j$.

Ejemplo con $n = 6$

j	i_j	a_j	b_j	c_j
0	2	0	1	–
1	3	1	1	1
2	4	1	2	2
3	5	2	3	3
4	6	3	5	5
5	7	5	8	8

Invariante y demostración

- **Invariante**: $i_j = j + 2$, $a_j = F_j$ y $b_j = F_{j+1}$ para toda $j \geq 0$.
- Lo demostraremos por inducción en j.
- Para $j = 0$ tenemos $i_0 = 2$, $a_0 = 0 = F_0$ y $b_0 = 1 = F_1$.
- Ahora suponga que $j \geq 0$ y que $i_j = j + 2$, $a_j = F_j$ y $b_j = F_{j+1}$. Entonces:
  - $i_{j+1} = i_j + 1 = (j + 2) + 1 = (j + 1) + 2$,
  - $a_{j+1} = b_j = F_{j+1}$ y
  - $b_{j+1} = c_{j+1} = a_j + b_j = F_j + F_{j+1} = F_{j+2}$.
- Que es lo que se quería demostrar.

Correctitud

- Ahora demostraremos que el algoritmo regresa F_n.
- Esto es claramente cierto para $n = 0$.
- Si $n > 0$ entonces se entra al ciclo.
- El algoritmo termina cuando $i_j = n + 1$.
- Por el invariante $i_j = j + 2$ esto ocurre cuando $j = n - 1$.
- Por el invariante $b_j = F_{j+1}$ el algoritmo regresa $b_{n-1} = F_n$.

Máximo iterativo

Función máximo(n)

1. $m \leftarrow A_1$, $i \leftarrow 2$.
2. Mientras $i \leq n$ haz:
   1. Si $A_i > m$ entonces $m \leftarrow A_i$.
   2. $i \leftarrow i + 1$.
3. Regresa m.

- Demostraremos que máximo(n) regresa máx$(A_1, \ldots, A_n)$.

Valores de las variables

- La variable i cumple que $i_0 = 2$ e $i_{j+1} = i_j + 1$.
- La variable m cumple que $m_0 = A_1$ y $m_{j+1} = \text{máx}(m_j, A_{i_j})$.

Ejemplo con $n = 4$ y $A = (3, 1, 4, 1)$

j	i_j	m_j
0	2	3
1	3	3
2	4	4
3	5	4

Invariante y demostración

- **Invariante**: $i_j = j + 2$ y $m_j = \text{máx}(A_1, \ldots, A_{j+1})$.
- Lo demostraremos por inducción en j.
- Para $j = 0$ tenemos $i_0 = 2$ y $m_0 = \text{máx}(A_1)$.
- Suponga que $j \geq 0$, $i_j = j + 2$ y $m_j = \text{máx}(A_1, \ldots, A_{j+1})$. Entonces $i_{j+1} = i_j + 1 = (j + 2) + 1 = (j + 1) + 2$ y

$$\begin{aligned} m_{j+1} &= \text{máx}(m_j, A_{i_j}) \\ &= \text{máx}(m_j, A_{j+2}) \\ &= \text{máx}(\text{máx}(A_1, \ldots, A_{j+1}), A_{j+2}) \\ &= \text{máx}(A_1, \ldots, A_{j+1}, A_{j+2}). \end{aligned}$$

- Que es lo que queríamos demostrar.

Correctitud

- Demostraremos que el algoritmo termina con m conteniendo el valor máximo en $A_1, \ldots, A_n$.
- El algoritmo termina cuando $i_j = n + 1$.
- Por el invariante $i_j = j + 2$ esto ocurre cuando $j = n - 1$.
- El valor final de m es $m_{n-1} = \text{máx}(A_1, \ldots, A_n)$.

Multiplicación iterativa

Función multiplica(y, z)

1. $x \leftarrow 0$.
2. Mientras $z > 0$ haz:
   1. Si z es impar entonces $x \leftarrow x + y$.
   2. $y \leftarrow 2y$.
   3. $z \leftarrow \lfloor z/2 \rfloor$.
3. Regresa x.

- Demostraremos que multiplica(y, z) regresa $x = yz$.

Valores de las variables

- La variable y cumple $y_0 = y$ y $y_{j+1} = 2y_j$.
- La variable z cumple $z_0 = z$ y $z_{j+1} = \lfloor z_j/2 \rfloor$.
- La variable x cumple $x_0 = 0$ y $x_{j+1} = x_j + y_j(z_j \bmod 2)$.

Ejemplo con $y = 6$ y $z = 5$

j	x_j	y_j	z_j
0	0	6	5
1	6	12	2
2	6	24	1
3	30	48	0

Invariante y demostración

- **Invariante**: $y_j z_j + x_j = y_0 z_0$ para toda $j \geq 0$.
- Lo demostraremos por inducción en j.
- Para $j = 0$ tenemos $y_0 z_0 + x_0 = y_0 z_0$.
- Supongamos que $j \geq 0$ y que $y_j z_j + x_j = y_0 z_0$. Entonces:

$$\begin{aligned} y_{j+1} z_{j+1} + x_{j+1} &= 2y_j \lfloor z_j/2 \rfloor + x_j + y_j(z_j \bmod 2) \\ &= y_j(2\lfloor z_j/2 \rfloor + (z_j \bmod 2)) + x_j \\ &= y_j z_j + x_j \\ &= y_0 z_0. \end{aligned}$$

- Que es lo que queríamos demostrar.
- **Nota**: usamos que $n = 2\lfloor n/2 \rfloor + (n \bmod 2)$ para todo entero $n \geq 0$.

Correctitud

- Demostraremos que el algoritmo termina con $x = yz$.
- A cada iteración del ciclo el valor de z disminuye.
- Por lo tanto existe un paso j tal que $z_j = 0$.
- En este momento el ciclo termina.
- Por el invariante $y_j z_j + x_j = y_0 z_0$ deducimos $x_j = y_0 z_0$.

Contenido

Implementación de algoritmos

- Considere los siguientes tres procesos:
  - Un programador toma un algoritmo y lo transcribe como un programa.
  - Un compilador toma un programa y lo vuelve un ejecutable.
  - Una computadora lo ejecuta con una entrada y obtiene una salida.
- En ellos aparece un factor constante que depende de varios factores:
  - La habilidad y efectividad del programador.
  - El lenguaje de programación y el compilador.
  - La velocidad y otros recursos del hardware.
- Es por eso que nos interesa medir el uso de recursos como una función del tamaño de la entrada **ignorando** los factores constantes.
- Usaremos **tres** notaciones que ignoran los factores constantes.

Notación O (Omicron)

- De manera informal diremos que $f(n)$ es $O(g(n))$ si f crece a lo mucho tan rápido como g.
- Dicho de otra forma, g es una cota **superior** para f.
- De manera formal, $f(n) \in O(g(n))$ si existen constantes $c \geq 0$ y $n_0 \geq 0$ tales que $f(n) \leq cg(n)$ para toda $n \geq n_0$.
- Generalmente demostraremos que $f(n) \in O(g(n))$ por inducción.

Primer ejemplo de notación O

- Demostraremos que $\log_2 n \in O(n)$.
- Para ello escogeremos $c = 1$ y $n_0 = 1$.
- Para $n = 1$ la desigualdad $\log_2 1 = 0 \leq 1$ es cierta.
- Ahora suponga que $n \geq 1$ y que $\log_2 n \leq n$. Entonces:

$$\begin{aligned} \log_2(n+1) &\leq \log_2 2n \\ &= \log_2 n + 1 \\ &\leq n + 1. \end{aligned}$$

- Que es lo que queríamos demostrar.

Segundo ejemplo de notación O

- Demostraremos que $2^{n+1} \in O(3^n/n)$.
- Para ello escogeremos $c = 1$ y $n_0 = 7$.
- Para $n = 7$ la desigualdad $2^8 = 256 \leq 312 \leq 3^7/7$ es cierta.
- Ahora suponga que $n \geq 7$ y que $2^{n+1} \leq 3^n/n$. Entonces:

$$\begin{aligned} 2^{n+2} &= 2 \cdot 2^{n+1} \\ &\leq 2 \cdot 3^n/n \\ &\leq \frac{3n}{n+1} \cdot \frac{3^n}{n} \\ &= 3^{n+1}/(n+1). \end{aligned}$$

- Que es lo que queríamos demostrar.
- **Nota**: Usamos que $n \geq 2$ si y sólo si $3n \geq 2n + 2$ si y sólo si $3n/(n+1) \geq 2$.

Notación Ω (Omega)

- De manera informal diremos que $f(n)$ es $\Omega(g(n))$ si f crece al menos tan rápido como g.
- Dicho de otra forma, g es una cota **inferior** para f.
- De manera formal, $f(n) \in \Omega(g(n))$ si existen constantes $c > 0$ y $n_0 \geq 0$ tales que $f(n) \geq cg(n)$ para toda $n \geq n_0$.
- **Nota**: en algunos lugares se define Ω de forma distinta.

Notación Θ (Zeta)

- De manera informal diremos que $f(n)$ es $\Theta(g(n))$ si f crece esencialmente tan rápido como g.
- Dicho de otra forma, g es un **múltiplo** de f.
- De manera formal, $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$.
- **Nota**: a la letra Θ se le llamaba **theta** hasta 1992.

Ejercicio

- Sea $f(n) = 3n^5 - 16n + 2$.
- ¿$f(n) \in O(n)$? ¿$f(n) \in O(n^5)$? ¿$f(n) \in O(n^{17})$?
- ¿$f(n) \in \Omega(n)$? ¿$f(n) \in \Omega(n^5)$? ¿$f(n) \in \Omega(n^{17})$?
- ¿$f(n) \in \Theta(n)$? ¿$f(n) \in \Theta(n^5)$? ¿$f(n) \in \Theta(n^{17})$?

Órdenes comunes

- Constante $O(1)$: determinar si un número es par.
- Logarítmico $O(\log n)$: búsqueda binaria.
- Polilogarítmico $O((\log n)^c)$: decidir si n es primo.
- Lineal $O(n)$: búsqueda lineal, suma de n bits.
- Linearítmico $O(n \log n)$: ordenamiento por mezcla.
- Cuadrático $O(n^2)$: ordenamiento de burbuja.
- Cúbico $O(n^3)$: algoritmo de Warshall.
- Polinomial $O(n^c)$: acoplamiento máximo.
- Exponencial $O(c^n)$: generación de cadenas c-arias.
- Factorial $O(n!)$: generación de permutaciones.

Primer teorema de la suma

Teorema

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$.

- $f_1(n) \leq c_1 g_1(n)$ para toda $n \geq n_1$.
- $f_2(n) \leq c_2 g_2(n)$ para toda $n \geq n_2$.
- Sea $n_0 = \max(n_1, n_2)$ y $c_0 = \max(c_1, c_2)$. Entonces para toda $n \geq n_0$

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c_0(g_1(n) + g_2(n)).$$

Segundo teorema de la suma

Teorema

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$.

- $f_1(n) \leq c_1 g_1(n)$ para toda $n \geq n_1$.
- $f_2(n) \leq c_2 g_2(n)$ para toda $n \geq n_2$.
- Sea $n_0 = \max(n_1, n_2)$ y $c_0 = c_1 + c_2$. Entonces para toda $n \geq n_0$:

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq (c_1 + c_2) \max(g_1(n), g_2(n)) \\ &= c_0 \max(g_1(n), g_2(n)). \end{aligned}$$

Teorema de la multiplicación

Teorema

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$.

- $f_1(n) \leq c_1 g_1(n)$ para toda $n \geq n_1$.
- $f_2(n) \leq c_2 g_2(n)$ para toda $n \geq n_2$.
- Sea $n_0 = \max(n_1, n_2)$ y $c_0 = c_1 \cdot c_2$. Entonces para toda $n \geq n_0$

$$f_1(n) \cdot f_2(n) \leq c_1 g_1(n) \cdot c_2 g_2(n) = c_0 g_1(n) g_2(n).$$

Tipos de análisis (tiempo de ejecución)

- **Peor caso**: se desea encontrar el máximo tiempo de ejecución usado por cualquiera de las posibles entradas de cierto tamaño.
- **Caso promedio**: se desea encontrar el tiempo de ejecución esperado dada una distribución de probabilidad (generalmente uniforme) de las entradas de cierto tamaño.
- **Probabilístico**: se desea encontrar el tiempo de ejecución esperado y la probabilidad de que esto ocurra.
- **Amortizado**: se desea encontrar el tiempo de ejecución promedio de una serie de ejecuciones.

Ejemplo

- Considere un algoritmo que toma tiempo $2n$ para todas las entradas de n bits excepto una para la que toma tiempo n^n.
- **Peor caso**: $\Theta(n^n)$.
- **Caso promedio**: $\Theta\left(\frac{n^n + (2^n - 1)(2n)}{2^n}\right) = \Theta\left(\frac{n^n}{2^n}\right)$.
- **Probabilístico**: $O(n)$ con probabilidad $1 - \frac{1}{2^n}$.
- **Amortizado**: Una secuencia de m ejecuciones con entradas **distintas** toma tiempo amortizado $O\left(\frac{n^n + (m-1)(2n)}{m}\right) = O\left(\frac{n^n}{m}\right)$.

Complejidad temporal

- Casi siempre haremos análisis de peor caso.
- ¿Cuánto tiempo toma ejecutar el algoritmo en el peor caso?
  - Asignación: $O(1)$.
  - Llamar a una función: $O(1)$.
  - Salir de una función: $O(1)$.
  - Decisión: el tiempo de la prueba más el peor tiempo de las ramas.
  - Ciclo: la suma de los tiempos de cada una de las iteraciones.
- Para los algoritmos iterativos será suficiente usar esto y los teoremas de la suma y del producto.
- Para los algoritmos recursivos será ligeramente distinto.

Tiempo de ejecución de la multiplicación

Función multiplica(y, z)

1. $x \leftarrow 0$.
2. Mientras $z > 0$ haz:
   1. Si z es impar entonces $x \leftarrow x + y$.
   2. $y \leftarrow 2y$.
   3. $z \leftarrow \lfloor z/2 \rfloor$.
3. Regresa x.

- Supongamos que y, z tienen n bits.
- La llamada, la asignación y el regreso cuestan $O(1)$ cada una.
- La decisión y las asignaciones dentro del ciclo cuestan $O(1)$ cada una.
- El ciclo se ejecuta un máximo de n veces.
- Por lo tanto todo el proceso toma tiempo $O(n)$.

Tiempo de ejecución del algoritmo de burbuja

Función burbuja(A, n)

1. Para $i \leftarrow 1$ a $n - 1$ haz:
   1. Para $j \leftarrow 1$ a $n - i$ haz:
      1. Si $A_j > A_{j+1}$ entonces intercambia A_j con A_{j+1}.

- La llamada, la decisión y el regreso cuestan $O(1)$ cada una.
- El ciclo interno se ejecuta $n - i$ veces y cuesta $O(n - i)$.
- El ciclo externo cuesta

$$O\left(\sum_{i=1}^{n-1}(n - i)\right) = O\left(\sum_{i=1}^{n-1} i\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2).$$

- También se puede probar que toma tiempo $\Omega(n^2)$ y por lo tanto tiempo $\Theta(n^2)$.

Facilitando el análisis

- En principio, se podría calcular exactamente el tiempo o el número de pasos necesarios para ejecutar un algoritmo.
- Pero esto es generalmente innecesario.
- En lugar de esto, identifica la **operación fundamental** que se usa en el algoritmo.
- Generalmente el tiempo de ejecución es un múltiplo constante de la cantidad de operaciones fundamentales.
- Así que no hay necesidad de hacer un análisis línea por línea.
- Sólo es necesario calcular exactamente la cantidad de operaciones fundamentales.

Ejemplo

- En el ejemplo del algoritmo de la burbuja, la operación fundamental es la **comparación**.
- El tiempo de ejecución será un factor constante del número total de comparaciones.
- La decisión hace 1 comparación.
- El ciclo interno hace $n - i$ comparaciones.
- El ciclo externo hace $\sum_{i=1}^{n-1}(n-i)$ comparaciones.
- En total se hacen $n(n-1)/2$ comparaciones.
- Esto es $O(n^2)$.

Detalles a considerar

- ¿Qué significa que el algoritmo de la multiplicación sea $O(n)$?
- Hicimos una suposición sobre el modelo de cómputo:
  - Supusimos que cada suma de n bits toma tiempo $O(1)$.
  - Esto sólo es cierto en el **modelo de palabras**.
  - En el **modelo de bits** la misma suma toma tiempo $O(n)$.
- No decir toda la verdad también es mentir:
  - El algoritmo de la multiplicación también toma tiempo $O(n^2)$.
- Hay que tener una idea de cuánto vale el factor constante.
  - No es lo mismo hacer $2n$ operaciones que $2^{2^{2^{2^2}}}$ n operaciones.
  - Un factor constante muy grande puede volver impráctico un algoritmo.

Algoritmos y problemas

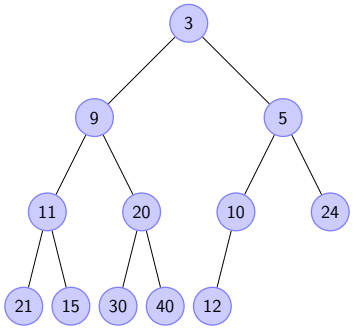
- Las notaciones O, Ω y Θ significan cosas distintas cuando se le aplican a algoritmos o a problemas.
- El algoritmo de burbuja toma tiempo $O(n^2)$:
  - ¿Pero es lo mejor que puedo decir? Tal vez soy demasiado flojo como para descubrirlo o tal vez no se sabe.
- El algoritmo de burbuja toma tiempo $\Theta(n^2)$:
  - Esto es lo mejor que se puede decir.
- El problema de ordenamiento toma tiempo $O(n \log n)$:
  - Existe un algoritmo que puede ordenar en tiempo $O(n \log n)$, pero no sé si existe un algoritmo más rápido.
- El problema de ordenamiento toma tiempo $\Theta(n \log n)$:
  - Existe un algoritmo que puede ordenar en tiempo $O(n \log n)$ y ningún otro algoritmo es más rápido.

Contenido

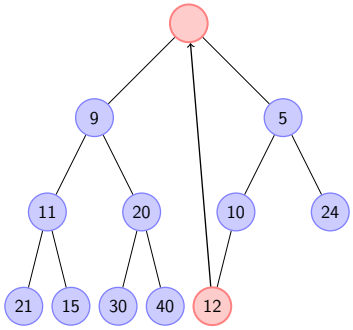
Montículos

- Una **cola de prioridad** tiene las siguientes operaciones:
  - Insertar un elemento.
  - Borrar y regresar el elemento más pequeño.
- El **montículo** es una implementación popular.
- Un montículo es un árbol binario con datos en los nodos que satisface las siguientes propiedades:
  - **Balance**: es un árbol binario completo excepto porque pueden faltar algunos nodos del lado derecho del último nivel.
  - **Estructura**: el valor de cada nodo padre es menor o igual que el de sus nodos hijos (y por lo tanto que el de todos sus descendientes).

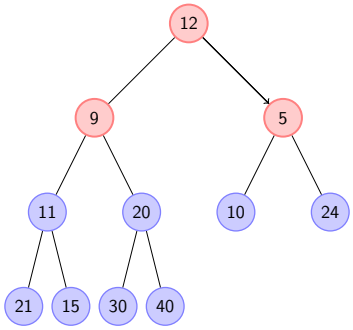
Ejemplo de un montículo



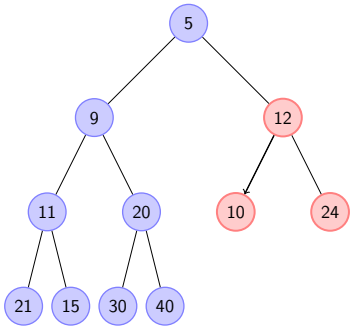
Ejemplo de borrado 1



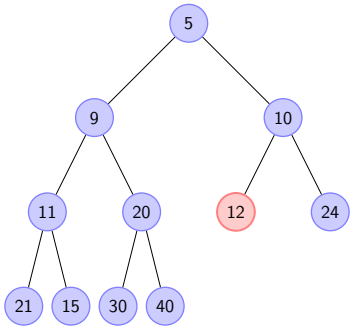
Ejemplo de borrado 2



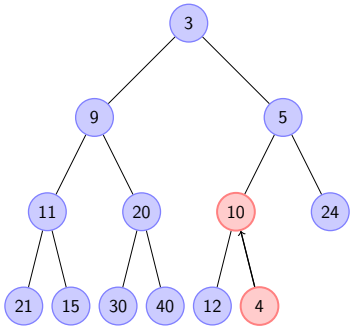
Ejemplo de borrado 3



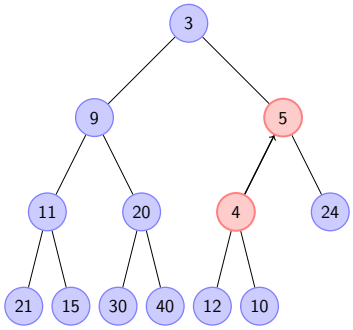
Ejemplo de borrado 4



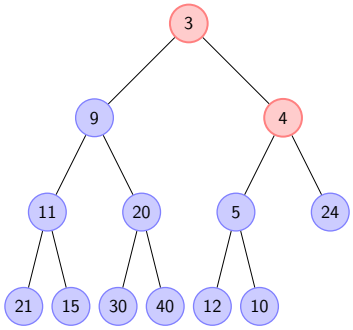
Ejemplo de inserción 1



Ejemplo de inserción 2



Ejemplo de inserción 3



Implementación de un montículo

- Un montículo con n nodos usa un arreglo $A_1, \ldots, A_n$.
- La raíz está almacenada en A_1.
- El hijo izquierdo del nodo A_i está en A_{2i}.
- El hijo derecho del nodo A_i está en A_{2i+1}.

Ejemplo de arreglo

$$A = [3, 9, 5, 11, 20, 10, 24, 21, 15, 30, 40, 12].$$

Análisis de las operaciones en un montículo

- Sea $\ell(n)$ el número de niveles de un montículo con n nodos.
- Borrado del mínimo:
  - Borrar la raíz toma tiempo $O(1)$.
  - Reemplazar la raíz toma tiempo $O(1)$.
  - Se hacen $O(\ell(n))$ intercambios.
- Inserción de un elemento:
  - Colocar la hoja toma tiempo $O(1)$.
  - Se hacen $O(\ell(n))$ intercambios.

¿Cuánto vale $\ell(n)$?

- Un árbol completo con k niveles tiene exactamente $2^k - 1$ nodos.
- Por lo tanto, un montículo con k niveles tiene al menos 2^{k-1} nodos y menos de 2^k nodos. En otras palabras:

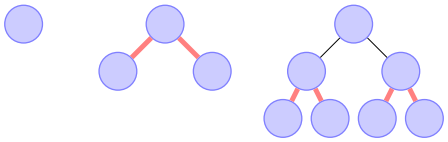
$$\begin{array}{rcl} 2^{k-1} \leq & n & < 2^k \\ k-1 \leq & \log_2 n & < k \\ k-1 \leq & \lfloor \log_2 n \rfloor & \leq k-1. \end{array}$$

- Por lo tanto $\ell(n) = k = \lfloor \log_2 n \rfloor + 1$.
- Esto significa que el borrado e inserción en un montículo toman tiempo $O(\log n)$ cada uno.

Ordenamiento por montículo

- Para ordenar n números:
  1. Insertar los n números en un montículo vacío.
  2. Borrar el mínimo n veces.
- Cada inserción cuesta $O(\log n)$. Por lo tanto la primera línea cuesta $O(n \log n)$.
- Cada borrado cuesta $O(\log n)$. Por lo tanto la segunda línea cuesta $O(n \log n)$.
- Por lo tanto el ordenamiento por montículo cuesta $O(n \log n)$ en el peor de los casos.

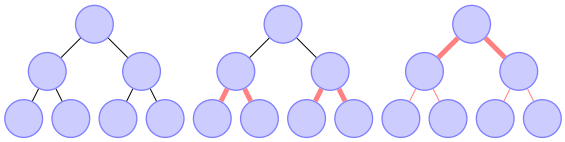
Construcción del montículo de arriba para abajo



- El costo de una inserción es proporcional a la altura.
- El número de inserciones es $\leq \sum_{j=0}^{\ell(n)-1} j2^j \in \Theta(n \log n)$.
- Se puede probar por inducción que

$$\sum_{j=1}^{k} j2^j = (k-1)2^{k+1} + 2 \in \Theta(k2^k).$$

Construcción del montículo de abajo para arriba



- El costo de una inserción es proporcional a la altura.
- El número de inserciones es

$$\leq \sum_{i=1}^{\ell(n)} (i-1)2^{\ell(n)-i} < 2^{\ell(n)} \sum_{i=1}^{\ell(n)} \frac{i}{2^i} \in O\left(n \sum_{i=1}^{\ell(n)} \frac{i}{2^i}\right).$$

- ¿Cuánto vale el último término?

Cálculo del último término

- No es difícil ver que el último término es ≤ 2:

$$\begin{aligned} \sum_{i=1}^{k} \frac{i}{2^i} &= \frac{1}{2^k} \sum_{i=1}^{k} i2^{k-i} \\ &= \frac{1}{2^k} \sum_{i=0}^{k-1} (k-i)2^i \\ &= \frac{k}{2^k} \sum_{i=0}^{k-1} 2^i - \frac{1}{2^k} \sum_{i=0}^{k-1} i2^i \\ &= \frac{k(2^k-1)}{2^k} - \frac{(k-2)2^k + 2}{2^k} \\ &= 2 - \frac{k+2}{2^k}. \end{aligned}$$

- Por lo tanto construir el montículo toma tiempo $O(n)$.

Contenido

Derivación de relaciones de recurrencia

- Para derivar una relación de recurrencia para el tiempo de ejecución $T(n)$ de un algoritmo recursivo:
  1. Decida qué será el tamaño n del problema.
  2. Vea cuáles valores de n se usan como base para la recursión. Puede ser un solo valor n_0 o varios.
  3. Calcule $T(n_0)$. Normalmente bastará decir que **es constante**.
  4. El valor general de $T(n)$ será normalmente la suma de varios valores de $T(m)$ (las llamadas recursivas) más el trabajo adicional hecho.

Forma general de una relación de recurrencia

- En general

$$T(n) = \begin{cases} c & \text{si } n = n_0 \\ aT(f(n)) + g(n) & \text{en otro caso} \end{cases}$$

donde:

1. c es el tiempo de ejecución del caso base,
2. n_0 es el tamaño del caso base,
3. a es la cantidad de llamadas recursivas,
4. $f(n)$ es el tamaño de las llamadas recursivas y
5. $g(n)$ es el tiempo de ejecución de todo el procesamiento no incluido en las llamadas recursivas.

- ¿Cómo cambia esto si las llamadas recursivas son de diferentes tamaños?

Solución de relaciones de recurrencia

- Una técnica se llama **sustitución repetida**.
- Dada una relación de recurrencia para $T(n)$:
  1. Sustituya algunas veces hasta que vea un patrón.
  2. Escriba una fórmula en términos de n y la cantidad i de sustituciones.
  3. Escoja el valor de i para que todas las referencias a $T(i)$ sean referencias al caso base.
  4. Calcule la suma resultante.
- No funcionará siempre, pero en la práctica funciona frecuentemente.

Contenido

Recurrencia para multiplicación de enteros

Función multiplica(y, z)

1. Si $z = 0$ entonces regresa 0.
2. Si z es impar
   1. entonces regresa multiplica$(2y, \lfloor z/2 \rfloor) + y$,
   2. si no regresa multiplica$(2y, \lfloor z/2 \rfloor)$.

- Sea $T(n)$ el tiempo de ejecución de multiplica(y, z) cuando z es entero de n bits. Entonces

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

para algunas constantes c y d.

Sustitución repetida

- Suponga que $n > 1$. Entonces:

$$\begin{aligned} T(n) &= T(n-1) + d \\ &= (T(n-2) + d) + d \\ &= T(n-2) + 2d \\ &= (T(n-3) + d) + 2d \\ &= T(n-3) + 3d. \end{aligned}$$

- Hay un patrón: **parece** que después de i sustituciones

$$T(n) = T(n-i) + id.$$

- Si escogemos $i = n - 1$ entonces

$$T(n) = T(n-(n-1)) + (n-1)d = T(1) + (n-1)d = c + (n-1)d.$$

- Pero esto **no es una demostración**. ¿Por qué?

Demostración por inducción

- Demostraremos que $T(n) = c + (n-1)d$ por inducción en n.
- Para $n = 1$ tenemos que $T(1) = c + (1-1)d = c$, lo cual es cierto.
- Ahora supongamos que $n \geq 1$ y que $T(n) = c + (n-1)d$. Entonces

$$\begin{aligned} T(n+1) &= T(n) + d \\ &= c + (n-1)d + d \\ &= c + nd. \end{aligned}$$

- Que es lo que queríamos demostrar.

Contenido

Ordenamiento por mezcla

Función ordenamezcla(L, n)

1. Si $n \leq 1$ entonces regresa L.
2. Parte L en dos listas L_1 y L_2 del **mismo** tamaño ($n_1 = n_2 = n/2$).
3. Regresa mezcla(ordenamezcla(L_1, n_1), ordenamezcla(L_2, n_2)).

- Vamos a hacer dos suposiciones:
  - Que n es una potencia de 2.
  - Que la función mezcla puede mezclar dos listas ordenadas con un total de n elementos en tiempo $O(n)$.
- Sea $T(n)$ el tiempo de ejecución de ordenamezcla(L, n). Entonces

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

para algunas constantes c y d.

Sustitución repetida

- Suponga que $n > 1$. Entonces:

$$\begin{aligned} T(n) &= 2T(n/2) + dn \\ &= 2(2T(n/4) + dn/2) + dn \\ &= 4T(n/4) + 2dn \\ &= 4(2T(n/8) + dn/4) + 2dn \\ &= 8T(n/8) + 3dn. \end{aligned}$$

- Aparece el patrón $T(n) = 2^i T(n/2^i) + idn$.
- Tomando $i = \log_2 n$ se tiene que

$$T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + dn \log_2 n = dn \log_2 n + cn.$$

- Por lo tanto $T(n) \in O(n \log n)$.
- De nuevo, **esto no es una demostración**.

Contenido

Un teorema general

Teorema

Si n es una potencia de c entonces la solución a la recurrencia

$$T(n) = \begin{cases} d & \text{si } n = 1 \\ aT(n/c) + bn & \text{si } n > 1 \end{cases}$$

está dada por:

- $T(n) \in O(n)$ si $a < c$,
- $T(n) \in O(n \log n)$ si $a = c$ o
- $T(n) \in O(n^{\log_c a})$ si $a > c$.

Inicio de la prueba

- Si $n = c^k$ obtenemos por sustitución

$$\begin{aligned} T(c^k) &= aT(c^{k-1}) + bc^k \\ &= a[aT(c^{k-2}) + bc^{k-1}] + bc^k \\ &= a^2 T(c^{k-2}) + bc^k \left(\frac{a}{c} + 1\right) \\ &= a^2[aT(c^{k-3}) + bc^{k-2}] + bc^k \left(\frac{a}{c} + 1\right) \\ &= a^3 T(c^{k-3}) + bc^k \left(\frac{a^2}{c^2} + \frac{a}{c} + 1\right). \end{aligned}$$

- Aparentemente, después de i sustituciones obtenemos:

$$T(c^k) = a^i T(c^{k-i}) + bc^k \left(\frac{a^{i-1}}{c^{i-1}} + \cdots + \frac{a^2}{c^2} + \frac{a}{c} + 1\right).$$

Prueba de la sustitución iterada

- Demostremos eso por inducción en i.
- El enunciado es trivial para $i = 0$ (dice que $T(c^k) = T(c^k)$).
- Supongamos que es cierto para i sustituciones.
- Entonces, en la siguiente sustitución:

$$\begin{aligned} T(c^k) &= a^i T(c^{k-i}) + bc^k \left(\frac{a^{i-1}}{c^{i-1}} + \cdots + \frac{a^2}{c^2} + \frac{a}{c} + 1\right) \\ &= a^i[aT(c^{k-i-1}) + bc^{k-i}] + bc^k \left(\frac{a^{i-1}}{c^{i-1}} + \cdots + \frac{a^2}{c^2} + \frac{a}{c} + 1\right) \\ &= a^{i+1} T(c^{k-(i+1)}) + bc^k \left(\frac{a^i}{c^i} + \cdots + \frac{a^2}{c^2} + \frac{a}{c} + 1\right) \end{aligned}$$

que es lo que queríamos demostrar.

Simplificación del lado derecho

- Así, después de k sustituciones tenemos que:

$$\begin{aligned} T(c^k) &= a^k T(1) + bc^k \left(\frac{a^{k-1}}{c^{k-1}} + \cdots + \frac{a^2}{c^2} + \frac{a}{c} + 1\right) \\ &= da^k + bc^k \sum_{i=0}^{k-1} (a/c)^i. \end{aligned}$$

- En otras palabras, tenemos que:

$$T(n) = dn^{\log_c a} + bn \sum_{i=0}^{\log_c n - 1} (a/c)^i.$$

- Sólo nos falta calcular la suma del lado derecho.

Progresiones geométricas

- Defina $S_k = \sum_{i=0}^{k-1} \alpha^i$.
- Si $\alpha > 1$ entonces observe que

$$\alpha S_k - S_k = \sum_{i=1}^{k} \alpha^i - \sum_{i=0}^{k-1} \alpha^i = \alpha^k - 1$$

y por lo tanto $S_k = \frac{\alpha^k - 1}{\alpha - 1}$.

- Si $0 < \alpha < 1$ entonces defina $S = \sum_{i=0}^{\infty} \alpha^i$ y observe que

$$S - \alpha S = \sum_{i=0}^{\infty} \alpha^i - \sum_{i=1}^{\infty} \alpha^i = 1$$

y por lo tanto $S_k < S = \frac{1}{1-\alpha}$.

Primer caso

- Si $a < c$ entonces

$$\begin{aligned} T(n) &= dn^{\log_c a} + bn \sum_{i=0}^{\log_c n - 1} (a/c)^i \\ &< dn^{\log_c a} + bn \sum_{i=0}^{\infty} (a/c)^i \\ &= dn^{\log_c a} + \frac{bn}{1 - \frac{a}{c}}. \end{aligned}$$

- Como $\log_c a < 1$ entonces podemos concluir que

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

Segundo caso

- Si $a = c$ entonces

$$\begin{aligned} T(n) &= dn^{\log_c a} + bn \sum_{i=0}^{\log_c n - 1} (a/c)^i \\ &= dn + bn \log_c n. \end{aligned}$$

- Entonces podemos concluir que

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

Tercer caso

- Si $a > c$ entonces

$$\begin{aligned} T(n) &= dn^{\log_c a} + bn \sum_{i=0}^{\log_c n - 1} (a/c)^i \\ &= dn^{\log_c a} + bn \frac{(\frac{a}{c})^{\log_c n} - 1}{\frac{a}{c} - 1} \\ &= dn^{\log_c a} + b \frac{n^{\log_c a} - n}{\frac{a}{c} - 1}. \end{aligned}$$

- Entonces podemos concluir que

$$T(n) \in O(n^{\log_c a}).$$

Si n no es potencia de c

- Generalmente n no será una potencia exacta de c.
- ¿Qué se puede decir en esos casos?
- Esencialmente lo mismo.
- Suponga que $c^{k-1} \leq n \leq c^k$ entonces

$$T(c^{k-1}) \leq T(n) \leq T(c^k).$$

- En los tres casos se puede ver que $O(T(c^{k-1})) = O(T(c^k))$ y por lo tanto $T(n)$ es del mismo orden.

Contenido

Divide y vencerás

- Una de las técnicas más comunes de diseño de algoritmos es la llamada **divide y vencerás**.
- Para resolver un problema usando esta técnica se debe:
  - Dividir el problema en subproblemas más pequeños.
  - Resolver los subproblemas más pequeños.
  - Combinar sus soluciones para resolver el problema grande.
- **Ejemplos:** búsqueda binaria, ordenamiento por mezcla, etc.

Contenido

Máximo y mínimo por separado

- Considere el arreglo de enteros $a = (a_1, a_2, \ldots, a_n)$.
- Se puede encontrar el máximo de a con $n - 1$ comparaciones.
- Se puede encontrar el mínimo de a con $n - 1$ comparaciones.
- Por lo tanto se puede encontrar el máximo y el mínimo del arreglo a con $2n - 2 \in O(n)$ comparaciones.
- ¿Se podrán encontrar con menos de $O(n)$ comparaciones?
- ¿Se podrán encontrar con menos de $2n - 2$ comparaciones?

Máximo y mínimo simultaneamente

- Divida el arreglo a la mitad.
- Encuentre recursivamente el máximo y el mínimo en cada mitad.
- Regrese el máximo de los máximos y el mínimo de los mínimos.

Función maxmin(i, j)

1. Si $j - i \leq 1$ entonces
   1. regresa máx(a_i, a_j) y mín(a_i, a_j).
2. Si no entonces
   1. haz $(b, d) \leftarrow$ maxmin$(i, \lfloor \frac{1}{2}(i + j) \rfloor)$
   2. haz $(c, e) \leftarrow$ maxmin$(\lfloor \frac{1}{2}(i + j) \rfloor + 1, j)$
   3. regresa máx(b, c) y mín(d, e).

Análisis de correctitud

- Por inducción en $n = j - i + 1$.
- El algoritmo claramente funciona para $n \leq 2$.
- Supongamos que el algoritmo funciona para $j - i + 1 < n$.
- ¿Cuánto valen los parámetros de las llamadas recursivas?
- Depende de la paridad de $j - i + 1$. Por ejemplo, si $j - i + 1$ es par

$$\lfloor \frac{1}{2}(i + j) \rfloor = \frac{1}{2}(i + j - 1)$$

y por lo tanto

$$\lfloor \frac{1}{2}(i + j) \rfloor - i + 1 = \frac{1}{2}(j - i + 1) < j - i + 1.$$

- En todos los casos el **rango** disminuye, eventualmente se llega al caso base y el algoritmo funciona.

Análisis de tiempo de ejecución

- Sea $T(n)$ la cantidad de comparaciones hecha por maxmin(i,j) cuando $n = j - i + 1$.
- Supongamos que $n = 2^k$ para algún entero k.
- En este caso, cada uno de los subarreglos mide $n/2 = 2^{k-1}$ y por lo tanto tenemos que

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

Sustitución

- Sustituyendo obtenemos que

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= \dots \\ &= 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j \text{ para toda } 1 \le j < k \\ &= 2^{k-1} T(2) + \sum_{j=1}^{k-1} 2^j \\ &= 2^{k-1} + (2^k - 2) \\ &= \frac{3}{2}n - 2. \end{aligned}$$

Contenido

2. Divide y vencerás
   - Análisis de máximo y mínimo
   - Análisis de multiplicación de enteros grandes
   - Análisis de multiplicación de matrices grandes
   - Análisis de Quicksort
   - Análisis de selección lineal
   - Análisis de búsqueda binaria

Multiplicación de enteros

- Conocemos un algoritmo para multiplicar dos enteros de n bits.
- Cuando analizamos ese algoritmo descubrimos que hace $O(n)$ sumas.
- Sin embargo, cada suma hace $O(n)$ operaciones de bits.
- Por lo tanto, ese algoritmo hace $O(n^2)$ operaciones de bits.
- ¿Se podrá multiplicar dos enteros haciendo menos multiplicaciones?

Primera idea

- Suponga que y y z son dos enteros de n bits.
- Suponga también que n es una potencia de 2.
- Divida y y z en dos mitades (cada una con $n/2$ bits)

$$\begin{aligned} y &= a2^{n/2} + b \\ z &= c2^{n/2} + d. \end{aligned}$$

- Por lo tanto

$$\begin{aligned} yz &= (a2^{n/2} + b)(c2^{n/2} + d) \\ &= ac2^n + (ad + bc)2^{n/2} + bd. \end{aligned}$$

Primer análisis

- Esta idea nos sirve para calcular yz con 4 multiplicaciones de números de $n/2$ bits y algunas sumas y corrimientos.
- El tiempo de ejecución queda dado por

$$T(n) = \begin{cases} \alpha & \text{si } n = 1 \\ 4T(n/2) + \beta n & \text{si } n > 1 \end{cases}$$

- El teorema general nos dice que $T(n) \in O(n^2)$.
- Esto **no** mejora el algoritmo anterior.

Segunda idea

- Sin embargo, yz se puede calcular de esta otra forma:
  1. Sea $u = (a + b)(c + d)$.
  2. Sea $v = ac$.
  3. Sea $w = bd$.
  4. Sea $x = v2^n + (u - v - w)2^{n/2} + w$.
- Verifiquemos que esto funciona:

$$\begin{aligned} x &= v2^n + (u - v - w)2^{n/2} + w \\ &= ac2^n + ((a + b)(c + d) - ac - bd)2^{n/2} + bd \\ &= ac2^n + (ad + bc)2^{n/2} + bd \\ &= yz. \end{aligned}$$

Segundo análisis

- Esta nueva idea nos sirve para calcular yz con 3 multiplicaciones de números de $n/2$ bits y algunas sumas y corrimientos.
- El tiempo de ejecución queda dado por

$$T(n) = \begin{cases} \gamma & \text{si } n = 1 \\ 3T(n/2) + \delta n & \text{si } n > 1 \end{cases}$$

- El teorema general nos dice que $T(n) \in O(n^{\log_2 3}) \approx O(n^{1.585})$.
- Esto **sí** mejora el algoritmo anterior.

Algoritmos de multiplicación

- ¿Qué tan rápido se puede multiplicar dos números de n bits?
- El algoritmo obvio tarda $O(n^2)$.
- El algoritmo de divide y vencerás tarda $O(n^{\log_2 3})$ (descubierto en 1960 por Karatsuba).
- Entre 1963 y 1966 Toom y Cook descubrieron un algoritmo que tarda $O(n^{\log_3 5})$.
- En 1971 Schönhage y Strassen descubrieron un algoritmo que tarda $O(n \log n \log\log n)$ usando la transformada rápida de Fourier.
- En 2007 Führer descubrió un algoritmo que tarda $O(2^{O(\log^* n)} n \log n)$.
- Se cree que ningún algoritmo puede tardar menos que $\Omega(n \log n)$.

Contenido

Multiplicación de matrices

- Sean Y y Z dos matrices de $n \times n$.
- El algoritmo común y corriente para calcular el producto $X = YZ$ hace $O(n^3)$ operaciones (multiplicaciones y sumas).
- Si cada operación se tarda $O(1)$ entonces este algoritmo se tarda $O(n^3)$.
- ¿Podemos hacerlo más rápido?

División de las matrices

- Dividamos cada una de X, Y y Z en cuatro matrices:

$$X = \begin{pmatrix} I & J \\ K & L \end{pmatrix} Y = \begin{pmatrix} A & B \\ C & D \end{pmatrix} Z = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

- Entonces tenemos que

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH. \end{aligned}$$

Análisis

- Sea $T(n)$ el tiempo que nos toma multiplicar dos matrices de $n \times n$ con esta idea.
- Podemos suponer que $n = 2^k$.
- Entonces

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 8T(n/2) + dn^2 & \text{si } n > 1 \end{cases}$$

Sustitución

- Sustituyendo obtenemos

$$\begin{aligned} T(n) &= 8T(n/2) + dn^2 \\ &= 8(8T(n/4) + d(n/2)^2) + dn^2 \\ &= 8^2 T(n/2^2) + 2^1 dn^2 + 2^0 dn^2 \\ &= 8^i T(n/2^i) + dn^2 \sum_{j=0}^{i-1} 2^j \\ &= 8^k T(1) + dn^2(2^k - 1) \\ &= cn^3 + dn^2(n-1). \end{aligned}$$

- Por lo tanto $T(n) \in O(n^3)$, lo cual no es una mejora.

Algoritmo de Strassen

1. $M_1 = (A + C)(E + F)$.
2. $M_2 = (B + D)(G + H)$.
3. $M_3 = (A - D)(E + H)$.
4. $M_4 = A(F - H)$.
5. $M_5 = (C + D)E$.
6. $M_6 = (A + B)H$.
7. $M_7 = D(G - E)$.
8. $I = M_2 + M_3 - M_6 - M_7$.
9. $J = M_4 + M_6$.
10. $K = M_5 + M_7$.
11. $L = M_1 - M_3 - M_4 - M_5$.

Análisis de correctitud

- Se puede verificar que los cálculos anteriores funcionan.
- Por ejemplo:

$$\begin{aligned} J &= M_4 + M_6 \\ &= A(F - H) + (A + B)H \\ &= AF - AH + AH + BH \\ &= AF + BH. \end{aligned}$$

Análisis

- Sea $T(n)$ el tiempo que nos toma multiplicar dos matrices de $n \times n$ con el algoritmo de Strassen.
- Podemos suponer que $n = 2^k$.
- Entonces

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 7T(n/2) + dn^2 & \text{si } n > 1 \end{cases}$$

Sustitución

- Sustituyendo obtenemos

$$\begin{aligned} T(n) &= 7T(n/2) + dn^2 \\ &= 7(7T(n/4) + d(n/2)^2) + dn^2 \\ &= 7^2T(n/2^2) + 7^1dn^2/4^1 + 7^0dn^2/4^0 \\ &= 7^iT(n/2^i) + dn^2\sum_{j=0}^{i-1}(7/4)^j \\ &= 7^kT(1) + dn^2\frac{(7/4)^k - 1}{7/4 - 1} \\ &= cn^{\log_2 7} + \frac{4}{3}d(n^{\log_2 7} - n^2). \end{aligned}$$

- Por lo tanto $T(n) \in O(n^{\log_2 7}) \approx O(n^{2.807})$, lo cual **sí** es una mejora.

Algoritmos de multiplicación de matrices

- ¿Qué tan rápido se puede multiplicar dos matrices de $n \times n$?
- El algoritmo obvio tarda $O(n^3)$.
- El algoritmo de Strassen tarda $O(n^{\log_2 7})$ (descubierto en 1971).
- En 1990 Coppersmith y Winograd descubrieron un algoritmo que tarda $O(n^{2.376})$.
- Ningún algoritmo puede tardar menos de $O(n^2)$.
- Muchos investigadores creen que **sí** existen algoritmos que tardan $O(n^2)$.

Contenido

Algoritmo genérico

- Sea S una lista de n enteros **distintos**.

Función quicksort(S)

1. Si $|S| \leq 1$ entonces
   1. Regresa S
2. Si no entonces
   1. Escoge un elemento a de S **¿cómo?**
   2. Sean $S_<$, $S_=$ y $S_>$ los elementos de S que son $< a$, $= a$ y $> a$ respectivamente
   3. Regresa (quicksort($S_<$), $S_=$, quicksort($S_>$))

- El elemento a se llama **pivote**.

Análisis del caso promedio

- Sea $T(n)$ la cantidad **promedio** de comparaciones usadas por quicksort cuando ordena n enteros distintos.
- Claramente $T(0) = T(1) = 0$.
- Suponga que a es el elemento i más pequeño de S. Entonces
  - $|S_<| = i - 1$,
  - $|S_=| = 1$ y
  - $|S_>| = n - i$.
- Observe que S se puede dividir en $S_<$, $S_=$ y $S_>$ usando $n - 1$ comparaciones.

Ecuación recurrente

- Las llamadas recursivas usan $T(i-1)$ y $T(n-i)$ comparaciones.
- El valor de i puede ser cualquiera entre 1 y n.
- La **probabilidad** de cada valor es la misma.
- Por lo tanto, para $n \geq 2$ tenemos

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)).$$

Simplificación

- Observe que

$$\sum_{i=1}^{n}(T(i-1) + T(n-i)) = \sum_{i=1}^{n}T(i-1) + \sum_{i=1}^{n}T(n-i)$$

- Equivalentemente

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

- Por lo tanto

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

- ¿Cómo resolvemos esta ecuación? ¡No por sustitución!

Reescritura de la ecuación

- Multiplicando la ecuación por n tenemos que
$$nT(n) = n^2 - n + 2\sum_{i=0}^{n-1} T(i).$$
- Reescribiendo esto mismo para $n-1$ tenemos que
$$(n-1)T(n-1) = (n-1)^2 - (n-1) + 2\sum_{i=0}^{n-2} T(i).$$
- Restando estas dos ecuaciones obtenemos
$$nT(n) - (n-1)T(n-1) = 2n - 2 + 2T(n-1)$$
es decir $nT(n) = (n+1)T(n-1) + 2(n-1)$.
- Dividiendo entre $n(n+1)$ tenemos que
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

Cambio de variable

- Definiendo $S(n) = \frac{1}{n+1}T(n)$ tenemos que $S(0) = S(1) = 0$ y
$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)}.$$
- Por lo tanto $S(n) = 0$ para $n \leq 1$ y $S(n) \leq S(n-1) + \frac{2}{n}$ para $n \geq 2$.
- Ahora sí, por sustitución repetida tenemos que
$$\begin{aligned} S(n) &\leq S(n-1) + \frac{2}{n} \\ &\leq S(n-2) + \frac{2}{n-1} + \frac{2}{n} \\ &\leq S(n-i) + 2\sum_{j=n-i+1}^{n} \frac{1}{j}. \end{aligned}$$

Solución

- Por lo tanto
$$S(n) \leq S(1) + 2\sum_{j=2}^{n} \frac{1}{j} \leq 2\int_1^n \frac{dx}{x} = 2\ln n.$$
- Y finalmente
$$\begin{aligned} T(n) &= (n+1)S(n) \\ &\leq 2(n+1)\ln n \\ &= 2(n+1)\log_2 n/\log_2 e \\ &\approx 1.386 n \log_2 n. \end{aligned}$$
- Es decir, quicksort hace $O(n \log n)$ comparaciones en **promedio**.

Análisis del peor caso

- Quicksort es **ligeramente** mejor que ordenamiento por mezcla en el caso promedio.
- ¿Pero qué pasa en el **peor** caso?
- Por ejemplo, si $i = 1$ a cada paso entonces $T(1) = 0$ y $T(n) = T(n-1) + n - 1$ para $n \geq 2$.
- En este caso se puede demostrar que $T(n) \in \Theta(n^2)$.
- Así que quicksort es **mucho peor** que ordenamiento por mezcla en el peor caso.

Análisis del mejor caso

- Por ejemplo, si $i = n/2$ a cada paso entonces $T(1) = 0$ y para $n > 1$
$$T(n) = 2T(n/2) + n - 1.$$
- En este caso es fácil demostrar que $T(n) = n \log n + O(n)$.
- Por lo tanto, quicksort hace sólo 39 % más comparaciones en el caso promedio que en el mejor caso.

Análisis de correctitud

- La prueba es larga y tediosa, pero la idea principal es que se debe demostrar que a cada paso se reduce el tamaño de la lista en las llamadas recursivas.
- Este análisis no depende de qué se escoja como pivote. Sin embargo, ¿qué se debería tomar como pivote?
  - Una versión simple toma al primer o al último elemento de la lista.
  - Otra versión toma al elemento justo al **centro** de la lista.
  - Otra versión toma un elemento **seudoaleatorio** de la lista.
  - Otra versión toma un elemento cercano a la **mediana**.
- Cada una de estas versiones tiene sus ventajas y desventajas.

Complejidad del ordenamiento

- ¿Qué tan rápido se pueden ordenar n elementos?
- Eso depende de qué operaciones se permita hacer con ellos.
- Quicksort y ordenamiento por mezcla hacen **comparaciones**.
- Demostraremos que los algoritmos de ordenamiento basados en comparaciones deben hacer al menos $\Omega(n \log n)$ comparaciones.
- Existen algoritmos sencillos que hacen $O(n)$ operaciones (de otro tipo) para ordenar n elementos.

Cota inferior

- Todo algoritmo de ordenamiento debe ordenar su entrada.
- La entrada puede venir ordenada de $n!$ formas distintas.
- El algoritmo no sabe cuál de esas entradas es.
- Como cada comparación sólo puede tener dos valores posibles, el espacio de búsqueda se reduce a lo mucho a la mitad con cada una.
- Por lo tanto, si el algoritmo puede ordenar en un máximo de $T(n)$ comparaciones entonces
$$2^{T(n)} \geq n!$$
- Es decir $T(n) \geq \log n!$

Aproximando $\log n!$

- Observe que

$$(n!)^2 = (1 \cdot 2 \cdots n)(n \cdots 2 \cdot 1) = \prod_{k=1}^{n} k(n+1-k).$$

- La expresión $k(n+1-k)$ toma su valor mínimo cuando $k = 1$ o $k = n$ y toma su valor máximo cuando $k = (n+1)/2$.
- Por lo tanto

$$\prod_{k=1}^{n} n \leq (n!)^2 \leq \prod_{k=1}^{n} \frac{(n+1)^2}{4}.$$

- Es decir $n^{n/2} \leq n! \leq (n+1)^n/2^n$.
- Equivalentemente $\frac{1}{2} n \log n \leq \log n! \leq n \log \frac{n+1}{2}$.
- Por lo tanto $\log n! \in \Theta(n \log n)$.

Conclusión

- Entonces $T(n) \in \Omega(n \log n)$.
- Por lo tanto, cualquier algoritmo de ordenamiento basado en comparaciones debe hacer $\Omega(n \log n)$ comparaciones en el **peor caso**.
- Ordenamiento por mezcla cumple con la cota pero quicksort es peor.
- También se puede demostrar que cualquier algoritmo de ordenamiento basado en comparaciones debe hacer $\Omega(n \log n)$ comparaciones en el **caso promedio**.
- Ordenamiento por mezcla y quicksort cumplen con esta cota.

Contenido

k-selección

- Sea S un arreglo con n enteros distintos.
- Sea $1 \leq k \leq n$.
- El problema de k-**selección** consiste en encontrar el k-ésimo elemento más pequeño de S.
- Una forma de resolver el problema es ordenando S y regresando S_k.
- Este algoritmo se tarda $O(n \log n)$.
- ¿Podemos hacerlo más rápido?

Divide y vencerás

- Consideremos un algoritmo parecido a quicksort.

Función selección(S, k)

1. Si $|S| \leq 1$ entonces
   1. Regresa S_1
2. Si no entonces
   1. Escoge un elemento a de S **¿cómo?**
   2. Sean $S_<$, $S_=$ y $S_>$ los elementos de S que son $< a$, $= a$ y $> a$ respectivamente
   3. Sea $j = |S_<|$.
   4. Si $k = j + 1$ regresa a.
   5. Si no, si $k \leq j$ regresa selección$(S_<, k)$
   6. Si no, regresa selección$(S_>, k - j - 1)$.

Análisis del tiempo de ejecución

- Sea $T(n, k)$ el número **promedio** de comparaciones usadas por selección(S, k) en una lista S de tamaño n.
- Sea $T(n)$ el máximo de $T(n, 1), T(n, 2), \ldots, T(n, n)$.
- Obviamente $T(1) = 0$.

Tiempo de la llamada recursiva

- Como $|S_<| = j$ y $|S_>| = n - j - 1$ la llamada recursiva hace en promedio $\leq T(j)$ ó $\leq T(n - j - 1)$ comparaciones.
- Observe que j puede tomar cualquier valor de 0 a $n - 1$ con la misma probabilidad.
- Por lo tanto el tiempo promedio de la llamada recursiva es

$$\leq \frac{1}{n} \sum_{j=0}^{n-1} \left(T(j) \text{ ó } T(n-j-1) \right).$$

- ¿Cuándo es $T(j)$ y cuándo es $T(n - j - 1)$?
  - Si $k < j + 1$ hace recursión en $S_<$ y tarda $T(j)$.
  - Si $k = j + 1$ termina.
  - Si $k > j + 1$ hace recursión en $S_>$ y tarda $T(n - j - 1)$.

Desigualdad recursiva

- Por lo tanto el tiempo promedio de la llamada recursiva es

$$\leq \frac{1}{n} \left(\sum_{j=0}^{k-2} T(n-j-1) + \sum_{j=k}^{n-1} T(j) \right).$$

- Como separar S en $S_<$, $S_=$ y $S_>$ hace $n - 1$ comparaciones

$$\begin{aligned} T(n) &\leq \frac{1}{n} \left(\sum_{j=0}^{k-2} T(n-j-1) + \sum_{j=k}^{n-1} T(j) \right) + n - 1 \\ &= \frac{1}{n} \left(\sum_{j=n-k+1}^{n-1} T(j) + \sum_{j=k}^{n-1} T(j) \right) + n - 1. \end{aligned}$$

Escogiendo el valor de k

- ¿Qué valor de k maximiza esta expresión?

$$\sum_{j=n-k+1}^{n-1} T(j) + \sum_{j=k}^{n-1} T(j).$$

- Disminuir el valor de k borra un término de la suma izquierda y agrega un término a la suma derecha.
- Recordemos que $T(1) \leq T(2) \leq \ldots \leq T(n)$.
- Por lo tanto debemos escoger $k \approx n - k + 1$.
- En particular, si n es impar entonces $k = \frac{1}{2}(n+1)$.

Prueba de la desigualdad

- Por lo tanto

$$T(n) \leq \frac{2}{n}\left(\sum_{j=(n+1)/2}^{n-1} T(j)\right) + n - 1.$$

- Ahora demostraremos por inducción que $T(n) \leq 4(n-1)$.
- Esto es cierto para $n = 1$ pues $0 = T(1) \leq 4(1-1) = 0$.
- Ahora supongamos que $T(j) \leq 4(j-1)$ para toda $j < n$.

Paso de inducción

- Entonces:

$$\begin{aligned} T(n) &\leq \frac{2}{n}\left(\sum_{j=(n+1)/2}^{n-1} T(j)\right) + n - 1 \\ &\leq \frac{8}{n}\left(\sum_{j=(n+1)/2}^{n-1} (j-1)\right) + n - 1 \\ &= \frac{8}{n}\left(\sum_{j=(n-1)/2}^{n-2} j\right) + n - 1 \\ &= \frac{8}{n}\left(\sum_{j=1}^{n-2} j - \sum_{j=1}^{(n-3)/2} j\right) + n - 1. \end{aligned}$$

Continuación

- Simplificando:

$$\begin{aligned} T(n) &\leq \frac{8}{n}\left(\frac{(n-1)(n-2)}{2} - \frac{(n-3)(n-1)}{8}\right) + n - 1 \\ &= \frac{1}{n}\left(4n^2 - 12n + 8 - n^2 + 4n - 3\right) + n - 1 \\ &= 4n - 9 + \frac{5}{n} \\ &\leq 4n - 4. \end{aligned}$$

- Y por lo tanto $T(n) \in O(n)$ **en promedio**.
- En el peor caso $T(n) \in O(n^2)$ (como quicksort).

Contenido

Búsqueda binaria

- Encuentre el índice de x en el arreglo ordenado $A_i, \ldots, A_j$.
- Suponiendo que x **sí** está en el arreglo.

Función binaria(A, x, i, j)

1. Si $i = j$ entonces regresa i.
2. Si no:
   1. Haz $m = \lfloor \frac{i+j}{2} \rfloor$.
   2. Si $x \leq A_m$ entonces regresa binaria(A, x, i, m).
   3. Si no regresa binaria$(A, x, m+1, j)$.

Ecuación recursiva

- Supongamos que n es una potencia de 2.
- Sea $T(n)$ el máximo número de comparaciones usadas por la función en un arreglo con n elementos.
- Entonces

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T(n/2) + 1 & \text{si } n > 1 \end{cases}$$

Solución de la ecuación recursiva

- Por lo tanto

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= (T(n/4) + 1) + 1 \\ &= T(n/2^2) + 2 \\ &= T(n/2^i) + i \\ &= T(1) + \log n \\ &= \log n. \end{aligned}$$

- Y entonces $T(n) \in O(\log n)$.

Contenido

Contenido

Conteo de combinaciones

- Para escoger r objetos de entre n
  - se escoge el primer objeto y después se escogen $r-1$ objetos de entre los $n-1$ restantes o
  - no se escoge el primer objeto y después se escogen r objetos de entre los $n-1$ restantes.
- Por lo tanto
$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}.$$
- El caso base es cuando $r = 0$ o $r = n$.

Algoritmo de divide y vencerás

Función escoge(n, r)

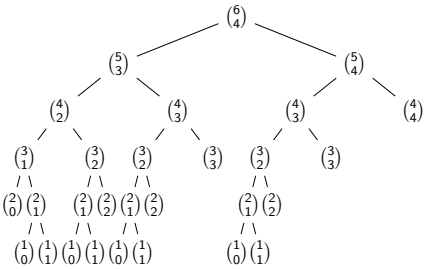
1. Si $r = 0$ o $r = n$ entonces regresa 1.
2. Si no regresa escoge$(n-1, r-1)$ + escoge$(n-1, r)$.

- Se puede demostrar que funciona por inducción en n.

Análisis del tiempo de ejecución

- Sea $T(n, r)$ la cantidad de veces que se llega al caso base cuando se llama a escoge(n, r).
- Entonces
$$T(n,r) = \begin{cases} 1 & \text{si } r = 0 \text{ o } r = n \\ T(n-1, r-1) + T(n-1, r) & \text{en otro caso} \end{cases}$$
- De donde inmediatamente $T(n, r) = \binom{n}{r}$.
- ¿Qué tan malo es esto?
- Se puede demostrar que $T(n, n/2) \in \Theta(2^n)$.

¿Por qué es tan lento?

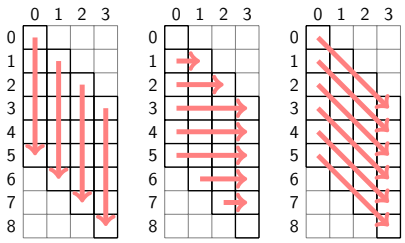


- El problema es que se calcula el mismo valor una y otra vez.

Algoritmo de programación dinámica

- Usemos un arreglo $P_{i,j}$ para almacenar $\binom{i}{j}$.
- Observe que para calcular $P_{i,j}$ sólo se necesita saber cuánto valen $P_{i-1,j-1}$ y $P_{i-1,j}$.
- Se puede escoger un orden adecuado para calcular las entradas de P de modo que cada cálculo sólo depende de otros cálculos ya hechos.
- Hay varias posibilidades para ese orden, por ejemplo, la del famoso **triángulo de Pascal**.
- Todas ellas se pueden ejecutar en tiempo $O(nr)$.
- Con cuidado sólo necesitarán espacio $O(r)$.

Varias formas de llenar la tabla P



Contenido

El problema de la mochila

- Suponga que tiene n objetos con volúmenes $v_1, v_2, \ldots, v_n$ y una mochila de volumen s.
- ¿Existirá un subconjunto de los objetos cuyo volumen total sea exactamente s?
- Existen muchas variantes del problema, por ejemplo:
  - los objetos tienen costos y se quiere maximizar el costo,
  - se desea maximizar el volumen que se llena de la mochila,
  - hay una cierta cantidad de cada tipo de objeto, etc.

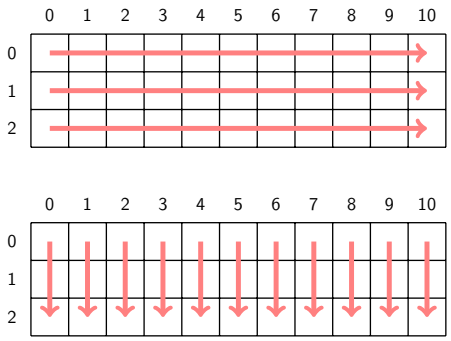
Algoritmo de divide y vencerás

- Sea $f(i,j)$ la respuesta a la pregunta ¿existe un subconjunto de los primeros i objetos tal que su volumen total sea j?
- ¿Cuándo es $f(i,j)$ verdadera?
  - Si $i = 0$ y $j = 0$ (caso base).
  - Si $i \geq 1$ y $f(i-1,j)$ es verdadera (no se usa el objeto j).
  - Si $i \geq 1$, $j \geq v_i$ y $f(i-1, j-v_i)$ es verdadera.
- En todos los demás casos $f(i,j)$ es falsa.
- Es fácil ver que este algoritmo se tarda $\Theta(2^n)$ en calcular $f(n,s)$.

Algoritmo de programación dinámica

- Almacenemos el valor de $f(i,j)$ en la entrada $F_{i,j}$ de una tabla.
- Observe que para calcular $F_{i,j}$ se necesita saber cuánto valen $F_{i-1,j}$ y $F_{i-1,j-v_i}$.
- En otras palabras, se necesita saber cuánto valen $F_{i-1,0}, F_{i-1,1}, \ldots, F_{i-1,j}$.
- Se puede escoger un orden adecuado para calcular las entradas de P de modo que cada cálculo sólo depende de otros cálculos ya hechos.
- Hay varias posibilidades para ese orden.
- Todas ellas se pueden ejecutar en tiempo $O(ns)$.
- Con cuidado sólo necesitarán espacio $O(n)$.

Dos formas de llenar la tabla F



Contenido

Producto encadenado de matrices

- Considere el problema de calcular el producto de n matrices rectangulares
$$M = M_1 \times M_2 \times \cdots \times M_n$$
donde la matriz M_i tiene r_{i-1} renglones y r_i columnas.
- Suponga que se usará el algoritmo común para multiplicar dos matrices.
- Entonces una multiplicación de una matriz de $p \times q$ por otra matriz de $q \times r$ requiere pqr operaciones.
- La multiplicación es **asociativa**: se puede evaluar en cualquier orden.
- Pero algunos órdenes requieren menos operaciones que otros. El problema es calcular la menor cantidad de operaciones posible.

Ejemplos

- Suponga que $r = (10, 20, 50, 1, 100)$.
- Si se multiplica de derecha a izquierda
$$M = M_1 \times (M_2 \times (M_3 \times M_4))$$
entonces se requieren
$$50 \cdot 1 \cdot 100 + 20 \cdot 50 \cdot 100 + 10 \cdot 20 \cdot 100 = 125000$$
operaciones.
- En cambio, si se comienza en el centro
$$M = (M_1 \times (M_2 \times M_3)) \times M_4$$
entonces se requieren
$$20 \cdot 50 \cdot 1 + 10 \cdot 20 \cdot 1 + 10 \cdot 1 \cdot 100 = 2200$$
operaciones.

Algoritmo de fuerza bruta

- Una posibilidad es la de intentar todas las formas de multiplicar n matrices, es decir, todas las formas de colocar paréntesis correctamente.
- Sea $X(n)$ la cantidad de formas de poner paréntesis a n matrices. Entonces $X(1) = X(2) = 1$ y

$$X(n) = \sum_{k=1}^{n-1} X(k)X(n-k)$$

lo cual se puede ver de considerar el producto

$$(M_1 \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_n).$$

- Por lo tanto $X(3) = 2$ y $X(4) = 5$.

Análisis del algoritmo de fuerza bruta

- Demostremos por inducción que $X(n) \geq 2^{n-2}$, lo cual es cierto para $1 \leq n \leq 4$. Ahora suponga que $n \geq 5$

$$\begin{aligned} X(n) &= \sum_{k=1}^{n-1} X(k)X(n-k) \\ &\geq \sum_{k=1}^{n-1} 2^{k-2}2^{n-k-2} \\ &= \sum_{k=1}^{n-1} 2^{n-4} \\ &= (n-1)2^{n-4} \\ &\geq 2^{n-2}. \end{aligned}$$

- En realidad se puede demostrar que $X(n) = \frac{1}{n}\binom{2n-2}{n-1}$.

Algoritmo de divide y vencerás

- Sea $c(i,j)$ la cantidad mínima de operaciones para calcular $M_i \times \cdots \times M_j$.
- ¿Cuál es el costo de partir el producto en M_k?

$$(M_i \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_j).$$

- Debe ser $c(i,k)$ mas $c(k+1,j)$ mas la cantidad de operaciones necesaria para un producto de $r_{i-1} \times r_k$ por $r_k \times r_j$.
- Es decir $c(i,k) + c(k+1,j) + r_{i-1}r_k r_j$.
- El producto se puede partir en $i \leq k < j$.
- El caso base es $c(i,i) = 0$.

Análisis del tiempo de ejecución

- Sea $T(n)$ el **peor** tiempo de ejecución para calcular $c(i,j)$ cuando $j - i + 1 = n$.
- Entonces $T(0) = c$ y para $n > 0$ tenemos

$$T(n) = \sum_{m=1}^{n-1} (T(m) + T(n-m)) + dn.$$

- Por lo tanto, para $n > 0$ tenemos

$$T(n) = 2\sum_{m=1}^{n-1} T(m) + dn.$$

- Entonces

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

es decir

$$T(n) = 3T(n-1) + d.$$

- Esto se puede resolver por sustitución para obtener $T(n) \in \Theta(3^n)$.

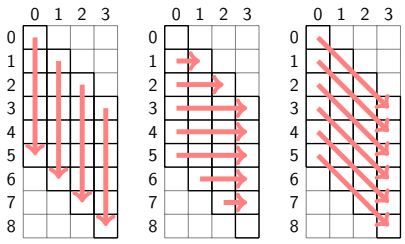
Algoritmo de programación dinámica

- De nuevo, estos dos algoritmos son demasiado lentos porque resuelven los mismos subproblemas una y otra vez.
- Almacenemos $c(i,j)$ en la entrada $A_{i,j}$ de una matriz.
- Por lo tanto $A_{i,i} = 0$ y para $i < j$ tenemos

$$A_{i,j} = \min_{i \leq k < j} \left(A_{i,k} + A_{k+1,j} + r_{i-1}r_k r_j\right).$$

- Este algoritmo se ejecuta en tiempo $O(n^3)$.

Tres formas de llenar la tabla A



Contenido

Operaciones con árboles binarios

- Los árboles binarios de búsqueda tienen cuatro operaciones interesantes:
  1. Encontrar una clave x en un árbol T.
  2. Regresar la clave más pequeña en un árbol T.
  3. Borrar la clave x de un árbol T.
  4. Insertar la clave x en un árbol T.
- Si el árbol binario tiene d niveles entonces todas estas operaciones se pueden implementar en tiempo $O(d)$.
- ¿Pero qué tan grande puede ser d en términos de la cantidad n de nodos del árbol?
- En el peor de los casos $O(n)$ y al menos $\Omega(\log n)$.

Análisis del algoritmo de inserción

- ¿Cuál es el tiempo promedio que se toma en insertar n claves distintas en un árbol binario de búsqueda inicialmente vacío?
- Suponga que queremos insertar en algún orden aleatorio las claves $x_1, x_2, \ldots, x_n$ donde $x_1 < x_2 < \cdots < x_n$.
- El tiempo de ejecución es proporcional a la cantidad $T(n)$ de comparaciones entre claves.
- Observe que es igualmente probable que cualquier clave x_j $(1 \leq j \leq n)$ sea la raíz del árbol (aquella que se haya insertado primero).

Obtención de la recursión

- Suponga que la raíz es x_j.
- Todas las claves $x_1, x_2, \ldots, x_{j-1}$ van a la izquierda de la raíz, por lo que se necesitan $j-1$ comparaciones con la raíz y $T(j-1)$ comparaciones entre ellas.
- Todas las claves $x_{j+1}, x_{j+2}, \ldots, x_n$ van a la derecha de la raíz, por lo que se necesitan $n-j$ comparaciones con la raíz y $T(n-j)$ comparaciones entre ellas.
- Por lo tanto si x_j es la raíz se necesitan

$$T(j-1) + T(n-j) + n - 1$$

comparaciones entre claves.

Solución de la recursión

- Por lo tanto $T(0) = 0$ y para $n > 0$

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{j=1}^{n}(T(j-1) + T(n-j)) + n - 1 \\ &= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + n - 1. \end{aligned}$$

- ¡Pero esto es exactamente la misma fórmula que la de quicksort!
- Por lo tanto $T(n) \in O(n \log n)$ en promedio.
- Y cada una de las n inserciones toma $O(\log n)$ en promedio.

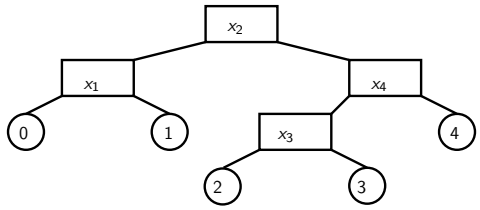
Contenido

Árboles binarios de búsqueda óptima

- Se tienen n claves $x_1 < x_2 < \cdots < x_n$.
- Para cada $1 \leq i \leq n$ se tiene la probabilidad p_i de que se solicite una búsqueda de x_i.
- Para cada $0 \leq i \leq n$ se tiene la probabilidad q_i de que se solicite una búsqueda de x con $x_i < x < x_{i+1}$ (donde $x_0 = -\infty$ y $x_{n+1} = +\infty$).
- Se desea construir un árbol binario de búsqueda que minimice la cantidad esperada de comparaciones por solicitud de búsqueda.

Nodos ficticios

- Añadamos nodos **ficticios** llamados $0, 1, \ldots, n$.
- El nodo i estará en el lugar al que llegaríamos si se hiciera una búsqueda fallida de x con $x_i < x < x_{i+1}$.



El costo de un nodo

- La **profundidad** $P(x)$ de un nodo x se define como 0 si x es la raíz y como $P(y) + 1$ si y es el padre de x.
- La cantidad de comparaciones hechas para encontrar a x_i es $P(x_i) + 1$ y esto ocurre con probabilidad p_i.
- La cantidad de comparaciones hechas en la búsqueda fallida de x con $x_i < x < x_{i+1}$ es $P(i)$ y esto ocurre con probabilidad q_i.

El costo del árbol de búsqueda

- Entonces la cantidad esperada de comparaciones es

$$\sum_{h=1}^{n} p_h(P(x_h) + 1) + \sum_{h=0}^{n} q_h P(h)$$

donde el primer sumando corresponde con los nodos reales y el segundo con los nodos ficticios.

- Llamémosle a esto el **costo** del árbol de búsqueda.
- Ahora queremos encontrar el árbol de búsqueda de menor costo.

El peso del árbol de búsqueda

- Sea $T_{i,j}$ el árbol de costo mínimo $c_{i,j}$ con nodos reales $x_{i+1}, \ldots, x_j$ y nodos ficticios $i, i+1, \ldots, j$.
- Estamos interesados en $T_{0,n}$ y en $c_{0,n}$.
- Definamos el **peso** de $T_{i,j}$ como

$$w_{i,j} = \sum_{h=i+1}^{j} p_h + \sum_{h=i}^{j} q_h.$$

- ¿Qué significa el peso de un árbol?
- Observe que aumentar la profundidad de $T_{i,j}$ en 1 (haciéndolo hijo de otro nodo) incrementa el costo de $T_{i,j}$ en $w_{i,j}$.

Construcción recursiva de $T_{i,j}$

- $T_{i,i}$ es un árbol vacío con $w_{i,i} = q_i$ y $c_{i,i} = 0$.
- Si $i < j$ entonces para encontrar $T_{i,j}$ se debe escoger una raíz x_k $(i+1 \leq k \leq j)$ con hijo izquierdo $T_{i,k-1}$ e hijo derecho $T_{k,j}$.
- Observe que si $k = i+1$ no hay hijo izquierdo y si $k = j$ no hay hijo derecho.

Cálculo recursivo de $c_{i,j}$

- Suponga que se escogió a x_k como la raíz de $T_{i,j}$, entonces:

$$\begin{aligned} c_{i,j} &= (c_{i,k-1} + w_{i,k-1}) + (c_{k,j} + w_{k,j}) + p_k \\ &= c_{i,k-1} + c_{k,j} + (w_{i,k-1} + w_{k,j} + p_k) \\ &= c_{i,k-1} + c_{k,j} + \sum_{h=i+1}^{j} p_h + \sum_{h=i}^{j} q_h \\ &= c_{i,k-1} + c_{k,j} + w_{i,j}. \end{aligned}$$

- Por supuesto, en realidad

$$c_{i,j} = \min_{i+1 \leq k \leq j} (c_{i,k-1} + c_{k,j} + w_{i,j}).$$

Algoritmo de programación dinámica

- Almacenemos $c_{i,j}$ en $C[i,j]$ y $w_{i,j}$ en $W[i,j]$.
- Estas dos cantidades se pueden calcular en tiempo $O(n^3)$.

1. Para cada $0 \leq i \leq n$ haz $W[i,i] \leftarrow 0$ y $C[i,i] \leftarrow 0$.
2. Para cada $1 \leq \ell \leq n$ haz
   1. Para cada $0 \leq i \leq n - \ell$ haz
      1. $j \leftarrow i + \ell$.
      2. $W[i,j] \leftarrow W[i,j-1] + p_j + q_j$.
      3. $C[i,j] \leftarrow \min_{i+1 \leq k \leq j}(C[i,k-1] + C[k,j] + W[i,j])$.

Contenido

Caminos de costo mínimo

- Sea $D = (V, A)$ una gráfica dirigida con n vértices y suponga que cada arco $(u, v) \in A$ tiene un costo $c_{u,v} \geq 0$.
- Un **camino** en D de u a w es una secuencia de $k \geq 0$ arcos

$$(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$$

con $v_0 = u$ y $v_k = w$.
- El costo de ese camino es la suma de los costos de sus arcos

$$c_{v_0,v_1} + c_{v_1,v_2} + \cdots + c_{v_{k-1},v_k}.$$

- Nos interesa encontrar los caminos de costo mínimo entre todas las parejas de vértices de D.

Algoritmo de programación dinámica

- Sea A_k la matriz de $n \times n$ que almacena en $A_k[i,j]$ el costo mínimo de un camino de i a j pasando únicamente por $1, 2, \ldots, k$.
- Obviamente, para $k = 0$ tenemos que:
  - $A_0[i,i] = 0$ para toda $1 \leq i \leq n$.
  - $A_0[i,j] = c_{i,j}$ si existe un arco de i a j en D.
  - $A_0[i,j] = +\infty$ en cualquier otro caso.
- Además, la matriz A_n almacena los costos mínimos de los caminos entre todas las parejas de vértices. ¿Por qué?

Definición recursiva de A_k

- Ya sabemos cuánto vale A_0. Supongamos que $k > 0$.
- ¿Cómo son los caminos de i a j que pasan por $1, 2, \ldots, k$?
  - Si no pasan por k entonces son los mismos que pasan por $1, 2, \ldots, k-1$.
  - Si pasan por k entonces sólo lo hacen una vez.
- Entonces los costos mínimos de esos caminos son:
  - $A_{k-1}[i,j]$ si no pasan por k.
  - $A_{k-1}[i,k] + A_{k-1}[k,j]$ si pasan por k.
- Por lo tanto

$$A_k[i,j] = \min\{A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]\}.$$

Usando sólo un arreglo

- Observe que todas las entradas de A_k dependen del renglón k y la columna k de A_{k-1}.
- Pero ese renglón y columna no cambian de A_{k-1} a A_k:
  - $A_k[k,j] = \min\{A_{k-1}[k,j], A_{k-1}[k,k] + A_{k-1}[k,j]\} = A_{k-1}[k,j]$.
  - $A_k[i,k] = \min\{A_{k-1}[i,k], A_{k-1}[i,k] + A_{k-1}[k,k]\} = A_{k-1}[i,k]$.
- Y por lo tanto podemos usar el mismo arreglo en el cálculo.

Algoritmo de Floyd

Inicialización

Para $i \leftarrow 1$ hasta n haz
1. Para $j \leftarrow 1$ hasta n haz
   1. Si $(i,j) \in A$ entonces $A[i,j] \leftarrow c_{i,j}$.
   2. Si no entonces $A[i,j] \leftarrow +\infty$.
2. $A[i,i] \leftarrow 0$.

Cálculo

Para $k \leftarrow 1$ hasta n haz
1. Para $i \leftarrow 1$ hasta n haz
   1. Para $j \leftarrow 1$ hasta n haz
      1. Si $A[i,k] + A[k,j] < A[i,j]$ entonces $A[i,j] \leftarrow A[i,k] + A[k,j]$.

- Este algoritmo se ejecuta en tiempo $O(n^3)$.

Contenido

Cerradura transitiva

- Dada una gráfica dirigida $D = (V, A)$ se desea saber para cada par de vértices u y v si existe algún camino de u a v.
- La **cerradura transitiva** de D es la gráfica dirigida $D^* = (V, A^*)$ tal que hay un arco de u a v en A^* si existe algún camino de u a v en D.
- Una forma sencilla de resolver este problema es ejecutando el algoritmo de Floyd con costos unitarios para los arcos de D.
- Al final de la ejecución del algoritmo $A[i,j] \neq +\infty$ si y sólo si existe algún camino de i a j.
- Pero hay un algoritmo más elegante usando una matriz booleana.

Algoritmo de Warshall

Inicialización

Para $i \leftarrow 1$ hasta n haz
1. Para $j \leftarrow 1$ hasta n haz
   1. Si $(i,j) \in A$ entonces $A[i,j] \leftarrow V$.
   2. Si no entonces $A[i,j] \leftarrow F$.
2. $A[i,i] \leftarrow V$.

Cálculo

Para $k \leftarrow 1$ hasta n haz
1. Para $i \leftarrow 1$ hasta n haz
   1. Para $j \leftarrow 1$ hasta n haz
      1. $A[i,j] \leftarrow A[i,j]$ o ($A[i,k]$ y $A[k,j]$).

Contenido

Contenido

Algoritmos glotones

- Se comienza con una solución a un subproblema pequeño.
- Se va construyendo una solución al problema completo.
- A cada paso se toma una decisión que parece buena en ese momento.
- Desventajas:
  1. No siempre funcionan.
  2. Las decisiones tomadas pueden ser desastrosas.
  3. Es difícil demostrar que son correctos.
- Ventajas:
  1. Cuando funcionan son muy rápidos.
  2. Como son sencillos son fáciles de implementar.

Almacenamiento óptimo en cinta

- Suponga que tiene n archivos de longitudes $m_1, m_2, \ldots, m_n$.
- Encuentre el mejor orden para almacenarlos en una cinta suponiendo:
  1. Cada lectura inicia con la cinta rebobinada.
  2. Cada lectura toma tiempo **proporcional** a la longitud del archivo que se desea mas las longitudes de los archivos anteriores.
  3. Se leerán todos los archivos.

Ejemplo

- Suponga que $n = 3$, $m_1 = 5$, $m_2 = 10$ y $m_3 = 3$.
- Hay $3! = 6$ órdenes posibles:
  - 1,2,3: $(5 + 10 + 3) + (5 + 10) + 5 = 38$.
  - 1,3,2: $(5 + 3 + 10) + (5 + 3) + 5 = 31$.
  - 2,1,3: $(10 + 5 + 3) + (10 + 5) + 10 = 43$.
  - 2,3,1: $(10 + 3 + 5) + (10 + 3) + 10 = 41$.
  - 3,1,2: $(3 + 5 + 10) + (3 + 5) + 3 = 29$.
  - 3,2,1: $(3 + 10 + 5) + (3 + 10) + 3 = 34$.
- El mejor orden es $3, 1, 2$.

Un algoritmo glotón

- Considere el siguiente algoritmo:
  1. Comienza con la cinta vacía.
  2. Para i desde 1 hasta n haz:
     1. Escoge el siguiente archivo más corto.
     2. Escríbelo a continuación en la cinta.
- Este algoritmo toma lo que parece la mejor decisión a corto plazo sin revisar si ésta es la mejor decisión a largo plazo.
- ¿Será ésta una decisión inteligente?
- Por supuesto, el algoritmo corre en tiempo $O(n \log n)$.

Análisis de correctitud

- Suponga que tenemos los archivos $a_1, a_2, \ldots, a_n$ de longitudes $m_1, m_2, \ldots, m_n$.
- Sea $i_1, i_2, \ldots, i_n$ una permutación de $1, 2, \ldots, n$ y suponga que almacenamos los archivos en el orden

$$a_{i_1}, a_{i_2}, \ldots, a_{i_n}.$$

- ¿Cuánto cuesta esto?
- Leer el k-ésimo archivo a_{i_k} de la cinta cuesta $\sum_{j=1}^{k} m_{i_j}$.
- Por lo tanto leer todos los archivos de la cinta cuesta

$$\sum_{k=1}^{n} \sum_{j=1}^{k} m_{i_j} = \sum_{k=1}^{n} (n - k + 1) m_{i_k}.$$

¿Qué es lo que queremos demostrar?

- El algoritmo glotón escoge los archivos a_i en orden no decreciente de su tamaño m_i.
- Queremos demostrar que ésta es la permutación de costo mínimo.
- **Por demostrar**: Cualquier permutación en orden no decreciente de tamaño tiene costo mínimo.
- Sea $\pi = (i_1, i_2, \ldots, i_n)$ una permutación de $1, 2, \ldots, n$ que **no** está en orden no decreciente de tamaño.
- Demostraremos que π no puede ser de costo mínimo.

Una permutación más barata

- Como $m_{i_1}, m_{i_2}, \ldots, m_{i_n}$ no está en orden no decreciente entonces debe existir $1 \leq j < n$ tal que $m_{i_j} > m_{i_{j+1}}$.
- Sea π' la permutación obtenida de π al intercambiar i_j e i_{j+1}.
- La diferencia de costos de las permutaciones π y π' es:

$$\begin{aligned} m(\pi) - m(\pi') &= ((n - j + 1)m_{i_j} + (n - j)m_{i_{j+1}}) \\ &\quad -((n - j + 1)m_{i_{j+1}} + (n - j)m_{i_j}) \\ &= (n - j + 1)(m_{i_j} - m_{i_{j+1}}) - (n - j)(m_{i_j} - m_{i_{j+1}}) \\ &= m_{i_j} - m_{i_{j+1}} \\ &> 0. \end{aligned}$$

- Por lo tanto π no puede ser de costo mínimo.

Contenido

Problema continuo de la mochila

- Este problema es parecido a uno que vimos antes.
  1. Se tienen n objetos $A_1, A_2, \ldots, A_n$.
  2. Se tiene una mochila de volumen S.
  3. El objeto A_i tiene volumen v_i.
  4. El objeto A_i tiene peso p_i.
  5. Una fracción $0 \leq x_i \leq 1$ del objeto A_i tiene volumen $x_i v_i$ y peso $x_i p_i$.
- Se desea llenar la mochila tanto como sea posible usando **fracciones** de los objetos de modo que el peso sea el mínimo posible.

Ejemplo

- Suponga que $S = 20$, que los volúmenes son $v_1 = 18$, $v_2 = 10$ y $v_3 = 15$ y que los pesos son $p_1 = 25$, $p_2 = 15$ y $p_3 = 24$.
- Algunas posibilidades para las fracciones son:

(x_1, x_2, x_3)	volumen total	peso total
(1, 1/5, 0)	20	28.0
(1, 0, 2/15)	20	28.2
(0, 1, 2/3)	20	31.0
(0, 1/2, 1)	20	31.5

- La primera opción es la mejor de las que llevamos.
- ¿Pero cómo saber si es la mejor de todas?

Algoritmo glotón (informal)

- Defina la **densidad** del objeto A_i como p_i/v_i.
- Use tanto como sea posible de los objetos de baja densidad.
- En otras palabras, procese los objetos en orden de densidad creciente.
- Si todo cabe, úselo todo.
- Si no, llene el espacio restante con una fracción del objeto actual.
- Descarte el resto.

Algoritmo glotón (formal)

- Primero, ordene los objetos en orden de densidad no decreciente, de modo que $p_i/v_i \leq p_{i+1}/v_{i+1}$ para toda $1 \leq i < n$.
- Después haga lo siguiente:
  1. $v \leftarrow S$.
  2. $i \leftarrow 1$.
  3. Mientras $v_i \leq v$ haz:
     1. $x_i \leftarrow 1$.
     2. $v \leftarrow v - v_i$.
     3. $i \leftarrow i + 1$.
  4. $x_i \leftarrow v/v_i$.
- Para $j \leftarrow i + 1$ a n haz
  1. $x_j \leftarrow 0$.
- Obviamente este algoritmo se ejecuta en tiempo $O(n \log n)$.

Correctitud

- Demostraremos que el algoritmo glotón produce una solución de costo mínimo.
- Sea $X = (x_1, x_2, \ldots, x_n)$ la solución generada por el algoritmo glotón.
- Si todas las x_i son iguales a 1 entonces la solución es óptima.
- En caso contrario, sea j el entero más pequeño tal que $x_j \neq 1$.
- Del algoritmo sabemos que $x_i = 1$ para $1 \leq i < j$, que $0 \leq x_j < 1$ y que $x_i = 0$ para $j < i \leq n$.
- Por lo tanto

$$\sum_{i=1}^{j} x_i v_i = S.$$

Estrategia de prueba

- Sea $Y = (y_1, y_2, \ldots, y_n)$ una solución de peso mínimo.
- Demostraremos que X debe tener el mismo peso que Y.
- Si $X = Y$ ya acabamos. Si no, sea k el menor entero tal que $x_k \neq y_k$.
- Transformaremos Y en X conservando el peso.
- En particular, transformaremos Y en una solución $Z = (z_1, z_2, \ldots, z_n)$ que se **parece** más a X.
- Para ser precisos, $x_i = y_i = z_i$ para toda $1 \leq i < k$ y $z_k = x_k$.
- El primer paso será demostrar que $y_k < x_k$.
- **Caso 1**: Suponga que $k < j$, entonces $x_k = 1$ y como $x_k \neq y_k$ entonces $y_k < x_k$.

Caso 2

- Ahora suponga que $k = j$. Como $x_k \neq y_k$ entonces podemos suponer para una contradicción que $y_k > x_k$.

$$S = \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{k-1} y_i v_i + y_k v_k + \sum_{i=k+1}^{n} y_i v_i = \sum_{i=1}^{k-1} x_i v_i + y_k v_k + \sum_{i=k+1}^{n} y_i v_i.$$

- Reorganizando algunos términos obtenemos que

$$S = \sum_{i=1}^{k} x_i v_i + (y_k - x_k) v_k + \sum_{i=k+1}^{n} y_i v_i = S + (y_k - x_k) v_k + \sum_{i=k+1}^{n} y_i v_i > S.$$

- Esto contradice que Y es una solución, por lo que $y_k < x_k$.

Caso 3

- Finalmente suponga que $k > j$. Entonces $x_k = 0$ y $y_k > 0$ y por lo tanto

$$S = \sum_{i=1}^{j} y_i v_i + \sum_{i=j+1}^{n} y_i v_i = \sum_{i=1}^{j} x_i v_i + \sum_{i=j+1}^{n} y_i v_i = S + \sum_{i=j+1}^{n} y_i v_i > S.$$

- Como esto no es posible, este caso no puede ocurrir.
- Así en cualquier caso ya sabemos que $y_k < x_k$.

Construcción de Z

- Ahora suponga que incrementamos el valor de y_k a x_k y que decrementamos tantos de los valores $y_{k+1}, \ldots, y_n$ como sea necesario de modo que el volumen total permanezca en S.
- Llamemos $Z = (z_1, z_2, \ldots, z_n)$ a esta nueva solución. Entonces:
  1. $(z_k - y_k)v_k > 0$.
  2. $\sum_{i=k+1}^{n} (z_i - y_i)v_i < 0$.
  3. $(z_k - y_k)v_k + \sum_{i=k+1}^{n} (z_i - y_i)v_i = 0$.

El costo de Z

- Calculemos el peso de Z:

$$\begin{aligned}\sum_{i=1}^{n} z_i p_i &= \sum_{i=1}^{k-1} z_i p_i + z_k p_k + \sum_{i=k+1}^{n} z_i p_i \\ &= \sum_{i=1}^{k-1} y_i p_i + z_k p_k + \sum_{i=k+1}^{n} z_i p_i \\ &= \sum_{i=1}^{n} y_i p_i - \sum_{i=k}^{n} y_i p_i + z_k p_k + \sum_{i=k+1}^{n} z_i p_i \\ &= \sum_{i=1}^{n} y_i p_i + (z_k - y_k) p_k + \sum_{i=k+1}^{n} (z_i - y_i) p_i \\ &= \sum_{i=1}^{n} y_i p_i + (z_k - y_k)\frac{v_k p_k}{v_k} + \sum_{i=k+1}^{n} (z_i - y_i)\frac{v_i p_i}{v_i}.\end{aligned}$$

El costo de Z (continuación)

- Usando la segunda propiedad de Z y las densidades decrecientes:

$$\begin{aligned}\sum_{i=1}^{n} z_i p_i &\leq \sum_{i=1}^{n} y_i p_i + (z_k - y_k)\frac{v_k p_k}{v_k} + \sum_{i=k+1}^{n} (z_i - y_i)\frac{v_i p_k}{v_k} \\ &= \sum_{i=1}^{n} y_i p_i + \left((z_k - y_k)v_k + \sum_{i=k+1}^{n} (z_i - y_i)v_i\right)\frac{p_k}{v_k} \\ &= \sum_{i=1}^{n} y_i p_i.\end{aligned}$$

- Donde al final usamos la tercera propiedad de Z.

Fin del análisis

- Entonces el peso de Z es a lo mucho el peso de Y.
- Pero Y es una solución de peso mínimo.
- Entonces el peso de Z es **igual** al peso de Y.
- Pero Z se **parece** más a X: coinciden en k posiciones.
- Si repetimos este proceso podemos transformar Y en X manteniendo el peso a cada paso.
- Por lo tanto X es una solución de peso mínimo.

Contenido

Caminos más cortos desde un vértice

- Sea $D = (V, A)$ una gráfica dirigida.
- Cada arco $(v, w) \in A$ tiene costo $c(v, w) \geq 0$.
- Supondremos que $c(v, v) = 0$ para toda $v \in V$ y que $c(v, w) = +\infty$ en cualquier otro caso.
- Sea $s \in V$ un vértice especial llamado **fuente**.
- Se desea encontrar un camino más corto desde s hacia cada uno de los vértices de D.
- Sea $d(v, w)$ el costo del camino más corto de v a w.

Idea del algoritmo de Dijkstra

- Mantengamos un conjunto S de vértices para los cuales conozcamos la distancia mínima desde la fuente.
  1. Al principio $S = \{s\}$.
  2. Luego iremos añadiendo vértices uno por uno.
  3. Al final $S = V$.
- Diremos que un camino es **interno** si todos sus vértices internos (los que no son ni el primero ni el último) están en S.
- Mantengamos un arreglo D tal que $D[w]$ es el costo del camino interno más corto de s a w.
- Si $w \in S$ entonces ese es el camino más corto de s a w. ¿Por qué?
- Un vértice w está en la **frontera** si $w \notin S$ pero existe un arco (u, w) con $u \in S$.
- Todos los caminos internos terminan en S o en la frontera.

Agregando un vértice a S

- ¿Cuál vértice agregamos a S?
- El vértice $w \in V \setminus S$ con menor valor $D[w]$.
- Ya que el valor de D para los vértices que no están en la frontera es infinito, se tiene que w estará en la frontera.
- Probemos que para ese vértice $D[w] = d(s, w)$. Sea x el primer vértice en la frontera en el camino más corto de s a w, entonces:

$$\begin{aligned}D[w] &\geq d(s, w) \\ &= d(s, x) + d(x, w) \\ &= D[x] + d(x, w) \\ &\geq D[w] + d(x, w) \\ &\geq D[w].\end{aligned}$$

Observación

- Probemos que si u se pone en S antes que v entonces $d(s, u) \leq d(s, v)$.
- Haremos la prueba por inducción en el número de vértices que ya están en S en el momento en el que se agrega v.
- La hipótesis es claramente cierta cuando $|S| = 1$.
- Ahora suponga que es cierta cuando $|S| < n$ y considere lo que pasa cuando $|S| = n$.
- En particular, considere lo que pasa cuando se agrega v a S.
- Sea x el último vértice interno en el camino más corto a v.

Primer caso

- Supongamos que x entró a S antes que u.
- Considere el momento en el que u entró a S.
- Como x ya estaba en S, v estaba en la frontera.
- Pero como se escogió a u para entrar a S antes que v se debió haber tenido que $D[u] \leq D[v]$.
- Como se escogió a u, también se tiene que $d(s,u) = D[u]$.
- Por la definición de x y como x entró a S antes que u, $d(s,v) = D[v]$.
- Por lo tanto $d(s,u) = D[u] \leq D[v] = d(s,v)$.

Segundo caso

- Ahora supongamos que x entró a S después que u.
- Como x entró a S antes que v, en el momento en el que entró se tenía que $|S| < n$.
- Por la hipótesis de inducción $d(s,u) \leq d(s,x)$.
- Y por la definición de x, $d(s,x) \leq d(s,v)$.
- Por lo tanto $d(s,u) \leq d(s,v)$.

Actualizando el vector de costos

- Los valores de D en los vértices en la frontera o fuera de ella pueden cambiar, debido a que puede haber nuevos caminos internos conteniendo a w.
- Esto puede pasar de dos formas:
  1. w es el último vértice interno en un camino interno nuevo o
  2. w es algún vértice interno en un camino interno nuevo.
- Los valores de D en los vértices en S ya no cambian.
- Por lo tanto, para todo vértice $v \in V$, cuando w se mueve de la frontera a S se tiene que

$$D[v] = \text{mín}\{D[v], D[w] + c(w,v)\}.$$

Algoritmo de Dijkstra

1. $S \leftarrow \{s\}$.
2. Para cada $v \in V$ haz:
   1. $D[v] \leftarrow c(s,v)$.
3. Para cada $i \leftarrow 1$ a $n-1$ haz:
   1. Escoge $w \in V \setminus S$ con la $D[w]$ más pequeña.
   2. $S \leftarrow S \cup \{w\}$.
   3. Para cada vértice $v \in V$ haz:
      1. $D[v] = \text{mín}\{D[v], D[w] + c(w,v)\}$.

Implementación con un vector

- Almacenemos S como un vector de bits.
- La inicialización de S toma tiempo $O(n)$.
- La inicialización de D toma tiempo $O(n)$.
- El ciclo principal se realiza $O(n)$ veces.
  - Allí se hace una búsqueda lineal en tiempo $O(n)$.
  - Una actualización de S en tiempo $O(1)$.
  - Y se hacen $O(n)$ actualizaciones de D.
- En total esta implementación toma tiempo $O(n^2)$.

Implementación con un montículo

- Almacenemos $S' = V \setminus S$ en un montículo indexado por D.
- La inicialización de S' toma tiempo $O(n \log n)$.
- La inicialización de D toma tiempo $O(n)$.
- Analizemos con cuidado el ciclo principal.
  - Se realizan $O(n)$ extracciones del mínimo en tiempo $O(n \log n)$.
  - Se ajusta el montículo una vez por cada arco en tiempo $O(m \log n)$.
- En total esta implementación toma tiempo $O(m \log n)$.

¿Cuál implementación se debe usar?

- Eso depende del número m de aristas.
- Si $m \log n \in \Theta(n^2)$ ambas implementaciones son buenas.
- Se debe usar la primera implementación si la gráfica dirigida es **densa**. Técnicamente, si $m \in \Omega(n^2/\log n)$.
- Se debe usar la segunda implementación si la gráfica dirigida es **dispersa**. Técnicamente, si $m \in o(n^2/\log n)$.
- Si en la segunda implementación se usa un **montículo de Fibonacci** entonces la complejidad disminuye a $O(m + n \log n)$.
- En este caso, basta que $m \in o(n^2)$ para que esta tercera implementación sea mejor que la primera.

Calculando los caminos más cortos

- Para poder recuperar los caminos mantendremos un segundo arreglo P tal que $P[v]$ es el **predecesor** de v en el camino interno más corto.

1. $S \leftarrow \{s\}$.
2. Para cada $v \in V$ haz:
   1. $D[v] \leftarrow c(s,v)$.
   2. Si $(s,v) \in A$ entonces $P[v] \leftarrow s$ y si no $P[v] \leftarrow 0$.
3. Para cada $i \leftarrow 1$ a $n-1$ haz:
   1. Escoge $w \in V \setminus S$ con la $D[w]$ más pequeña.
   2. $S \leftarrow S \cup \{w\}$.
   3. Para cada vértice $v \in V$ haz:
      1. Si $D[w] + c(w,v) < D[v]$ entonces $D[v] = D[w] + c(w,v)$ y $P[v] \leftarrow w$.

Reconstruyendo los caminos

- Para cada vértice $v \neq s$ sabemos que $P[v]$ es su predecesor en un camino más corto de s a v.
- Por lo tanto, el camino más corto de s a v es el camino más corto de s a $P[v]$ seguido del arco $(P[v], v)$.
- Un algoritmo recursivo muy sencillo obtiene los caminos:

Procedimiento camino(s, v)

1. Si $v \neq s$ entonces camino$(s, P[v])$.
2. Imprime v.

- La complejidad de este algoritmo es lineal en la longitud (número de vértices) del camino.

Contenido

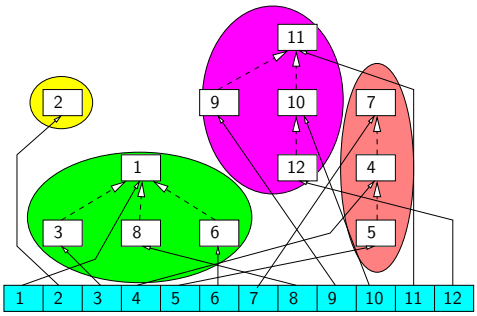
El problema de unión y búsqueda

- Se tiene el conjunto $\{1, 2, \ldots, n\}$.
- Al principio éste se encuentra **particionado** en n subconjuntos **disjuntos**, cada uno con un elemento.
- Se desea llevar a cabo las siguientes operaciones:
  - encuentra(x) regresa el **nombre** del subconjunto que contiene a x.
  - une(x, y) combina los dos subconjuntos que contienen a x y a y en uno solo.
- ¿Qué usaremos como el nombre de un subconjunto?
- Usaremos a uno de sus elementos.

Una estructura de datos

- Usaremos un bosque y un arreglo.
- El bosque constará de nodos y apuntadores a estos.
- Los elementos del conjunto estarán almacenados en los nodos.
- Cada **hijo** tendrá un apuntador a su **padre**.
- El arreglo P contiene un apuntador P_i al nodo que contiene a i.

Ejemplo de la estructura

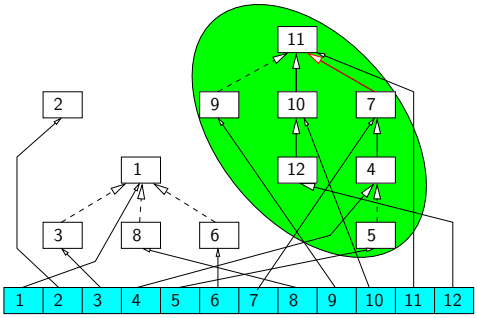


Cuatro conjuntos con nombres 1, 2, 7 y 11.

Implementación de las operaciones

- Para hacer encuentra(x) se sigue la cadena de apuntadores que comienza en P_x hasta llegar a la raíz del árbol que contiene a x. Se regresa el elemento del conjunto almacenado en la raíz.
- Para hacer une(x, y) se siguen las cadenas de apuntadores que comienzan en P_x y P_y hasta llegar a las raíces de los árboles que contienen a x y a y, respectivamente.
  - Si estas raíces son iguales no es necesario hacer nada.
  - En caso contrario, se hace que la raíz de uno de los árboles apunte a la raíz del otro.

Ejemplo de la estructura



Unión de los conjuntos que contienen al 4 y al 12.

Análisis y mejora

- El tiempo de ejecución de los algoritmos es proporcional a la altura de los árboles.
- Pero esto puede ser $\Omega(n)$ para algunos árboles.
- Una mejor idea es cambiar el último paso de une(x, y) para que haga que la raíz del árbol **más pequeño** apunte a la raíz del árbol **más grande**.
- Esto mejorará el tiempo de ejecución a $O(\log n)$.
- Observe que se debe saber el tamaño de cada árbol.
  - Al principio todos los árboles tienen un nodo y esta cuenta se almacena en el nodo.
  - Cada vez que se haga una unión se actualiza la cuenta en la raíz en tiempo $O(1)$.
- Estos algoritmos todavía se pueden mejorar a $O(\log^* n)$ usando **compresión de caminos**, pero por el momento no es de nuestro interés.

Análisis de la mejora

- Demostraremos por inducción en n que si se usa esta mejora entonces la altura de un árbol con n nodos es a lo mucho $\lfloor \log n \rfloor + 1$.
- Para $n = 1$ es obvio. Suponga que es cierto para árboles con menos de n nodos y sea T un árbol con n nodos construido por el algoritmo.
- T se formó uniendo dos árboles T_1 y T_2 con m y $n - m$ nodos, respectivamente. Podemos suponer que $m \leq n/2$.
- Entonces la altura de T es

$$\begin{aligned} \text{altura}(T) &= \text{máx}\{\text{altura}(T_1) + 1, \text{altura}(T_2)\} \\ &= \text{máx}\{\lfloor \log m \rfloor + 2, \lfloor \log(n - m) \rfloor + 1\} \\ &\leq \text{máx}\{\lfloor \log n/2 \rfloor + 2, \lfloor \log n \rfloor + 1\} \\ &= \lfloor \log n \rfloor + 1. \end{aligned}$$

Contenido

Árboles abarcadores

- Sean $G = (V, E)$ y $H = (U, F)$ dos gráficas.
- H es una **subgráfica abarcadora** de G si $U = V$ y $F \subseteq E$.
- H es un **bosque abarcador** de G si es una subgráfica abarcadora de G que además es un bosque (es decir, no tiene ciclos).
- H es un **árbol abarcador** de G si es un bosque abarcador de G que además es un árbol (es decir, es conexo).

Árboles abarcadores de costo mínimo

- Sea $G = (V, E)$ una gráfica con costos en las aristas (c_e es el costo de $e \in E$).
- El costo de una subgráfica de G es la suma de los costos de sus aristas.
- Se desea encontrar un árbol abarcador de costo mínimo.

Propiedades de árboles y bosques

- Sea $T = (V, F)$ un árbol. Entonces:
  - Para toda $u, v \in V$ existe un único camino de u a v en T.
  - Si se agrega cualquier arista $\{u, v\} \notin F$ a T entonces se forma un único ciclo.
- Sea $B = (V, A)$ un bosque abarcador de $G = (V, E)$. Suponga que B está formado por los árboles $(V_1, A_1), (V_2, A_2), \ldots, (V_k, A_k)$ y sea $e \in E \setminus A$ la arista más barata con exactamente un extremo en V_i. Entonces:
  - Existe un árbol $T = (V, F)$ tal que $e \in F$ y que es el más barato de entre todos los que contienen al bosque B.

Demostración de la propiedad de los bosques

- Sea S un árbol que contiene a B pero que no contiene a e.
- Si se agrega e a S se obtiene un único ciclo.
- Existe en S otra arista d que tiene exactamente un extremo en V_i.
- Por lo tanto $c_d \geq c_e$.
- Ahora considere el árbol S' que se obtiene de S al cambiar d por e.
- Es obvio que S' no es más caro que S.

Algoritmo general para árboles abarcadores mínimos

- Esta propiedad implica que el siguiente algoritmo general funciona:

1. Sea B el bosque formado por los árboles que constan de un solo vértice.
2. Mientras haya más de un árbol en B haz:
   1. Escoge un árbol cualquiera (V_i, A_i) de B.
   2. Sea e una arista de costo mínimo con exactamente un extremo en V_i.
   3. Une dos de los árboles de B usando la arista e.
3. Regresa B.

- Existen muchas formas de llevar esto a cabo.
- Dos de ellas son los algoritmos de Prim y de Kruskal.

Algoritmo un poco más formal

1. $B \leftarrow \emptyset$.
2. Para $j \leftarrow 1$ a n haz:
   1. $V_j \leftarrow \{j\}$.
   2. $B \leftarrow B \cup \{(V_j, \emptyset)\}$.
3. Mientras haya más de un árbol en B haz:
   1. Escoge un árbol cualquiera (V_i, A_i) de B.
   2. Sea $e = (u, v)$ una arista de costo mínimo con $u \in V_i$, $v \in V_j$ y $j \neq i$.
   3. $V_i \leftarrow V_i \cup V_j$, $A_i \leftarrow A_i \cup A_j \cup \{e\}$.
   4. Borra (V_j, A_j) de B.
4. Regresa B.

Diseño del algoritmo de Prim

- El algoritmo de Prim simplemente toma $i = 1$ en cada iteración.
- Para ello se necesita recordar las aristas en A_1 y los vértices en V_1.
- También se necesita almacenar las aristas que salen de V_1 en una estructura de datos que nos permite escoger rápidamente la de menor costo.
- Esto se puede hacer con una cola de prioridad Q.
- De esta manera el algoritmo completo se ejecuta en tiempo $O(e \log e)$.

Algoritmo de Prim

1. $A_1 \leftarrow \emptyset$, $V_1 \leftarrow \{1\}$, $Q \leftarrow \emptyset$.
2. Para cada $w \in V$ tal que $(1, w) \in E$ haz:
   1. Inserta $(1, w)$ en Q.
3. Mientras $|V_1| < n$ haz:
   1. Elimine $e = (u, v)$ del frente de Q con $u \in V_1$.
   2. Si $v \notin V_1$ entonces haz:
      1. $A_1 \leftarrow A_1 \cup \{e\}$.
      2. $V_1 \leftarrow V_1 \cup \{v\}$.
      3. Para cada $w \in V$ tal que $(v, w) \in E$ inserta (v, w) en Q.
4. Regresa B.

Diseño del algoritmo de Kruskal

- A cada iteración del algoritmo escoge $e = (u, v)$ como la arista de menor costo sin usar y suponga que $u \in V_i$ y $v \in V_j$.
- Ahora necesitamos recordar los conjuntos de vértices $V_1, V_2, \ldots, V_k$ de los árboles del bosque abarcador. Esto es una partición.
- También necesitamos recordar las aristas del bosque abarcador.
- Además necesitamos almacenar las aristas sin usar en una estructura de datos que nos permita escoger rápidamente la de menor costo.
- Todo esto lo podemos hacer con un bosque de unión y búsqueda.
- De esta manera el algoritmo completo se ejecuta en tiempo $O(e \log e)$.

Algoritmo de Kruskal

1. $A \leftarrow \emptyset$.
2. Inicializa la estructura B de unión y búsqueda con V.
3. Ordena las aristas de E en orden de costo creciente.
4. Mientras haya más de un árbol en B haz:
   1. Sea $e = (u, v)$ la siguiente arista por orden de costo.
   2. Si busca$(u) \neq$ busca(v) entonces:
      1. une(u, v).
      2. $A \leftarrow A \cup \{(u, v)\}$.
5. Regresa A.

Contenido

Búsqueda exhaustiva

- Algunas veces el mejor algoritmo para un problema es intentar todas las posibles soluciones.
- Esto siempre será lento, pero hay algunas técnicas generales:
  - generar las 2^n cadenas binarias de longitud n,
  - generar las k^n cadenas k-arias de longitud n,
  - generar las $n!$ permutaciones de n objetos o
  - generar las $\binom{n}{r}$ combinaciones de r objetos escogidos de entre n.
- La **búsqueda con retroceso** acelera la búsqueda exhaustiva a través de la **poda**.

Diseño de un algoritmo de búsqueda con retroceso

- Se escoge un objeto básico (cadenas, permutaciones, combinaciones).
- Se comienza con el código de divide y vencerás para la generación del objeto básico elegido.
- Se coloca el código para probar la propiedad deseada en el caso base de la recursión.
- Se coloca el código de poda antes de la llamada recursiva.

Formas generales

Algoritmo de generación genera(m)

1. Si $m = 0$ entonces procesa(A) y si no:
   1. Para cada posible opción j haz:
      1. $A[m] = j$ y genera($m - 1$).

Algoritmo de búsqueda con retroceso genera(m)

1. Si $m = 0$ entonces procesa(A) y si no:
   1. Para cada posible opción j haz:
      1. Si j es consistente con $A[m + 1, \ldots, n]$ entonces $A[m] = j$ y genera($m - 1$).

Contenido

Cadenas de bits

- Se desea generar todas las cadenas de n bits.
- Usaremos divide y vencerás.
- Mantendremos la cadena de bits actual en un arreglo $A[1, \ldots, n]$.
- El objetivo es llamar a procesa(A) para cada cadena binaria A.
- Se llama al procedimiento binario(n).

Procedimiento binario(m)

1. Si $m = 0$ entonces procesa(A) y si no:
   1. $A[m] = 0$ y binario($m - 1$).
   2. $A[m] = 1$ y binario($m - 1$).

Prueba de correctitud I

- Demostraremos por inducción en m que para toda $m \geq 0$ el procedimiento binario(m) llama a procesa(A) exactamente una vez para cada cadena de m bits almacenada en $A[1, \ldots, m]$.
- Esto es claramente cierto para $m = 0$.
- Ahora suponga que binario($m - 1$) llama a procesa(A) exactamente una vez para cada cadena de $m - 1$ bits almacenada en $A[1, \ldots, m - 1]$.

Prueba de correctitud II

- Observe que binario(m):
  - primero pone $A[m]$ en 0 y llama a binario($m - 1$) y
  - luego pone $A[m]$ en 1 y llama a binario($m - 1$).
- Por la hipótesis de inducción, esto llama a procesa(A) exactamente una vez para cada cadena de m bits almacenada en $A[1, \ldots, m]$:
  - primero con las cadenas que terminan en 0 y
  - luego con las cadenas que terminan en 1.
- Por lo tanto el enunciado es cierto.

Análisis del tiempo de ejecución

- Sea $T(n)$ el tiempo de ejecución de binario(n). Suponga que procesa(A) toma tiempo $O(1)$. Entonces

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

- Por lo tanto, usando sustitución repetida obtenemos la solución

$$T(n) = (c + d)2^{n-1} - d.$$

- Y así $T(n) \in O(2^n)$.
- Como hay 2^n cadenas de n bits entonces el algoritmo es óptimo.

Cadenas k-arias

- Se desea generar todas las cadenas de n dígitos del 0 al $k - 1$.
- Mantendremos la cadena k-aria actual en un arreglo $A[1, \ldots, n]$.
- El objetivo es llamar a procesa(A) para cada cadena k-aria A.
- Se llama al procedimiento cadena(n).
- La prueba de correctitud y el análisis son similares a los anteriores.
- El procedimiento cadena(n) toma tiempo en $O(k^n)$ y es óptimo.

Procedimiento cadena(m)

1. Si $m = 0$ entonces procesa(A) y si no:
   1. Para cada j de 0 a $k - 1$ haz:
      1. $A[m] = j$ y
      2. cadena($m - 1$).

Cadenas generalizadas

- Observe que k puede ser diferente para cada dígito de la cadena.
- Suponga que $A[m]$ puede tomar los valores $0, \ldots, D[m] - 1$.

Procedimiento cadena(m)

1. Si $m = 0$ entonces procesa(A) y si no:
   1. Para cada j de 0 a $D[m] - 1$ haz:
      1. $A[m] = j$ y
      2. cadena($m - 1$).

Contenido

Problema de la mochila

- Recordemos que el problema de la mochila consiste en decidir si existe algún subconjunto de $\{s_1, s_2, \ldots, s_n\}$ que sume S.
- Usemos una cadena de bits $A[1, \ldots, n]$ para resolver este problema.
- $A[i] = 1$ significará que usaremos el objeto i y $A[i] = 0$ que no.
- Si revisamos todas las cadenas binarias podemos resolver el problema.
- Pero además podemos hacer una poda. Si S es la suma restante entonces:
  - hacer $A[m] = 0$ siempre será legal pero
  - hacer $A[m] = 1$ sólo será legal si $s_m \leq S$.

Algoritmo

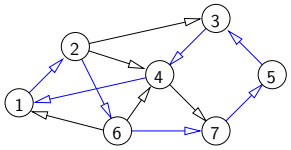
- Para imprimir todas las soluciones llame a mochila(n, S).
- Este algoritmo toma tiempo $O(2^n)$ (mas el tiempo que se tarde en imprimir las t soluciones $O(tn)$).

Procedimiento mochila(m, ℓ)

1. Si $m = 0$ entonces:
   1. Si $\ell = 0$ entonces imprime(A).
2. Si no:
   1. $A[m] = 0$ y mochila$(m-1, \ell)$.
   2. Si $s_m \leq \ell$ entonces
      1. $A[m] = 1$ y mochila$(m-1, \ell - s_m)$.

Ciclos hamiltonianos

- Un **ciclo hamiltoniano** en una gráfica dirigida es un ciclo que pasa a través de cada vértice exactamente una vez.
- El problema es: dada una gráfica dirigida $D = (V, A)$ encontrar un ciclo hamiltoniano (si es que tiene alguno).



Descripción del algoritmo

- Almacenemos la gráfica con listas de adyacencia (para cada vértice v su lista es la de los vértices w para los cuales $(v, w) \in A$).
- Almacenemos el ciclo hamiltoniano como un arreglo $H[1, \ldots, n]$:

$$H[n] \to H[n-1] \to \cdots \to H[2] \to H[1] \to H[n].$$

- Usaremos el algoritmo de cadenas generalizadas.
- **Poda**: mantendremos un arreglo $U[1, \ldots, n]$ tal que $U[m]$ es verdadero si y sólo si el vértice m no ha sido usado. Al entrar a m:
  - si $U[m]$ es verdadero será legal, pero
  - si $U[m]$ es falso será ilegal y podemos podar.
- Al principio todas las entradas de U serán verdaderas, excepto $U[n]$.
- Las listas de adyacencia se mantendrán en el arreglo N y el vector D almacenará el grado de cada uno de los vértices.

Algoritmo

Procedimiento hamilton(m)

1. Si $m = 0$ entonces procesa(H) y si no:
2. Para $j = 1$ hasta $D[H[m+1]]$ haz:
   1. $w \leftarrow N[H[m+1], j]$.
   2. Si $U[w]$ es verdadero entonces:
      1. Haga $U[w]$ falso.
      2. $H[m] = w$ y hamilton$(m-1)$.
      3. Haga $U[w]$ verdadero.

Procedimiento procesa(H)

1. Haga b falso.
2. Para $j = 1$ hasta $D[H[1]]$ haz:
   1. Si $N[H[1], j] = H[n]$ entonces haga b verdadero.
3. Si b es verdadero imprime(H).

Notas sobre el algoritmo

- Al principio del algoritmo debemos hacer $H[n] = n$.
- El procedimiento procesa(H) cierra el ciclo. Hasta ese momento H contiene un **camino hamiltoniano** de $H[n]$ a $H[1]$ y falta revisar si hay un arco de $H[1]$ a $H[n]$.
- La poda se da cuando se revisa si $U[w]$ es verdadero.
- En ese caso $U[w]$ se hace falso para indicar que ya usamos w y posteriormente $U[w]$ se hace verdadero para indicar que ya no lo estamos usando.

Análisis

- ¿Cuánto se tarda hamilton(n) en encontrar un ciclo hamiltoniano?
- En el peor de los casos no hay ciclos hamiltonianos y hay que revisar.
- Sea $T(n)$ el tiempo de ejecución de hamilton(n). Suponga que la gráfica tiene grado máximo d. Entonces

$$T(n) = \begin{cases} bd & \text{si } n = 0 \\ dT(n-1) + c & \text{si } n > 0 \end{cases}$$

- Por sustitución repetida obtenemos $T(n) \in O(d^n)$.

Agente viajero

- Suponga que cada arco de $D = (V, A)$ tiene un costo (c_a para $a \in A$).
- El problema del agente viajero es el de encontrar un ciclo hamiltoniano de costo mínimo (donde el costo es la suma de los costos de los arcos usados).
- Un problema relacionado es el de encontrar un ciclo hamiltoniano de costo $\leq B$ para alguna cota B.
- Resulta que es suficiente resolver la versión acotada: el problema original se puede resolver usando búsqueda binaria.

Búsqueda binaria

- Suponga que se tiene un procedimiento BTSP(x) que regresa un ciclo hamiltoniano de costo $\leq x$ si es que existe.
- Para resolver el problema original llame a TSP($0, B$):

Procedimiento TSP(a, b)

1. Si $a = b$ entonces regresa BTSP(a).
2. Si no haz $m = \lfloor \frac{1}{2}(a+b) \rfloor$.
3. Si BTSP(m) tiene éxito entonces regresa TSP(a, m).
4. Si no regresa TSP($m+1, b$).

- Se puede tomar B como n veces el costo de la arista más cara.
- El procedimiento TSP(a, b) se llama $O(\log B)$ veces.

Algoritmo modificado

Procedimiento hamilton(m, b)

1. Si $m = 0$ entonces procesa(H, b) y si no:
2. Para $j = 1$ hasta $D[H[m+1]]$ haz:
   1. $w \leftarrow N[H[m+1], j]$.
   2. Si $U[w]$ es verdadero y $C[H[m+1], w] \leq b$ entonces:
      1. Haga $U[w]$ falso.
      2. $H[m] = w$.
      3. hamilton($m-1, b - C[H[m+1], w]$).
      4. Haga $U[w]$ verdadero.

Procedimiento procesa(H, b)

1. Si $C[H[1], n] \leq b$ entonces imprime(H).

Contenido

Permutaciones

- Se desea generar todas las permutaciones de n enteros distintos.
- Se puede utilizar la siguiente solución:
  - Se hará búsqueda con retroceso a través de todas las cadenas n-arias de longitud n, podando aquellas que no sean permutaciones.
  - Para ello se mantendrá un arreglo $U[1, \ldots, n]$ donde $U[m]$ es verdadero si y sólo si m no ha sido usada.
  - Se mantendrá a la permutación actual en un arreglo $A[1, \ldots, n]$.
  - El objetivo es llamar a procesa(A) exactamente una vez para cada permutación almacenada en $A[1, \ldots, n]$.

Algoritmo

- Observe que este algoritmo es muy parecido al que usamos para caminos hamiltonianos.
- Esto se debe a que un camino hamiltoniano es una permutación de los vértices de la gráfica.

Procedimiento permuta(m)

1. Si $m = 0$ entonces procesa(A) y si no:
2. Para $j = 1$ hasta n haz:
   1. Si $U[j]$ es verdadero entonces:
      1. Haga $U[j]$ falso.
      2. $A[m] = j$.
      3. permuta($m-1$).
      4. Haga $U[j]$ verdadero.

Análisis I

- Este algoritmo claramente se ejecuta en tiempo $O(n^n)$.
- Pero este análisis no es el mejor posible porque no considera la poda.
- Calculemos la cantidad de veces que se pregunta si $U[j]$ es verdadero.
- ¿Cuál es la situación cuando se hace esa pregunta?
- Para alguna $0 \leq i < n$ el algoritmo:
  - ha llenado las últimas i entradas de A con parte de una permutación y
  - está intentando n candidatos para llenar la siguiente entrada de A.
- ¿Cómo se ven esas i entradas?

Análisis II

- Son i entradas escogidas de entre n y permutadas de alguna forma.
- Entonces existen $\binom{n}{i} i!$ formas de llenar esa parte.
- Y por lo tanto la cantidad de veces que se hace la pregunta es

$$\begin{aligned} n\sum_{i=0}^{n-1} \binom{n}{i} i! &= n\sum_{i=0}^{n-1} \frac{n!}{(n-i)!} \\ &= n \cdot n! \sum_{i=1}^{n} \frac{1}{i!} \\ &\leq (n+1)! \cdot (e-1) \\ &\leq 1.719(n+1)! \end{aligned}$$

- Y por lo tanto toma tiempo $O((n+1)!)$ (mucho mejor que $O(n^n)$).

Permutaciones más rápido

- El algoritmo visto **no es óptimo**.
- Genera $n!$ permutaciones en tiempo $O((n+1)!)$.
- Un mejor algoritmo usa la estrategia de divide y vencerás:
  - Al principio se hace $A[i] = i$ para toda $1 \leq i \leq n$.
  - En lugar de probar los valores $1, \ldots, n$ en $A[n]$ se intercambian los valores de $A[1, \ldots, n-1]$ con $A[n]$.
- De esta forma primero se generan las $(n-1)!$ permutaciones con n en la última posición y así sucesivamente hasta que se generan las $(n-1)!$ permutaciones con 1 en la última posición.

Algoritmo

Procedimiento permuta(m)

1. Si $m = 1$ entonces procesa(A), si no:
   1. permuta($m-1$).
   2. Para $i = m-1$ hasta 1 haz:
      1. Intercambia $A[m]$ con $A[i]$.
      2. permuta($m-1$).
      3. Intercambia $A[m]$ con $A[i]$.

Análisis

- Sea $T(n)$ la cantidad de intercambios hechos por permuta(n).
- Entonces $T(1) = 0$ y $T(n) = nT(n-1) + 2(n-1)$ para $n \geq 2$.
- Demostraremos por inducción en n que $T(n) \leq 2n! - 2$.
- Esto es claramente cierto para $n = 1$. Si lo suponemos cierto para n:

$$\begin{aligned} T(n+1) &= (n+1)T(n) + 2n \\ &\leq (n+1)(2n! - 2) + 2n \\ &= 2(n+1)! - 2. \end{aligned}$$

- Por lo tanto $T(n) \in O(n!)$ y el algoritmo es óptimo.

Contenido

Ciclos hamiltonianos en gráficas densas

- Usemos una matriz de adyacencia: $M[i,j]$ es verdadero si $(i,j) \in A$.

Procedimiento hamilton(m)

1. Si $m = 0$ entonces procesa(H) y si no:
2. Para $j = m$ hasta 1 haz:
   1. Si $M[H[m+1], H[j]]$ es verdadero entonces:
      1. Intercambia $H[m]$ con $H[j]$.
      2. hamilton($m-1$).
      3. Intercambia $H[m]$ con $H[j]$.

Procedimiento procesa(H)

1. Si $M[H[1], n]$ es verdadero imprime(H).

Análisis

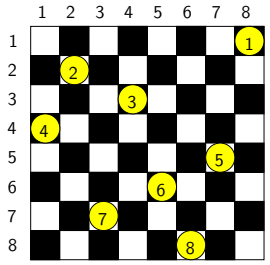
- Tenemos dos algoritmos para encontrar circuitos hamiltonianos.
  - El que usa cadenas generalizadas que corre en $O(d^n)$.
  - El que usa permutaciones que corre en $O((n-1)!)$.
- ¿Cuál es el mejor de los dos?
- Depende del valor de d (grado máximo de la gráfica).
- El algoritmo de cadenas es mejor si $d \leq \frac{n}{e}$.
- Para justificar esto se necesita la aproximación de Stirling

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

que no hemos demostrado en clase.

Problema de las reinas

- ¿De cuántas formas se pueden poner n reinas en un tablero de ajedrez de $n \times n$ de modo que ninguna reina ataque a ninguna otra?
- Si almacenamos en $A[i]$ el renglón en el que se encuentra la reina de la columna i obtenemos una permutación.



Contenido

Combinaciones

- Queremos generar todas las combinaciones de n objetos escogidos r a la vez. Recordemos que existen $\binom{n}{r}$ de ellas.
- Mantendremos a la combinación actual en el arreglo $A[1, \ldots, r]$.

Procedimiento escoge(m, q)

1. Si $q = 0$ entonces procesa(A).
2. Si no, para $i = q$ hasta m haz:
   1. $A[q] = i$.
   2. escoge($i-1, q-1$).

Correctitud

- Demostraremos que el algoritmo funciona en tres partes:
  1. Primero mostraremos que sólo se procesan combinaciones.
  2. Luego que se procesan el número correcto de combinaciones.
  3. Finalmente que ninguna combinación se procesa dos veces.
- Esto probará que cada combinación se procesa exactamente una vez.

Primera parte: sólo combinaciones

- Demostraremos por inducción en r que en las combinaciones procesadas por escoge(n, r) el arreglo $A[1, \ldots, r]$ contiene una combinación del 1 al n.
- Esto es cierto para $r = 0$ así que supongamos que $r > 0$ y que para toda $i \geq r - 1$ en las combinaciones procesadas por escoge$(i, r - 1)$ el arreglo $A[1, \ldots, r - 1]$ contiene una combinación del 1 al $i - 1$.
- Ahora observe que escoge(n, r) llama a escoge$(i - 1, r - 1)$ con i en el rango $r \leq i \leq n$.
- Por la hipótesis de inducción y ya que $A[r] = i$ en las combinaciones procesadas por escoge(n, r) tenemos que el arreglo $A[1, \ldots, r]$ contiene una combinación de $r - 1$ objetos del 1 al $i - 1$ seguida del valor i, que forma una combinación de r objetos del 1 al i.

Segunda parte: número correcto

- Demostraremos por inducción en r que una llamada a escoge(n, r) procesa exactamente $\binom{n}{r}$ combinaciones.
- Esto es cierto para $r = 0$ así que supongamos que $r > 0$ y que una llamada a escoge$(i, r - 1)$ procesa exactamente $\binom{i}{r-1}$ combinaciones para toda $r - 1 \leq i \leq n$.
- Ahora observe que escoge(n, r) llama a escoge$(i - 1, r - 1)$ con i en el rango $r \leq i \leq n$.
- Entonces, por la hipótesis de inducción se procesa un total de

$$\sum_{i=r}^{n} \binom{i-1}{r-1} = \sum_{i=r-1}^{n-1} \binom{i}{r-1} = \binom{n}{r}$$

  combinaciones.
- El último paso necesita la identidad $\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}$.

Tercera parte: sin repeticiones

- De la misma forma, se puede demostrar por inducción en r que una llamada a escoge(n, r) procesa combinaciones de r objetos escogidos del 1 al n sin repetir ninguna combinación.
- La prueba se deja como ejercicio.
- Que se genere el número exacto de combinaciones y que no se repita ninguna implica que el algoritmo genera cada combinación exactamente una vez.

Análisis

- Sea $T(n, r)$ la cantidad de asignaciones hechas por escoge(n, r).
- Demostraremos que

$$T(n, r) \leq r\binom{n}{r}$$

  para toda $0 \leq r \leq n$.
- La prueba será por inducción en r para $n \geq 1$ (y para $r = 0$ es cierto).
- Del algoritmo se obtiene que

$$T(n, r) = \sum_{i=r-1}^{n-1} T(i, r - 1) + (n - r + 1).$$

- Suponga que $r > 0$ y que para toda $i < n$ se cumple que

$$T(i, r - 1) \leq (r - 1)\binom{i}{r-1}.$$

Desigualdad

- Entonces, por la hipótesis de inducción y la identidad vista

$$\begin{aligned} T(n, r) &= \sum_{i=r-1}^{n-1} T(i, r - 1) + (n - r + 1) \\ &\leq \sum_{i=r-1}^{n-1} \left((r - 1)\binom{i}{r-1}\right) + (n - r + 1) \\ &= (r - 1) \sum_{i=r-1}^{n-1} \binom{i}{r-1} + (n - r + 1) \\ &= (r - 1)\binom{n}{r} + (n - r + 1) \\ &\leq r\binom{n}{r}. \end{aligned}$$

- Donde el último paso requiere la desigualdad $\binom{n}{r} \geq n - r + 1$.

Optimalidad

- El algoritmo visto corre en tiempo $O(r\binom{n}{r})$.
- Por lo que parece que no es óptimo debido a que hay $\binom{n}{r}$ combinaciones.
- Sin embargo, una inspección cuidadosa parece indicar que $T(n, r) < 2\binom{n}{r}$ cuando $r \leq n/2$.
- Cuando $r > n/2$ se podrían generar los objetos **no escogidos**.
- Demostraremos por inducción en r que si $1 \leq r \leq n/2$ entonces

$$T(n, r) \leq 2\binom{n}{r} - r.$$

- Haremos por separado los casos $r = 1$, $r = 2$ y $3 \leq r \leq n/2$.

El caso $r = 1$

- Observemos primero que $T(n, 1) = n$ para toda $n \geq 1$.
- Ahora observemos que

$$2\binom{n}{1} - 1 = 2n - 1 \geq n.$$

- Y por lo tanto

$$T(n, 1) \leq 2\binom{n}{1} - 1$$

  como se requería.

El caso $r = 2$

- Usemos $T(n,1) = n$ para calcular exactamente $T(n,2)$:

$$\begin{aligned} T(n,2) &= \sum_{i=1}^{n-1} T(i,1) + (n-1) \\ &= \sum_{i=1}^{n-1} i + (n-1) \\ &= \frac{1}{2}(n+2)(n-1). \end{aligned}$$

- Ahora observemos que $2\binom{n}{2} - 2 = n^2 - n - 2$.
- Entonces $T(n,2) \leq 2\binom{n}{2} - 2$ si y sólo si $\frac{1}{2}(n+2)(n-1) \leq n^2 - n - 2$ si y sólo si $n^2 - 3n - 2 \geq 0$ si y sólo si $(n+1)(n-4) + 2 \geq 0$ lo cual es cierto puesto que $n \geq 2r = 4$.

El caso $3 \leq r \leq n/2$

- Ahora demostraremos por inducción en n que si $3 \leq r \leq n/2$ entonces

$$T(n,r) \leq 2\binom{n}{r} - r.$$

- El caso base es $n = 6$ (la única opción es $r = 3$). En este caso el algoritmo hace 34 asignaciones para calcular 20 combinaciones. Como $2 \cdot 20 - 2 = 38 > 34$ se cumple la hipótesis en este caso.
- Ahora supongamos que $n > 6$ (lo que permitirá que $r \geq 3$).

Desigualdad

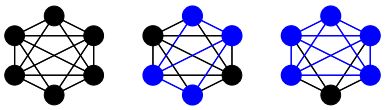
- Por la hipótesis de inducción y los casos $r = 1$ y $r = 2$:

$$\begin{aligned} T(n,r) &= \sum_{i=r-1}^{n-1} T(i,r-1) + (n-r+1) \\ &\leq \sum_{i=r-1}^{n-1} \left(2\binom{i}{r-1} - (r-1)\right) + (n-r+1) \\ &= 2\sum_{i=r-1}^{n-1} \binom{i}{r-1} - (n-r+1)(r-2) \\ &= 2\binom{n}{r} - (n-r+1)(r-2) \\ &\leq 2\binom{n}{r} - (n-n/2+1)(3-2) \\ &< 2\binom{n}{r} - r. \end{aligned}$$

Contenido

Subgráficas completas

- Sean $G = (V, E)$ y $H = (U, F)$ dos gráficas no dirigidas.
- Se dice que H es **completa** si F contiene a todas las aristas posibles entre los vértices en U.
- Se dice que H es una **subgráfica inducida** de G si $U \subseteq V$, $F \subseteq E$ y F contiene a todas las aristas de G que unen vértices en U.
- Finalmente, se dice que H es un **clan** de G si H es una subgráfica completa inducida de G.



El problema del clan

- Se tiene una gráfica no dirigida $G = (V, E)$ y un entero r.
- Se desea encontrar un clan de G con r vértices (si es que existe).
- Podemos usar la matriz de adyacencia de G para verificar que dos vértices son adyacentes en tiempo $O(1)$.
- Este algoritmo toma tiempo $O(r\binom{n}{r})$ si $r \leq n/2$.

Procedimiento clan(m, q)

1. Si $q = 0$ entonces imprime(A).
2. Si no, para $i = q$ hasta m haz:
   1. Si q y j están unidos por una arista para toda $q < j \leq r$ entonces:
      1. $A[q] = i$.
      2. clan$(i-1, q-1)$.

Conjunto independiente

- Un **conjunto independiente** es una subgráfica inducida que no contiene ninguna arista.
- El tamaño de un conjunto independiente es su cantidad de vértices.
- El problema del conjunto independiente es: Dada una gráfica $G = (V, E)$ y un número entero r, decidir si G tiene un conjunto independiente de tamaño r.
- Observe que un conjunto independiente de G es un clan de la gráfica complementaria $\bar{G} = (V, \bar{E})$, donde $\bar{E}$ contiene a todas las aristas posibles excepto a las de E.
- Por lo tanto el algoritmo recién visto resuelve el problema del conjunto independiente si se le aplica a $\bar{G}$.