Contents

Chapter 1

Introducción

1.1 Algoritmos

> *"Un algoritmo es un conjunto de pasos claramente definidos que a partir de una cierta entrada (input) produce una determinada salida (output)."*

Algoritmos y programas Debemos distinguir algoritmo de programa, un algoritmo es independiente del lenguaje en el cual se programa, de la máquina en la cual se implemente y de otras restricciones que hacen a la puesta en operación del algoritmo. Desde el punto de vista del estudio de los algoritmos los mismos pueden considerarse como entidades matemáticas abstractas independientes de restricciones tecnológicas.

Validación de un algoritmo Un algoritmo es *correcto* si el resultado que produce siempre resuelve un determinado problema a partir de una entrada válida. Demostrar ya sea en forma rigurosa o intuitiva que un algoritmo es correcto es el primer paso indispensable en el análisis de un algoritmo. A esta fase del análisis de algoritmos se la conoce como **validación del algoritmo**. Los algoritmos que no son correctos a veces pueden ser útiles si, por ejemplo, producen una respuesta aproximada a un problema particularmente difícil en forma eficiente.

> **Importante** El primer paso en todo análisis de un algoritmo es validarlo, es decir, demostrar que el algoritmo es correcto.

Algoritmos determinísticos Un algoritmo es determinístico si la respuesta que produce se puede conocer a partir de los datos de entrada. Un algoritmo es

no-determinístico cuando no es determinístico. Que un algoritmo sea o no sea determinístico no aporta dato alguno sobre la corrección del algoritmo.

El estudio de los algoritmos se puede dividir en dos grandes categorías: el análisis de algoritmos y el disenio de algoritmos.

1.1.1 Análisis de algoritmos

El análisis de algoritmos mediante el uso de herramientas como por ejemplo la evaluación de costos intenta determinar que tan eficiente es un algoritmo para resolver un determinado problema. En general el aspecto mas interesante a analizar de un algoritmo es su costo.

Costo de tiempo

Suele ser el mas importante, indica cuanto tiempo insume un determinado algoritmo para encontrar la solución a un problema, se mide en función de la cantidad o del tamanio de los datos de entrada.

Costo de espacio

Mide cuanta memoria (espacio) necesita el algoritmo para funcionar correctamente.

Consideremos por un momento un programa que juega al ajedrez, el programa funcionar en base a un único algoritmo que se llama *SearchBestMove*. Que devuelve para una posición de piezas dada cual es la mejor movida para un cierto bando (blancas o negras). Con este algoritmo el programa es sencillo, dependiendo de quien empiece lo único que debe hacer es aplicar el algoritmo cada vez que le toque, esperar la movida del contrario y luego aplicar el algoritmo a la nueva posición. Pensemos ahora en el algoritmo *SearchBestMove*, este algoritmo para ser óptimo debería analizar todas las posibles jugadas, todas las posibles respuestas a dichas jugadas y asi sucesivamente, formando un árbol de jugadas posibles del cual selecciona la jugada que produce mejor resultado final. Este aunque funcione en forma ultra-veloz tiene un grave problema de **espacio**, no hay memoria suficiente en ninguna máquina construida por el hombre para almacenar todas las posibles combinaciones de jugadas.

Importante El análisis del costo de un algoritmo comprende el costo en tiempo y en espacio.

1.1.2 Diseño de algoritmos

El disenio de algoritmos se encarga de encontrar cual es el mejor algoritmo para un problema determinado, en general existen algunos paradigmas básicos que pueden aplicarse para encontrar un buen algoritmo. Es claro que esta es una tarea difícil que requiere de conocimientos específicos y de una habilidad particular. Algunas de las técnicas mas utilizadas en el disenio de algoritmos son las siguientes:

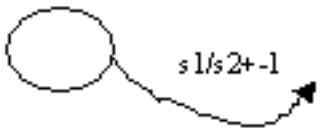
- Dividir para conquistar.
- Algoritmos aleatorizados.
- Programación dinámica.
- Algoritmos golosos (Greedy).
- Algoritmos de heurísticos.
- Reducción a otro problema conocido.
- Uso de estructuras de datos que solucionen el problema.

1.2 Modelos computacionales

Para poder estudiar en detalle un algoritmo debemos fijar un marco en el cual podamos probar y analizar un algoritmo, asi como también que permita comparar dos algoritmos entre sí. Este ambiente necesario para el estudio de los algoritmos se conoce como **"modelo computacional"**.

1.2.1 Máquina de Turing

La máquina de Turing es el mas básico de los modelos computacionales y fue propuesta por el célebre matemático *Alan Turing*. La máquina de Turing maneja tres símbolos que llamaremos "a" , "b" y "espacio". Además cuenta con una memoria infinita que puede verse como una cinta infinita. La máquina es capaz de leer en cada paso un símbolo de la cinta, escribir el mismo u otro símbolo en la cinta y avanzar o retroceder una posición. Por lo tanto podemos representar un programa para la máquina de Turing como un autómata de estados finitos donde cada estado tiene la siguiente notación.



Ver figura 1

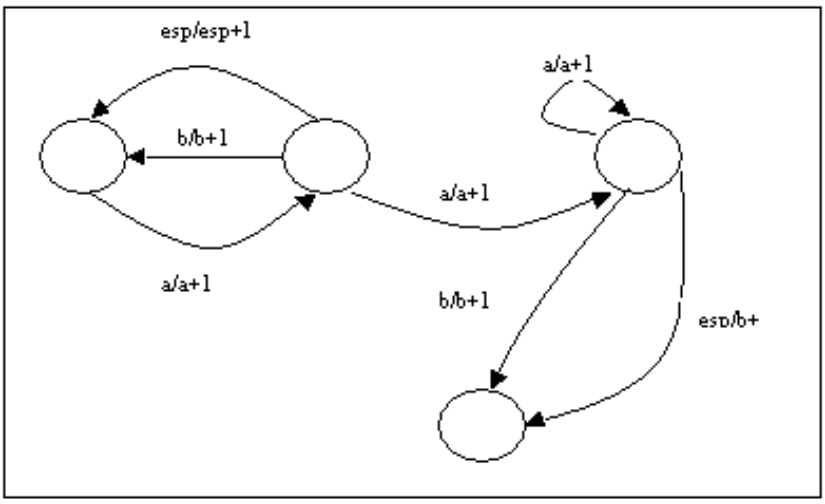
Donde $s1$ es el símbolo que la máquina lee de la cinta "a" , "b" o "λ" , $s2$ es el símbolo que la máquina escribe en la cinta en la misma posición donde leyó $s1$ y el +1 o -1 indica si se avanza o se retrocede una posición en la cinta. La flecha indica cual es el estado hacia el cual se desplaza la máquina, que podría ser el mismo estado donde estaba antes.

Aunque en principio pueda parecer increible cualquier problema que pueda resolver una computadora puede ser resuelto en la máquina de Turing. Esto fue demostrado oportunamente por Alan Turing y no es tema de esta materia.

Importante Cualquier problema que una computadora pueda resolver puede resolverse en una máquina de Turing.

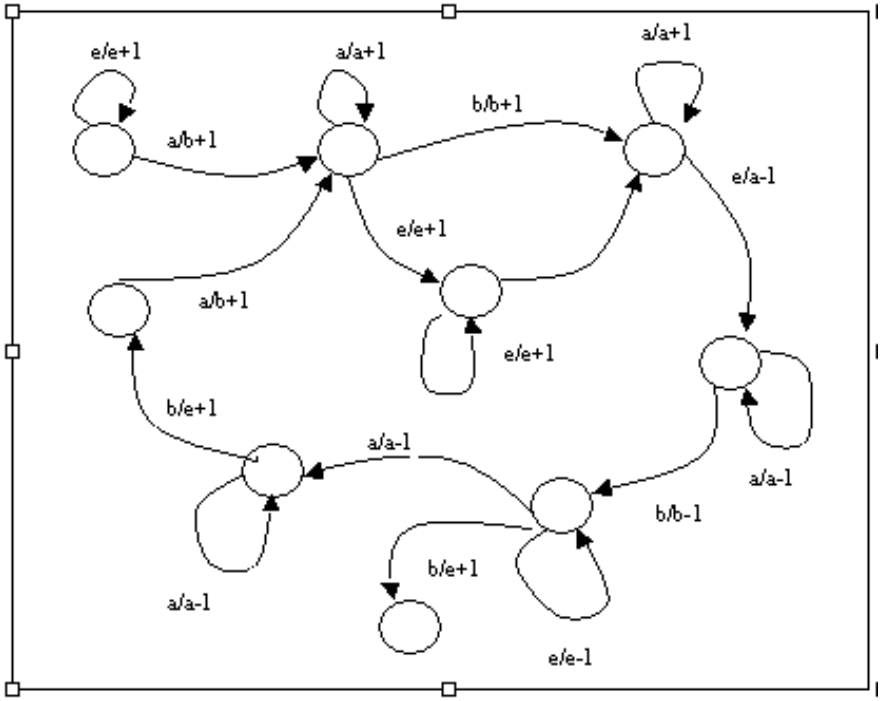
Ejemplo 1 En algun lugar de la cinta se pueden encontrar dos o mas "a" seguidas. Copiar una "b" a continuación de la última "a".

$$Ej : \text{“}\lambda aaa\lambda'' \rightarrow \text{“}\lambda\lambda\lambda aaab''$$



Ejemplo 2 Ejemplo 2: Copiar todas las"a" de la memoria a continuacin de las "b".

$$Ej : \text{“}\lambda\lambda\lambda aaa\lambda\lambda\lambda\lambda bbbbb\lambda'' \rightarrow \text{“}\lambda\lambda\lambda\lambda\lambda\lambda\lambda bbbbbaaa\lambda''$$



Considerando al símbolo “a” como un “1” y al smbolo “b” como un “0” la máquina de Turing es capaz de realizar operaciones aritméticas. Como sumar, restar multiplicar, dividir etc. Si representamos los caracteres de alguna forma binaria la máquina es capaz de trabajar con strings, finalmente podríamos llegar a conjeturar que la máquina es capaz de realizar cualquier operación con una memoria binaria y por ende es capaz de operar como cualquier computadora.

Costos En la máquina de Turing los costos se pueden medir de la siguiente forma:

Costo de tiempo Es la cantidad de transiciones (cambios de estado que realiza la máquina), contando tambien aquellas transiciones que parten de un estado y vuelven al mismo.

Por ejemplo nuestro programa para la mquina de Turing del ejemplo 1. Para el caso: “$aaaaa\lambda''$ insume 5 transiciones y la cinta queda de la forma: “$aaaab''$

Costo de espacio Se puede medir como la cantidad de estados del programa. Cada estado en la máquina de Turing representa una cierta memoria, por lo que medir la cantidad de estados es una buena forma de medir el costo espacial

de un algoritmo. Nuestro primer algoritmo para la máquina de Turing tiene 4 estados.

En la máquina de Turing el análisis de la eficiencia de un algoritmo pasa por averiguar si no existe un programa que pueda resolver el mismo problema en forma mas rápida (empleando menos transiciones) o bien utilizando menos espacio (menor cantidad de estados).

La máquina de Turing ofrece varias ventajas desde el punto de vista del estudio de algoritmos. En primer lugar es un ambiente computacional completo ya que como demostrara Alan Turing cualquier problema computacionalmente solucionable puede resolverse usando una máquina de Turing. En segundo lugar el modelo que plantea es ideal para el cálculo de costos de algoritmos ya sea en tiempo como en espacio. Lamentablemente la máquina de Turing es un modelo poco práctico ya que al ser demasiado abstracta programar resulta extremadamente complejo. Además no es una buena referencia para la obtención de programas "reales" ya que un programa hecho para la máquina de Turing no tiene relación alguna con un programa realizado en un lenguaje de propósito general. Es por esto que destacando la nobleza de la máquina de Turing pasamos a otros modelos un tanto mas cercanos a la realidad.

1.2.2 Maquina RAM de costo fijo

La máquina *RAM* proviene de Random Access Memory. Y es una máquina ideal muy similar a una computadora actual aunque con algunas simplificaciones. Los programas de la máquina *RAM* se almacenan en memoria (en la máquina de Turing los programas no se almacenan [1]. Puede suponerse que todos los accesos a memoria tienen el mismo costo (en tiempo) y que todas las instrucciones tienen un costo constante e idéntico (en tiempo). El set de instrucciones de la mquina RAM está compuesto por la gran mayoría de las instrucciones que podemos encontrar en un lenguaje de alto nivel. Un programa para la máquina RAM se escribe en un pseudocódigo especial en el cual vamos a adoptar algunas convenciones básicas.

Las llaves solo son necesarias si su ausencia afecta la claridad del código. Los pasajes de parámetros a una función se hacen por valor. El acceso a un elemento de un arreglo cuesta lo mismo que el acceso a una variable.

El **algoritmo 2** tiene 5 instrucciones de las cuales se ejecutan unicamente 4. Por lo tanto el costo del programa en tiempo es de $C * 4$, siendo C una constante que indica cuanto tiempo tarda en ejecutarse una instrucción. El espacio que necesita el programa es el espacio coupado por las variables A y B. Es decir $2 * Ec$ siendo Ec el espacio que ocupa una variable.

[1] Por lo menos no se almacenan en la memoria de la máquina(cinta)

Algorithm 1 Este es un ejemplo

$y \Leftarrow 1$ /* Asignacion
$A[2] \Leftarrow 1$ /* Asignacion a un elemento de un vector
/* if-then-else
if $n < 1$ **then**
  $X \Leftarrow w$
else
  $X \Leftarrow y$
end if
/* ciclo while
while $n \geq 0$ **do**
  $n \Leftarrow n - 1$
end while
/* ciclo for
for $i = 0$ to 10 **do**
  $A[i] \Leftarrow 0$
end for

Algorithm 2 Ejemplo simple

$a \Leftarrow 3$
$b \Leftarrow a * 5$
if $b == 5$ **then**
  $a \Leftarrow 2$
else
  $a \Leftarrow 1$
end if

La mquina RAM de costo fijo es mucho mas realista que la máquina de Turing ya que puede verse como, claramente, a partir de un algoritmo escrito para la máquina RAM podemos desarrollar un algoritmo escrito en C , Pascal u otro lenguaje para una computadora actual. Este acercamiento a la realidad en el código de la máquina RAM no se evidencia en cuanto al costo ya que el modelo planteado en el cual todas las instrucciones tienen un costo fijo no es real ya que hay algunas instrucciones que son claramente mas costosas que otras, por ejemplo una multiplicación insume mas tiempo que una suma en cualquier computadora.

1.2.3 Maquina RAM de costo variable

En la máquina RAM de costo variable cada instrucción I_i tiene asociado un costo C_i que le es propio y que depende del costo de implementar dicha instrucción.

Algorithm 3 Ejemplo simple II

$a \Leftarrow 3$
$b \Leftarrow a * 5$
if $b == 5$ **then**
  $a \Leftarrow 2$
else
  $a \Leftarrow 1$
end if

Este programa cuesta ahora

$$3 * C1 + 1 * C2 + 1 * C3$$

```
Donde:
C1 = costo de una asignaci\´{o}n.
C2 = costo de una multiplicaci\´{o}n.
C3 = costo de comparar dos n\´{u}meros.
```

El costo del espacio en la mquina RAM de costo variable es igual al de la máquina RAM de costo fijo.

Claramente observamos que pese a ganar claridad en cuanto a la escritura de programas perdemos precisión y se hace msa complejo el cálculo de costos. Mas adelante veremos algunas herramientas que permitan simplificar el cálculo de dichos costos.

Importante A partir de este momento adoptaremos como base para el análisis de los algoritmos que estudiaremos la máquina RAM de costo variable.

1.3 Un problema básico: elevando un número a una potencia.

Supongamos que estamos utilizando un modelo de máquina *RAM* (de costo variable) en la cual no disponemos de una instrucción que eleve un número a una potencia, como sí disponemos de una instrucción que multiplique a un número por otro podemos escribir el siguiente algoritmo.

Algorithm 4 Potencia(x,y) Calcula x^y

Require: $x > 0$
  if $y == 0$ **then**
    return 1
  else
    $result \Leftarrow x$
    for $I = 1$ to $y - 1$ **do**
      $result \Leftarrow result * x$
    end for
  end if
  return result

Cuánto cuesta el algoritmo en tiempo ?. La comparación del if se ejecuta siempre, y supongamos que y no es cero (lo cual va a pasar la mayoría de las veces). Se ejecuta una asignación y luego $y - 1$ veces una multiplicación.

$$T = C1 + C2 + (y - 1) \cdot (C3 + C1)$$

```
C1 = costo de comparar dos nmeros.
C2 = costo de realizar una asignacion.
C3 = costo de realizar una multiplicacion.
```

Observemos que el algoritmo no es eficiente ya que realiza demasiadas multiplicaciones, por ejemplo si queremos hacer 2^8 el algoritmo calcula.

$2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$

Cuando podría hacer

$$A = 2 * 2(2^2)$$

$$B = A * A(2^4)$$

$$C = B * B(2^8)$$

Con lo cual en lugar de 7 multiplicaciones se utilizan solamente tres. Podemos escribir entonces el siguiente algoritmo.

Algorithm 5 Potencia2(x,y) Calcula x^y

Require: $x > 0$
  if $y == 0$ **then**
    return 1
  end if
  $r \Leftarrow 1$
  if $y > 1$ **then**
    $r \Leftarrow Potencia2(x, y/2)$
    $r \Leftarrow r * r$
  end if
  if $y \bmod 2 == 1$ **then**
    $r \Leftarrow r * x$
  end if
  return r

Veamos que pasa con Potencia2(2,7)

```
r = 1
7 > 1
r = Potencia2(2,3)
  r2 = 1
  3 > 1
  r2 = Potencia2(2,1)
    r3 = 1;
    y <= 1
    y % 2 = 1
    r3 = 1*2
    return 2
  r2 = 2*2
  y % 2 = 1
  r2 = 4 * 2
  return 8;
```

```
r = 8 * 8
y % 2 = 1
r = 64 * 2
return 128;
```

Como podemos ver el algoritmo es recursivo y en nuestro ejemplo para $y = 7$ realizó 5 multiplicaciones. Contra las seis que haríamos en el algoritmo tradicional. Para valores mas grandes de y la diferencia entre el primer y el segundo algoritmo se hace mucho mas notoria.

Calcular el costo de nuestro nuevo algoritmo es uno de los temas a desarrollar en las próximas clases.

1.4 Resumen y aspectos clave

El propósito de esta clase fue introducir el estudio de los algoritmos, explicando en que consiste el análisis y el disenio de los algoritmos. Dentro del análisis de algoritmos explicamos la necesidad de contar con un **modelo computacional** que permita calcular el costo de un algoritmo. Vimos que el costo de un algoritmo se puede calcular en **tiempo** (cuánto tarda el algoritmo) o en **espacio** (cuánta memoria requiere un algoritmo). En general en el curso vamos a trabajar estudiando el costo en tiempo de los algoritmos, vigilando que el costo espacial no sea excesivo.

El disenio de algoritmos es un área que requiere de habilidad especial ya que no siempre es claro como encontrar un algoritmo eficiente para un determinado problema. Mencionamos algunos paradigmas básicos que veremos a lo largo del curso mediante los cuales podremos diseniar mejores algoritmos.

Entre los modelos computacionales revisados la máquina de Turing permite calcular con precisión el costo en tiempo o espacio de un algoritmo y permite tambien resolver cualquier problema, pero no nos permite escribir algoritmos en una forma cercana a la que ofrecen hoy en día los lenguajes de programación. La máquina RAM nos permite una escritura mas clara de los algoritmos pero introduce algunas complicaciones en el cálculo de costos, mas adelante veremos como resolver estos problemas.

Como una introducción al disenio y análisis de algoritmos presentamos el problema de elevar un número a una determinada potencia empleando la menor cantidad de multiplicaciones posibles.

1.5 Ejercicios

1. Realizar en la máquina de Turing un programa que sume dos números binarios almacenados en memoria (suponerlos de igual longitud).

2. Realizar en la máquina de Turing un programa que reste dos números binarios almacenados en memoria (suponerlos de igual longitud).

3. Utilizar el segundo algoritmo de potencia para calcular 2^8, 2^9, 2^{10}, intentar deducir cuantas multiplicaciones realiza el algoritmo en función del número y (Aproximadamente). Probar con mas valores si hace falta.

4. (Para pensar) Cual es la cantidad mínima de multiplicaciones que podemos emplear para calcular X^N, por ejemplo para $X = 11, X = 23, X = 18$ etc. En base a esto es óptimo nuestro segundo algoritmo o puede existir un algoritmo mejor?

5. Escribir una versión iterativa del algoritmo potencia2.

Chapter 2

Algoritmos iterativos

2.1 Algoritmos iterativos

Los algoritmos iterativos son aquellos que se basan en la ejecución de ciclos; que pueden ser de tipo for,while,repeat,etc.La gran mayoría de los algoritmos tienen alguna parte iterativa y muchos son puramente iterativos. Analizar el costo en tiempo de estos algoritmos implica entender cuantas veces se ejecuta cada una de las instrucciones del algoritmo y cual es el costo de cada una de las instrucciones.

2.1.1 Sort por burbujeo

Uno de los temas importantes del curso es el análisis de algoritmos de sort, por lo que vamos a empezar con uno de los mas simples: el burbujeo.

Algorithm 6 Burbujeo(A,n). Ordena el vector A.

```
for i = 1 to n − 1 do
  for j = i + 1 to n do
    if A[j] < A[i] then
      Swap(A[i], A[j])
    end if
  end for
end for
```

```
Ejemplo.

3 4 1 5 2 7 6 4  Compara A[1] con A[2]
3 4 1 5 2 7 6 4  Compara A[1] con A[3]
1 4 3 5 2 7 6 4  Compara A[1] con A[4]
1 4 3 5 2 7 6 4  Compara A[1] con A[5]
1 4 3 5 2 7 6 4  Compara A[1] con A[6]
```

```
1 4 3 5 2 7 6 4  Compara A[1] con A[7]
1 4 3 5 2 7 6 4  Compara A[1] con A[8]
1 4 3 5 2 7 6 4  Compara A[2] con A[3]
1 3 4 5 2 7 6 4  Compara A[2] con A[4]
1 3 4 5 2 7 6 4  Compara A[2] con A[5]
1 2 4 5 3 7 6 4  Compara A[2] con A[5]
1 2 4 5 3 7 6 4  Compara A[2] con A[6]
1 2 4 5 3 7 6 4  Compara A[2] con A[7]
1 2 4 5 3 7 6 4  Compara A[2] con A[8]
1 2 4 5 3 7 6 4  Compara A[3] con A[4]
1 2 4 5 3 7 6 4  Compara A[3] con A[5]
1 2 3 5 4 7 6 4  Compara A[3] con A[6]
1 2 3 5 4 7 6 4  Compara A[3] con A[7]
1 2 3 5 4 7 6 4  Compara A[3] con A[8]
1 2 3 5 4 7 6 4  Compara A[4] con A[5]
1 2 3 4 5 7 6 4  Compara A[4] con A[5]
1 2 3 4 5 7 6 4  Compara A[4] con A[6]
1 2 3 4 5 7 6 4  Compara A[4] con A[7]
1 2 3 4 5 7 6 4  Compara A[4] con A[8]
1 2 3 4 5 7 6 4  Compara A[5] con A[6]
1 2 3 4 5 7 6 4  Compara A[5] con A[7]
1 2 3 4 5 7 6 4  Compara A[5] con A[8]
1 2 3 4 4 7 6 5  Compara A[6] con A[7]
1 2 3 4 4 6 7 5  Compara A[6] con A[8]
1 2 3 4 4 5 7 6  Compara A[7] con A[8]
1 2 3 4 4 5 6 7  FIN.
```

Para un vector de 8 elementos el algoritmo insumió 32 comparaciones. Antes de analizar en forma genérica el costo en tiempo del burbujeo veamos ejemplos mas simples.

Ciclos simples

for $i = 1$ to n **do**
  instruccion
end for

El calculo en costo de un ciclo simple puede hacerse utilizando una sumatoria.

$$\sum_{i=1}^{n} C1$$

Donde C1 es el costo de la instrucción que se ejecuta en el ciclo. El resultado de la sumatoria es $n * C1$, lo cual es evidente porque la instrucción dentro del ciclo se ejecuta n veces.

Ciclos anidados independientes

for $I = 1$ to n **do**
  for $J = 3$ to m **do**
    Instruccion
  end for
end for

Podemos calcular el costo nuevamente usando sumatorias de la forma:

$$\sum_{i=1}^{n}\sum_{j=3}^{m} C1 = \sum_{i=1}^{n} (m-2) * C1 = n * (m-2) * C1$$

Ciclos anidados dependientes Cuando los ciclos dependen uno del otro podemos aplicar la misma tecnica.

for $I = 1$ to n **do**
  for $J = i$ to n **do**
    Instruccion
  end for
end for

$$\sum_{i=1}^{n}\sum_{j=i}^{n} C1 = \sum_{i=1}^{n} (n-i+1) * C1 =$$

$$= C1 * \sum_{i=1}^{n} n - i + 1 = C1 * (\sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1) =$$

$$= C1 * (n * n - \frac{n*(n+1)}{2} + n) = C1 * (n^2 - \frac{n^2+n}{2} + n) =$$

$$= C1 * (\frac{2n^2 - n^2 - n + 2n}{2}) = C1 * \frac{n^2+n}{2}$$

Antes de continuar vamos a recordar algunas sumatorias utiles.

$$\sum_{i=1}^{n} c = c * n$$

$$\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$$

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + 1$$

Análisis del algoritmo de burbujeo Pasemos nuevamente al algoritmo de burbujeo.

```
for i = 1 to n − 1 do
  for j = i + 1 to n do
    if A[j] > A[i] then
      Swap(A[i], A[j])
    end if
  end for
end for
```

El costo del algoritmo lo vamos a calcular suponiendo que C1 es el costo de efectuar la comparación y que C2 es el costo de efectuar el SWAP[1]. Sin embargo el SWAP no se hace siempre sino que se hace unicamente cuando la comparación da $A[j] > A[i]$, por ello debemos particionar el análisis en tres casos: el peor caso, el mejor caso y el caso medio.

Peor caso

En el peor caso el algoritmo siempre hace el Swap.Esto ocurre por ejemplo cuando el vector viene ordenado en el orden inverso al que utilizamos.

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} C1 + C2$$

$$T(n) = \sum_{i=1}^{n-1} (C1 + C2) * (n - 1 - (i + 1) + 1)$$

$$T(n) = \sum_{i=1}^{n-1} (C1 + C2) * (n - 1 - i)$$

$$T(n) = (C1 + C2) * (\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1} i)$$

[1]Para economizar espacio Swap(x,y) puede hacerse como x xor y; y xor x;x xor y. Ver para creer!

$$T(n) = (C1 + C2) * ((n-1) * n - (n-1) - \frac{(n-1) * n}{2})$$

$$T(n) = (C1 + C2) * (n^2 - n - n + 1 - \frac{n^2 - n}{2})$$

$$T(n) = (C1 + C2) * (\frac{1}{2}n^2 - \frac{3}{2}n + 1)$$

Por ejemplo para $N = 8$ el peor caso nos da: $(c1 + c2) * (32 - 12 + 1) = (c1 + c2) * 21$ un total de 20 comparaciones y 20 Swaps.

Mejor caso

En el mejor caso el vector ya viene ordenado por lo que nunca efectua ningn swap, el costo total es entonce el costo de las comparaciones que es igual a lo calculado antes.

$$T(n) = C1 * (\frac{1}{2}n^2 - \frac{3}{2}n + 1)$$

Caso medio

En el caso medio el algoritmo efectua todas las comparaciones y solo la mitad de los Swaps.

$$T(n) = C1 * (\frac{1}{2}n^2 - \frac{3}{2}n + 1) + C2 * \frac{1}{2}(\frac{1}{2}n^2 - \frac{3}{2}n + 1)$$

Importante Al analizar el tiempo de un algoritmo se debe analizar el mejor caso, el peor caso y el caso medio. Habitualmente el que mas interesa es el peor caso.

Otro algoritmo de sort Para continuar practicando el análisis de algoritmos iterativos vamos a analizar un algoritmo de sort completamente distinto del burbujeo: el sort por inserción.

2.1.2 Insert-sort

Quienes hayan jugado alguna vez algún juego de cartas en el cual se juegue con mas de 5 o 6 cartas en la mano habrán observado que algunos jugadores ponen sus cartas boca abajo en la mesa y luego van tomando de a una carta y la colocan en la posición que le corresponde en la mano. Esto es precisamente lo que hace el algoritmo de sort por inserción. [2]

Algorithm 7 InsertSort(A,n). Ordena un vector.

```
for J = 2 to n do
    key ⇐ A[j]
    /* Insertar A[j] en la secuencia ya ordenada A[1..j-1] */
    i ⇐ j − 1
    while i > 0 and A[i] < key do
        A[i + 1] ⇐ A[i]
        i ⇐ i − 1
    end while
    A[i + 1] ⇐ key
end for
```

Veamos el código.

```
Ejemplo.
3 4 1 5 2 7 6 4 j=2, key=4, i=1  no entra al while. A[2]=4.
3 4 1 5 2 7 6 4 j=3, key=1, i=2  entra al while.
A[3]  A[2]
3 4 4 5 2 7 6 4
i=1
A[2]  A[1]
3 3 4 5 2 7 6 4
i=0
sale del while
A[1]1
1 3 4 5 2 7 6 4 j=4, key=5,i=4 no entra al while
1 3 4 5 2 7 6 4 j=5, key=2,i=4 entra al while
A[5]  A[4]
1 3 4 5 5 7 6 4
i=3
A[4]  A[3]
1 3 4 4 5 7 6 4
i=2
A[3]A[2]
1 3 3 4 5 7 6 4
i=1
sale del while
A[2]  2
1 2 3 4 5 7 6 4 j=6,key=7,i=5 no entra al while
1 2 3 4 5 7 6 4 j=7,key=6,i=6 entra al while
A[7]  A[6]
1 2 3 4 5 7 7 4
i=5
```

[2]En particular suelo jugar al bridge, y los jugadores que hacen esto tardan siglos en acomodar sus cartas. Evidentemente no conocen mejores métodos de sort!

```
sale del while
A[6]  6
1 2 3 4 5 6 7 4 j=8,key=4,i=7 entra al while
A[8]  A[7]
1 2 3 4 5 6 7 7
i=6
A[7]  A[6]
1 2 3 4 5 6 6 7
i=5
A[6]  A[5]
1 2 3 4 5 5 6 7
i=4
sale del while
A[5]  4
1 2 3 4 4 5 6 7 fin.
```

Como podemos observar el algoritmo trabaja mejor si el vector ya está ordenado en cuyo caso el ciclo while no se ejecuta. En el peor caso por el contrario el ciclo while se ejecuta $j-1$ veces. Entonces podemos plantear.

Peor caso

```
Sea
C1= Costo de asignar a una variable un elemento de un vector.
C2= Costo de asignar a una variable el valor de otra menos 1.
C3= Costo de asignar a un elemento de un vector el valor de otro
    elemento de un vector.
C4= Costo de decrementar una variable.
C5= Costo de comparar un elemento de un vector con un numero
C6= Costo de comparar dos numeros.
```

$$T(n) = \sum_{j=2}^{n}(C1 + C2 + C5 + C6 + \sum_{i=1}^{j-1}(C3 + C4) + C3)$$

$$T(n) = \sum_{j=2}^{n}(C1 + C2 + C5 + C6 + (C3 + C4)(j-1) + C3)$$

$$T(n) = N * C1 + N * C2 + N * C5 + N * C6 + (C3 + C4) * \sum_{j=2}^{n}(j-1) + N * C3$$

$$T(n) = N(C1 + C2 + C5 + C6 + C3) + (C3 + C4) * (\sum_{j=2}^{n} j - \sum_{j=2}^{n} 1)$$

$$T(n) = N(C1 + C2 + C5 + C6 + C3) + (C3 + C4) * (\frac{(n-1) * n}{2} - (n-1))$$

$$T(n) = N(C1 + C2 + C5 + C6 + C3) + (C3 + C4) * (\frac{1}{2}n^2 - \frac{3}{2}n + 1)$$

Como podemos ver las fórmulas se van haciendo mas complejas debido a que no podemos eliminar las constantes C1..CN que nos impiden calcular con mayor simplicidad el tiempo del algoritmo.

Mejor caso

En el mejor caso el ciclo while no se ejecuta, notemos que para el mejor caso el algoritmo de inserción es mejor que el burbujeo ya aquel de todas formas tiene que efectuar todas las comparaciones.

$$T(n) = \sum_{j=2}^{n}(C1 + C2 + C5 + C6 + C3)$$

$$T(n) = N(C1 + C2 + C5 + C6 + C3)$$

Caso medio

Caso medio.En el caso medio el ciclo while se ejecuta $(j-1)/2$ veces por lo que la fórmula es muy similar a la del peor caso aunque dividiendo por dos los términos que multiplican a $C3$ y $C4$.

2.2 Orden de un algoritmo

El análisis de los algoritmos que hemos realizado hasta aquí es muy preciso pero resulta incómodo para entender que tan eficiente es un algoritmo o para poder compararlo contra otro algoritmo. Habitualmente no interesa cuanto tarda exactamente un algoritmo sino que nos interesa saber cual es la tasa de crecimiento del algoritmo en función de los datos de entrada. Para el estudio de algoritmos existe una notación muy práctica y utilizada universalmente en este tipo de problemas conocida como la gran "O" (Big-Oh notation) que define el orden de un algoritmo.

Importante Orden de un algoritmo: Tasa de crecimiento del tiempo que insume el algoritmo en función de la cantidad o tamao de los datos de entrada.

2.3 Notacion "O"

Esta notación sirve para clasificar el crecimiento de las funciones.

2.3.1 $O(g(n))$:

Sea una funcin $f(n)$, decimos que $f(n)$ pertenece a la familia de funciones $O(g(n))$ si existe una constante $C1$ y un número $n0$ tales que.

$$0 \leq f(n) \leq C1 * g(n) \, para \, n \geq no$$

Es decir que a partir de un cierto n la funcin $f(n)$ queda por debajo de la función $g(n)$ multiplicada por una constante.

2.3.2 $\Omega(g(n))$:

Sea una función $f(n)$, decimos que $f(n)$ pertenece a la familia de funciones $\Omega(g(n))$ si existe una constante $C1$ y un número no tales que.

$$C1 * g(n) \leq f(n) \, para \, n \geq no$$

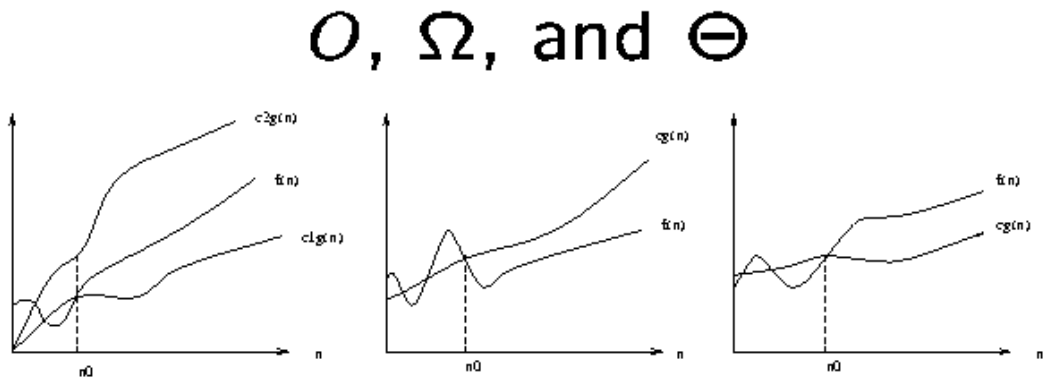
Es decir que a partir de un cierto punto la función siempre es mayor que $g(n)$ multiplicada por una constante.

2.3.3 $\Theta(g(n))$:

Sea una función $f(n)$, decimos que $f(n)$ pertenece a la familia de funciones $\Theta(g(n))$ si existen constantes $C1$ y $C2$ y un número no tales que.

$$C1 * g(n) \leq f(n) \leq C2 * g(n) \, para \, n \geq no$$

Es decir que a partir de un cierto número la función queda atrapada entre $C1 * g(n)$ y $C2 * g(n)$.



Esta notación que pretende clasificar el crecimiento de las funciones es apropiada para el estudio de los algoritmos ya que nos permite realizar algunas simplificaciones. Por ejemplo: $C1 * f(n) = \Theta(f(n))$ Además los sumandos de orden inferior se pueden obviar. $O(f(n)+O(g(n)) = O(f(n))$ sii f(n) crece mas rápido que $g(n)$.

Por ejemplo: $n^2 + n \in \Theta(n^2)$

La notación $\Omega(g(n))$ sirve para acotar el mejor caso de un algoritmo.

La notación $O(g(n))$ sirve para acotar el peor caso de un algoritmo.

Si $\Omega(f(n)) = O(f(n))$ entonces podemos decir que el algoritmo pertenece a $\Theta(f(n))$

Cuando $\Omega(g(n))$ es distinto a $O(g(n))$ debemos usar $O(g(n))$ para expresar el caso medio.

Como un abuso de notación usaremos.

$$f(n) = \Omega(g(n))$$
$$f(n) = O(g(n))$$
$$f(n) = \Theta(g(n))$$

Donde el signo igual debe leerse como "pertenece".

Aplicando esta notación a los algoritmos estudiados podemos ver que el mejor caso del burbujeo es $\Omega(n^2)$. El peor caso es $O(n^2)$ por lo tanto el algoritmo de burbujeo es $\Theta(n^2)$. Es decir que es un algoritmo de orden cuadrático.

El sort por inserción en cambio en el mejor caso es $\Omega(n)$ y en el peor caso es $O(n^2)$. En el caso medio el algoritmo es $O(n^2)$. Por lo que tambien tenemos un algoritmo de orden cuadrático pero que en el mejor de los casos es lineal.

Cuanto mas rápido sea el crecimiento de una función peor ser el rendimiento del algoritmo. Por eso analizando el orden del algoritmo podemos determinar cual es mas eficiente.

Algunas funciones típicas ordenadas por orden de crecimiento.

$$\sqrt{n} < \log n < n < n \log n < n^c < x^n < n! < n^n$$

Luego de analizar el orden del algoritmo de inserción y el burbujeo podemos llegar a la conclusión de que ambos son iguales en eficiencia, aunque como hemos visto trabajan en forma completamente distinta.

2.4 El problema de la distancia mínima en el plano

Un problema a resolver Se tiene un conjunto de n puntos pertenecientes al plano XY, cada punto esta representado por sus coordenadas X e Y. El problema consiste en encontrar el par de puntos tal que la distancia entre ambos puntos es mínima. El algoritmo debe devolver cuales son dichos puntos.

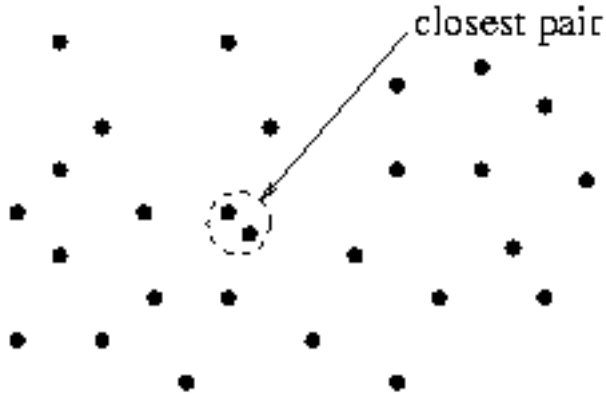


Figure 1: Closest Pair Problem.

2.4.1 Solución iterativa

La solución iterativa a este problema es simple, se recorre el vector de puntos y por cada punto se lo compara contra todos los demas calculando la distancia como $\sqrt{(x1-x2)^2+(y1-y2)^2}$, cada vez que una distancia sea inferior al mínimo anterior actualizamos cuales son los puntos mas cercanos. Cuando ya comparamos todos los puntos contra todos el resultado es un par de puntos cuya distancia seguro que es mínima.

2.4.2 Análisis

Claramente el tiempo de este algoritmo se puede calcular como $\sum_{i=1}^{n}\sum_{j=1}^{n} C$ Siendo C un cierto tiempo que es constante y que es lo que se tarda dentro del ciclo en hacer la comparación entre i y j, en calcular la distancia, realizar la nueva comparación y eventualmente asignar a min la nueva distancia. El resultado de la sumatoria es $\sum_{i=1}^{n} C \cdot \frac{n\cdot(n+1)}{2} = C\cdot n\cdot(n+1) = C(n^2+n) = O(n^2)$ Como ademas es evidente que tanto en el mejor como en el peor caso el algoritmo compara todos los puntos contra todos podemos decir que el algoritmo es $\Theta(n^2)$.

En general los algoritmos iterativos son sencillos y no muy eficientes, como punto de partida o como referencia de comparación siempre es bueno tener

Algorithm 8 Distmin(A,n). Dado un arreglo (A) de n puntos calcula la distancia minima entre dos puntos cualesquiera

```
min ⇐ +MAX
for I = 1 to n do
  for J = 1 to n do
    if i ≠ j then
      dist ⇐ Distancia(A[j], A[i])
      if dist < min then
        min ⇐ dist
        soli ⇐ i
        solj ⇐ j
      end if
    end if
  end for
end for
return (A[soli], A[solj])
```

presente la solución iterativa a un problema. Mas adelante veremos como conociendo otras técnicas de disenio de algoritmos vamos a poder mejorar el orden de este algoritmo para hacerlo mucho mas eficiente.

2.5 Resumen y puntos clave

En esta clase hemos visto cuales son las herramientas utilizadas para analizar algoritmos iterativos, estos son los mas comunes y todos estamos familiarizados con escribir este tipo de algoritmos. Para analizar el costo en tiempo de los algoritmos iterativos vimos que en realidad nos interesa conocer la *tasa de crecimiento u orden* del algoritmo por lo que podemos dejar de lado los factores constantes o de menor crecimiento. Introducimos la notación $O(n)$ que utilizaremos a lo largo de todo el curso para indicar el orden de un algoritmo. Para comparar dos algoritmos utilizaremos el orden de los mismos, cuanto menor sea la tasa de crecimiento mas eficiente será el algoritmo.

Como ejemplos de algoritmos iterativos analizamos dos algoritmos de sort clásicos: el burbujeo y el sort por inserción, vimos que ambos son de orden cuadrático aunque el de inserción en su mejor caso es lineal. El considerar mejor, peor y caso medio es importante en el análisis de los algoritmos.

El problema de hallar los puntos a distancia mínima en el plano es importante porque vamos a ver que hay varias formas de solucionar el problema empleando distintos algoritmos. En esta clase vimos la solución iterativa, de orden cuadrático.

2.6 Ejercicios

1. Analizar el orden del sort por selección.
2. Analizar la versión iterativa del algoritmo Potencia2.
3. Sugerir una mejora al algoritmo de burbujeo y analizar el nuevo algoritmo.
4. Implementar el algoritmo de burbujeo y el insert-sort. Probarlos para conjuntos de datos generados al azar y luego para conjuntos de datos que ya estn ordenados. Realizar una tabla que indique el tamanio de los datos y el tiempo que insumió cada algoritmo en los distintos casos. Probar con conjuntos de datos de distintos tamanios.
5. Listar las siguientes funciones en orden de crecimiento creciente. Indicar cual es el orden de cada una de las funciones.
   (a) n
   (b) 2^n
   (c) n^k con $k > 5$
   (d) $n \log n$
   (e) 192
   (f) $n - n^3 + 7n^5$
   (g) $\log n$
   (h) $\sqrt{n}$
   (i) $\log \log n$
   (j) 1
   (k) n^3
   (l) $(\log n)^2$
   (m) $n!$
   (n) $n^(1 + epsilon)$ con $0 < epsilon < 1$

Lecturas recomendadas En (CLR) se analizar el algoritmo Insert-Sort en la introducción. La notación "Big-Oh" se encuentra definida en el texto de la misma forma en que la definimos aqui. El libro tiene una sección sobre como resolver sumatorias que puede resultar de utilidad.

Chapter 3

Algoritmos Recursivos

3.1 Algoritmos recursivos

Una técnica bastante poderosa a la hora de realizar algoritmos consiste en programar en forma recursiva. Un algoritmo recursivo es aquel que en algún momento durante su ejecución se invoca a si mismo. Por ejemplo el siguiente programa sirve para calcular el factorial de un numero. En general el programar

Algorithm 9 Fact(n). Calcula el factorial de n en forma recursiva.

```
if n < 2 then
  return 1
else
  return n*Fact(n-1)
end if
```

en forma recursiva permite escribir menos código y algunas veces nos permite simplificar un problema difícil de solucionar en forma iterativa. En cuanto a la eficiencia un programa recursivo puede ser mas, menos o igual de eficiente que un programa iterativo. En primer lugar hay que senialar que los programas recursivos consumen espacio en memoria para la pila ya que cada una de las llamadas recursivas al algoritmo son apiladas una sobre otras para preservar el valor de las variables locales y los parámetros utilizados en cada invocación. Este aspecto lo vamos a considerar un costo en **espacio** ya que lo que estamos consumiendo es memoria. En cuanto al tiempo de ejecución del programa debemos calcularlo de alguna forma para poder obtener el **orden** del algoritmo. Para ello se recurre a plantear el tiempo que insume un programa recursivo de la siguiente forma.

Tiempo de ejecución del factorial

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n < 2 \\ 1 + T(n-1) & sino \end{array} \right\}$$

Las ecuaciones planteadas definen una recurrencia. Una recurrencia es un sistema de ecuaciones en donde una o mas de las ecuaciones del sistema estan definidas en función de si mismas. Para calcular el orden de un algoritmo recursivo es necesario resolver la recurrencia que define el algoritmo.

3.2 Recurrencias

Tomemos por ejemplo la siguiente recurrencia muy común en algoritmos recursivos.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 2T(\frac{n}{2}) + n & sino \end{array} \right\}$$

Estudiaremos tres técnicas que suelen utilizarse para resolver recurrencias.

- El método de sustitución
- El método iterativo
- El teorema maestro de las recurrencias.

3.2.1 El método de sustitución

En método de sustitución se utiliza cuando estamos en condiciones de suponer que el resultado de la recurrencia es conocido. En estos casos lo que hacemos es sustituir la solución en la recurrencia y luego demostrar que el resultado es valido por inducción.

Para nuestro ejemplo sabemos que

$$T(n) = (n \log n) + n$$

Veamos que pasa cuando $n = 1$
Si $n = 1 \Rightarrow T(1) = 1$

$$1 \log 1 + 1 = 0 + 1 = 1$$
$$Verifica$$

Hipótesis inductiva
si $n > 1 \Rightarrow T(n') = (n' \log n') + n'$ si $n' < n$
Demostración

$$T(n) = 2T(\frac{n}{2}) + n\,(1)$$

Como $\frac{n}{2} < n$
Por inducción

$$T(\frac{n}{2}) = (\frac{n}{2}) \log (\frac{n}{2}) + \frac{n}{2} \; (2)$$

Reemplazando (2) en (1)

$$\begin{aligned} T(n) = 2((\frac{n}{2}) \log (\frac{n}{2}) + (\frac{n}{2})) + n \\ (n \log \frac{n}{2} + n) + n \\ n \log n - \log 2 + 2n \\ n \log n - n + 2n \\ n \log n + n \\ = \Theta(n \log n) \end{aligned}$$

Por lo que queda demostrado.

El método de sustitución es simple pero requiere que uno ya conozca cual es la solución de la recurrencia. A veces sin embargo en recurrencias muy complejas para resolver por otros métodos puede llegar a ser conveniente intentar adivinar la solucion y luego demostrarla usando este metodo.

3.2.2 El método iterativo

El método iterativo es un forma bastante poderosa de resolver una recurrencia sin conocer previamente el resultado. En este método se itera sobre la recurrencia hasta que se deduce un cierto patrón que permite escribir la recurrencia usando sumatorias. Luego realizando una sustitución apropiada y resolviendo las sumatorias se llega al resultado de la recurrencia.

Utilizando el metodo iterativo en nuestro ejemplo observamos lo siguiente.

$$\begin{aligned} T(1) = 1 \\ T(n) = 2T(\frac{n}{2}) + n \\ T(n) = 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + n + n \\ T(n) = 4 * 2(T(\frac{n}{8}) + \frac{n}{4}) + n) + n = 8T(\frac{n}{8}) + n + n + n \\ T(n) = 2^k * T(\frac{n}{2^k}) + kn \end{aligned}$$

Como $T(1) = 1$ hacemos $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$
Reemplazando

$$T(n) = 2^{\log n} T(\frac{n}{2^{\log n}}) + n \log n$$
$$T(n) = 2^{\log n} T(1) + n \log n$$
$$T(n) = n + n \log n$$
$$T(n) = \Theta(n \log n)$$

Como podemos ver el método iterativo requiere de varios cálculos y puede llegar a requerir de ciertas manipulaciones algebraicas que resultan molestas pero en general puede aplicarse practicamente a cualquier tipo de recurrencia con resultado favorable.[1]

3.2.3 El teorema maestro de las recurrencias

Mediante este teorema podemos contar con una poderosa herramienta para resolver algunas recurrencias. El teorema dice lo siguiente.

Teorema: Sea:
$T(n) = aT(\frac{n}{b}) + n^k$
Si $a \geq 1$ y $b > 1$
Entonces

- Caso 1 Si $a > b^k \Rightarrow T(n) \in \Theta(n^{\log_b a})$
- Caso 2 Si $a = b^k \Rightarrow T(n) \in \Theta(n^k \log n)$
- Caso 3 Si $a < b^k \Rightarrow T(n) \in \Theta(n^k)$

Por ejemplo para $T(n) = 2T(\frac{n}{2}) + n$ tenemos que $a = 2$, $b = 2$, $k = 1$ luego $a = b^k$ y estamos en el caso 2. Por lo que el algoritmo es $\Theta(n^1 \log n)$

Lamentablemente no todas las recurrencias tienen la forma que requiere el teorema maestro por lo que a veces no queda mas remedio que aplicar el metodo iterativo.

3.2.4 Otras herramientas

Además de los tres métodos nombrados que son los básicos en cuanto a la resolución de recurrencias hay otras herramientas que pueden resultar útiles al resolver recurrencias.

[1] Algunas recurrencias en particular no son apropiadas para el método iterativo como veremos luego

Cambio de variables

En algunas recurrencias puede ser conveniente realizar un cambio de variables de forma tal de resolver una recurrencia ya conocida, luego aplicando la inversa del cambio podemos obtener la solución de la recurrencia original.

Supongamos que tenemos la siguiente recurrencia:

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ T(\sqrt{(}n)) + 1 & sino \end{array} \right\}$$

Sea $m \log n$. Entonces $n = 2^m$. Reemplazando 2^m por n en la recurrencia de $T(n)$ tenemos:

$$T(2^m) = T(2^{m/2}) + 1$$

Llamamos $S(m) = T(2^m)$. Entonces la recurrencia queda:

$$S(m) = S(m/2) + 1$$

Usando el teorema maestro sabemos que $S(m) = \Theta(\log m)$. Por lo tanto

$$T(n) = T(2^m) = S(m) = \Theta(lgm)$$

$$T(n) = \Theta(\log \log n)$$

Método de los multiplicadores

El método de los multiplicadores permite resolver recurrencias particularmente dificiles en donde podemos suponer que la forma de la solución se ajusta a $T(n) = \lambda^n$. Esto es válido en recurrencias de la forma $T(n) = a1T(n - x1) + a2T(n - x2) + \ldots + amT(n - xm)$

Sea la recurrencia:

$$T(n) = \left\{ \begin{array}{ll} 0 & si\, n = 0 \\ 4 & si\, n = 1 \\ 8T(n-1) - 15T(n-2) & sino \end{array} \right\}$$

Suponiendo que $T(n) = \lambda^n$ reemplazamos:

$$\lambda^n = 8\lambda^{n-1} - 15\lambda^{n-2}$$

$$\lambda^{n-2}(\lambda^2 - 8\lambda + 15) = 0$$

$$\lambda^{n-2}(\lambda - 5)(\lambda - 3) = 0$$

Como la recurrencia es lineal podemos plantear la forma:

$$T(n) = c_0 0^n + c_1 5^n + c_2 3^n$$

Como c_0 es 0 podemos calcular c_1 y c_2 de $T(0)$ y $T(1)$.

$$T(0) = c_1 5^0 + c_2 3^0$$

$$0 = c1 + c2$$
$$T(1) = c_1 5^1 + c_2 3^1$$
$$4 = 5c_1 + 3c_2$$

Resolviendo el sistema de 2x2 obtenemos $c_1 = 2$ y $c_2 = -2$ por lo que la forma final de la recurrencia es:

$$f(n) = 2 \cdot 5^n - 2 \cdot 3^n$$

3.3 Ejemplos

3.3.1 Ejemplo I. (Messier)

Sea la siguiente recurrencia (Messier)

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 3T(\frac{n}{4}) + n & sino \end{array} \right\}$$

De acuerdo al teorema maestro $a = 3$, $b = 4$, $k = 1$. $a < b^k$. Por lo tanto se aplica el caso 3. $\Theta(n)$

Verifiquemos el resultado del teorema maestro usando el metodo iterativo.

$$T(n) = 3T(\frac{n}{4}) + n$$
$$T(n) = 3(3T(\frac{n}{16}) + \frac{n}{4}) + n = 9T(\frac{n}{16}) + 3(\frac{n}{4}) + n$$
$$T(n) = 9(3T(\frac{n}{64}) + \frac{n}{16}) + 3(\frac{n}{4}) + n = 27T(\frac{n}{64}) + 9(\frac{n}{16}) + 3(\frac{n}{4}) + n$$
$$T(n) = 3^k T(\frac{n}{4^k}) + 3^{k-1}(\frac{n}{4^{k-1}}) + \ldots +$$
$$T(n) = 3^k T(\frac{n}{4^k}) + \sum_{i=0}^{k-1} \frac{3^i}{4^i} n$$

Como $T(1) = 1$ hacemos $\frac{n}{4^k} = 1 \Rightarrow n = 4^k \Rightarrow k = \log_4 n$
Reemplazando

$$T(n) = 3^{\log_4 n} T(1) + \sum_{i=0}^{\log_4 n - 1} \frac{3^i}{4^i} n$$
$$T(n) = n^{\log_4 3} + \sum_{i=0}^{\log_4 n - 1} \frac{3^i}{4^i} n$$

$$
\begin{aligned}
T(n) &= n^{\log_4 3} + n \sum_{i=o}^{\log_4 n - 1} (\frac{3}{4})^i \\
T(n) &= n^{\log_4 3} + n * \frac{(\frac{3}{4})^{\log_4 n} - 1}{(\frac{3}{4}) - 1} \\
\Rightarrow (\frac{3}{4})^{\log_4 n} &= n^{log_4 \frac{3}{4}} = n^{log_4 3 - log_4 4} \\
\Rightarrow &= n^{log_4 3 - 1} = \frac{n^{log_4 3}}{n} \\
T(n) &= n^{\log_4 3} + n * (\frac{\frac{n^{\log_4 3}}{n} - 1}{(\frac{3}{4}) - 1}) \\
T(n) &= n^{\log_4 3} + \frac{n \log_4 3 - 1}{-\frac{1}{4}} \\
T(n) &= n^{\log_4 3} - 4(n^{log_4 3}) \\
T(n) &= 4n - 3n^{log_4 3} \\
T(n) &= 4n - 3n^{0.79} \\
T(n) &= \Theta(n)
\end{aligned}
\tag{3.1}
$$

Como vemos la solución final tiene un término que es $\Theta(n)$ y otro que es $\Theta(n^k)$, cuando $k < 1$ ocurre que la función lineal crece mas rápido que la exponencial y por eso la recurrencia pertenece a $\Theta(n)$.

3.3.2 Ejemplo II

Sea la siguiente recurrencia.

$$
T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 2T(\frac{n}{2}) + n \log n & sino \end{array} \right\}
$$

Esta recurrencia también es bastante común en algunos algoritmos recursivos y lamentablemente no podemos escribirla en la forma en la cual vale el teorema maestro. Como queremos averiguar la solución tenemos que aplicar el método iterativo.

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n \log n \\
T(n) &= 2T(2T(\frac{n}{4}) + (\frac{n}{2}) \log \frac{n}{2}) + n \log n \\
T(n) &= 4T(\frac{n}{4}) + n \log \frac{n}{2} + n \log n \\
T(n) &= 4(2T(\frac{n}{8}) + (\frac{n}{4}) \log \frac{n}{4}) + n \log \frac{n}{2} + n \log n
\end{aligned}
$$

$$T(n) = 8T(\frac{n}{8}) + (n\log\frac{n}{4} + n\log\frac{n}{2} + n\log n)$$

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

Como $T(1) = 1 \Rightarrow \frac{n}{2^k} = 1 \Rightarrow k = \log n$
Reemplazando

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

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

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

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

$$T(n) = n + n(\log^2 n - (\frac{\log n(\log n - 1)}{2}))$$

$$T(n) = n + n(\log^2 n - (\frac{\log^2 n - \log n}{2}))$$

$$T(n) = n + n(\frac{log^2 n - \log n}{2})$$

$$T(n) = n + \frac{1}{2}n\log^2 n - \frac{1}{2}n\log n$$

$$T(n) = \Theta(n\log^2 n)$$

En este caso el termino de mayor crecimiento es $n\log^2 n$.

3.4 Cálculo de números de Fibonacci

La serie de Fibonacci es aquella en la cual $Fib(1) = 1$, $Fib(2) = 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$ para todo n natural. La serie tiene la siguiente forma:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

El problema consiste en hallar un algoritmo eficiente que permita calcular $Fib(n)$.

3.4.1 Solución recursiva

La solución recursiva para encontrar $Fib(n)$ es estremadamente sencilla.

Algorithm 10 Fib1(n). Calcula el 'n'ésimo numero de la serie de Fibonacci.

if $n < 3$ **then**
  return 1
else
  return $Fib(n-1) + Fib(n-2)$
end if

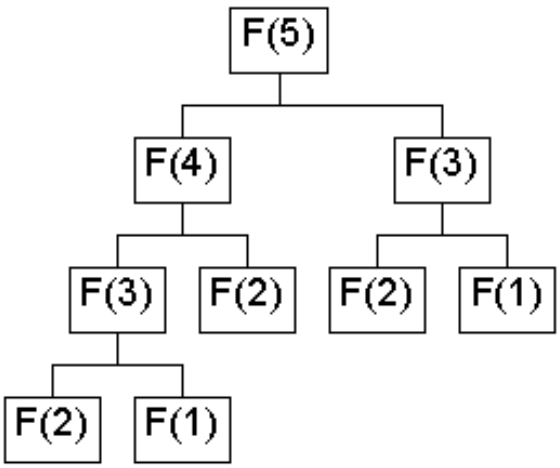
Para calcular el costo de este algoritmo planteamos la siguiente recurrencia.

$$T(n) = \left\{ \begin{array}{cc} 1 & si\ n < 3 \\ T(n-1) + T(n-2) & sino \end{array} \right\}$$

Aplicando el método iterativo obtenemos lo siguiente.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ T(n) = T(n-2) + T(n-3) + T(n-3) + T(n-4) &= T(n-2) + 2T(n-3) + T(n-4) \\ T(n) &= T(n-3) + T(n-4) + 2T(n-4) + 2T(n-5) + T(n-5) + T(n-6) \\ T(n) &= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) \\ T(n) &= T(n-4) + 4T(n-5) + 6T(n-6) + 4T(n-7) + T(n-8) \\ &\vdots \end{aligned}$$

La expansión de la recurrencia tiene la forma del binomio de newton. Y no parece haber una sumatoria conocida que nos permita expresar la recurrencia de forma tal de poder resolverla. En estos casos en los cuales la recurrencia resulta dificil de resolver por los metodos vistos puede ser útil utilizar un árbol para visualizar como funciona el algoritmo.



Como vemos en la figura para calcular Fib(5), necesitamos hacer Fib(4) y Fib(3), luego para Fib(4) necesitamos Fib(3) y Fib(2). Etc.... Los nodos internos del árbol insumen dos instrucciones (la comparación y la suma), mientras que las hojas del árbol solo insumen una instruccion. Por lo tanto la cantidad de instrucciones que se ejecutan para Fib(n) es.

$$T(n) = 2 * nodos + hojas$$

Observando el árbol podemos ver que cada nodo indica la cantidad de hojas debajo de dicho nodo, por eso para calcular Fib(n) necesitamos un árbol con Fib(n) hojas. Además podemos usar una formula útil para árboles binarios que indica que la cantidad de nodos internos es igual a la cantidad de hojas menos 1. (Esto se puede probar facilmente por inducción). Entonces.

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

$$T(n) = 3(Fib(n)) - 2$$

$$T(n) = \Theta(Fib(n))$$

Como se puede ver obtenemos que el orden de nuestro algoritmo es el orden de la serie de Fibonacci. Para saber cual es el orden de la serie de Fibonacci podemos usar la siguiente propiedad.

$$Fib(n) = \Phi^n$$

$$Con\,\Phi = \frac{\sqrt{5} - 1}{2} = 0,61803\ldots$$

Entonces.

$$T(n) = \Theta(\Phi^n) = \Theta(x^n)$$

Como podemos ver el algoritmo pertenece al orden exponencial, por lo tanto es bastante lento. Para $n = 45$ por ejemplo ejecuta mil millones de instrucciones!.

Una forma de ver que nuestro algoritmo es ineficiente es observando como para calcular $F(5)$ calculamos varias veces $F(1)$,$F(2)$,etc. Cuando en realidad solo necesitariamos hacerlo una vez. En este caso el algoritmo recursivo es claramente ineficiente por lo que planteamos una versión iterativa.

3.4.2 Solucion iterativa

Para la solución iterativa usamos un vector de $n + 1$ posiciones e inicializamos $v[1] = 1$,$v[2] = 1$. Luego con un ciclo simple calculamos cada posición del vector como la suma de las dos anteriores hasta llegar a la posición n que es la que buscamos.

Algorithm 11 Fib2(n). Calcula el 'n'ésimo numero de Fibonacci

```
/* Usamos un vector de tamanio n+1 llamado Fib[n+1] */
int Fib[n+1]
Fib[1] ⇐ Fib[2] ⇐ 1
for i = 3 to n do
   Fib[i] ⇐ Fib[i − 1] + Fib[i − 2]
end for
return Fib[n]
```

Como podemos ver claramente este algoritmo es $\Theta(n)$ lo cual implica una gran mejora con respecto a la versión recursiva. La técnica que usamos es sencilla, pero tiene el problema de requerir de un vector de tamanio $n+1$ para poder funcionar. Es decir que hemos reducido el costo en tiempo pero hemos aumentado el costo en espacio. En nuestra próxima versión tratamos de aplicar la misma técnica pero sin utilizar el vector.

Algorithm 12 Fib3(n). Calcula el n'ésimo numero de Fibonacci

```
a ⇐ 1
b ⇐ 1
for i = 3 to n do
   c ⇐ a + b
   a ⇐ b
   b ⇐ c
end for
return a
```

Esta nueva versión sigue siendo $\Theta(n)$ pero en lugar de utilizar un vector de n+1 posiciones usamos tres variables a,b,c. El algoritmo insume mas instrucciones que la versión que usaba el vector pero sigue siendo lineal por lo que asintoticamente es equivalente a Fib2 pero con un costo espacial mucho menor.

3.4.3 Una solución mejor

Una de las curiosidades de la serie de Fibonacci es que existen varias formas curiosas de calcular un número de Fibonacci. Una de las formas es sumando los dos números anteriores de la serie que es el método que estuvimos usando hasta ahora tanto en la versión recursiva como en la versión iterativa. Otra forma es calculando Φ^n. Esta forma tiene la dificultad de que Φ, el número áureo, es irracional lo cual nos obligaría a calcular Fib(n) en forma aproximada. Otra forma de calcular Fib(n) utilizando potencias es usando la siguiente matriz.

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

La matriz M tiene una curiosa propiedad, cuando hacemos $M^{n-1}\ M[0,0]$ es igual a Fib(n).

Para utilizar este nuevo método tenemos que disponer de un algoritmo veloz para elevar una matriz a una potencia. En la primera clase del curso habíamos obtenido el siguiente algoritmo para elevar un número a una potencia.

Algorithm 13 Potencia2(x,y) Calcula x^y

Require: $x > 0$
  if $y == 0$ **then**
    return 1
  end if
  $r \Leftarrow 1$
  if $y > 1$ **then**
    $r \Leftarrow Potencia2(x, y/2)$
    $r \Leftarrow r * r$
  end if
  if $y \bmod 2 == 1$ **then**
    $r \Leftarrow r * x$
  end if
  return r

Claramente el algoritmo se puede adpatar facilmente para elevar una matriz en lugar de un número a cualquier potencia. Este es un algoritmo recursivo definido por la recurrencia.

$$\begin{cases} T(n) = 1 & si\ n = 0 \\ T(n) = T(\frac{n}{2}) + 1 & sino \end{cases}$$

No hace falta tener en cuenta para expresar la recurrencia que el costo difiere agregando una multiplicación o no segun n sea par o impar. Aplicando el teorema maestro (caso 2). Tenemos que $T(n) = \Theta(\log n)$. Por lo tanto sabemos que disponemos de un algoritmo que calcula una potencia de orden logaritmico (que crece menos que una función lineal). Por lo que podemos mejorar el cálculo de los numeros de Fibonacci de la siguiente forma.

Algorithm 14 Fib3(n) Calcula el 'n'esimo numero de Fibonacci

  Potencia2(M,n-1)
  return M[0,0]

Y como sabemos este algoritmo es de orden logaritmico.

3.4.4 Conclusiones

Como hemos visto calcular un número de Fibonacci no es tan sencillo como parece, la versión recursiva del algoritmo era de orden exponencial, luego pudimos construir una versión iterativa de orden lineal y finalmente una version de orden logaritmico. En general se requiere de mucho análisis, esfuerzo e investigación para encontrar la forma mas eficiente de resolver un determinado problema.

3.5 Resumen y puntos clave

El proposito de esta clase fue presentar las herramientas necesarias para el estudio de algoritmos recursivos. Vimos como el costo en tiempo de un algoritmo recursivo puede expresarse mediante una *recurrencia*. Para resolver recurrencias explicamos tres técnicas.

El *método de sustitución* permite demostrar la solución de una recurrencia aplicando induccí'on. Esto es útil cuando ya conocemos la solución o cuando estamos en condiciones de adivinarla.

El *método iterativo* nos permite reducir una recurrencia a una serie de sumatorias que luego resolvemos para resolver la recurrencia.

El *teorema maestro de las recurrencias* nos porporciona una "receta" para resolver recurrencias que responden a una forma determinada.

Por último estudiamos el problema del cálculo de un número de Fibonacci, y vimos como mediante distintas aproximaciones pudimos mejorar nuestro algoritmo inicial que era muy ineficiente hasta encontrar un algoritmo muy bueno. Este proceso de *refinamiento* de un algoritmo es muy importante en el disenio de algoritmos para llegar a una solución óptima.

3.6 Ejercicios

1. Demostrar usando el método de sustitución que la recurrencia

$$T(n) = \left\{ \begin{array}{cc} 1 & si\ n = 1 \\ T(\frac{n}{5}) + T(\frac{3n}{4}) + n & sino \end{array} \right\}$$

   Es $\Theta(n)$

2. Resolver las siguientes recurrencia usando el método iterativo y aplicando el teorema maestro cuando corresponda. Para todas las recurrencias vale que $T(n) = 1$ si $n = 1$.

   (a) $T(n) = 3T(\frac{n}{2}) + n^2$

(b) $[T(n) = T(n-1) + (\frac{1}{n})$

(c) $[T(n) = \sqrt{n} + n$

(d) $[T(n) = 2T(\frac{n}{2}) + (3n - 1)$

Chapter 4

Divide & Conquer

4.1 Divide & Conquer

En este módulo vamos a estudiar una técnica muy poderosa para el disenio de algoritmos conocida por la frase "Dividir para conquistar". Los algoritmos de este tipo se caracterizan por estar diseniados siguiendo estrictamente las siguientes fases:

- Dividir Se divide el problema en partes mas pequenias.
- Conquistar Se resuelven recursivamente los problemas mas chicos.
- Combinar Los problemas mas chicos se combinan para resolver el grande.

Los algoritmos que utilizan este principio son en la mayoría de los casos netamente recursivos. La técnica es sumamente útil, hay muchos problemas para los cuales aplicar un algoritmo de tipo Divide & Conquer permite resolver el problema en forma mas sencilla y eficiente.

4.2 Merge Sort

El algoritmo de Merge Sort es un ejemplo clásico de algoritmo que utiliza el principio de dividir para conquistar. El algoritmo permite ordenar un vector en forma mas eficiente que los méodos de sort vistos hasta el momento como el burbujeo y el sort por inserción. Si el vector tiene mas de dos elementos se lo divide en dos mitades, se invoca recursivamente al algoritmo y luego se hace un merge de las dos mitades ordenadas.

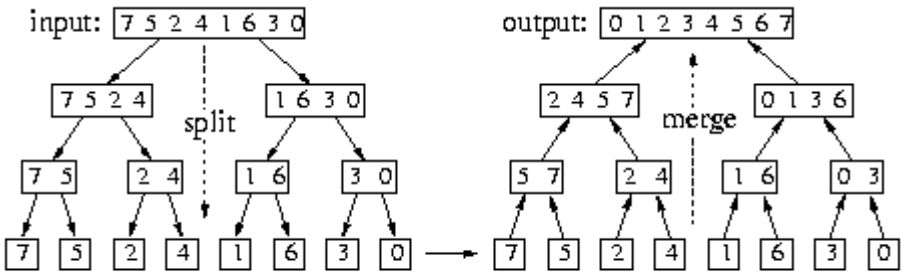


Figure 1: MergeSort.

Algorithm 15 MergeSort(A,p,r). Ordena un vector desde p hasta r.

---

if $p < r$ **then**
  $q \Leftarrow (p + r)/2$
  MergeSort(A,p,q)
  MergeSort(A,q+1,r)
  Merge(A,p,q,r)
end if

El tiempo que insume este algoritmo queda definido por la siguiente recurrencia.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 2T(\frac{n}{2}) + \Theta(Merge) & sino \end{array} \right\}$$

Para calcular la recurrencia necesitamos el algoritmo que realiza el merge.

Análisis del algoritmo Merge Puede verse claramente que el algoritmo de merge requiere de dos pasadas a los vectores a mergear, una para realizar el merge y otra para copiar del vector auxiliar al vector original. El tiempo del algoritmo de merge es $T(n) = 2n$ por lo tanto.

$$Merge : T(n) = \Theta(n)$$

. Por lo que la recurrencia del algoritmo de MergeSort es:

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 2T(\frac{n}{2}) + n & sino \end{array} \right\}$$

Esta es una recurrencia conocida. Aplicando el teorema maestro podemos obtener que $MergeSort = \theta(n \log n)$ Lo cual mejora el orden de los algoritmos de sort que habiamos estudiado que es cuadrático.

Algorithm 16 Merge(A,p,q,r). Mergea A[p,q] .. A[q+1,r]

```
/* Usamos el array auxiliar B[p,r] */
int B[p,r]
i ⇐ k ⇐ p
j ⇐ q + 1
while i ≤ q and j ≤ r do
  if A[i] ≤ A[j] then
    B[k] ⇐ A[i]
    i ⇐ i + 1
  else
    B[k] ⇐ A[j]
    j ⇐ j + 1
  end if
  k ⇐ k + 1
end while
/* Copiamos elementos que quedaban de la parte izquierda */
while i ≤ q do
  B[k] ⇐ A[i]
  i ⇐ i + 1
  k ⇐ k + 1
end while
/* Copiamos elementos que quedaban de la parte derecha */
while j ≤ r do
  B[k] ⇐ A[j]
  j ⇐ j + 1
  k ⇐ k + 1
end while/* Copiamos desde el vector auxiliar */
for i = p to r do
  A[i] ⇐ B[i]
end for
```

4.2.1 Propiedades

El algoritmo de MergeSort es un algoritmo de sort que presenta algunas propiedades interesantes. En primer lugar el orden del algoritmo es $\Theta(n \log n)$, y esto ocurre tanto en el peor caso, como en el mejor caso, como en el caso medio ya que el tiempo que insume el algoritmo no depende de la disposición inicial de los elementos del vector. En segundo lugar es un algoritmo que requiere de un espacio extra de almacenamiento para poder funcionar. A los algoritmos que realizan el sort dentro del vector mismo se los denomina algoritmos de sort "in-situ". Claramente el algoritmo de MergeSort no pertenece a esta familia.

4.2.2 Optimizaciones y trucos

El orden del algoritmo de MergeSort no se puede mejorar, lo que si podemos hacer es aplicar algunas mejorar para que los factores constantes que inciden en el tiempo que insume el algoritmo sean menores. Uno de los trucos mas utilizados consiste en evitar el tener que copiar todo el vector auxiliar en el original cada vez que se hace el procedimiento de Merge. Esto se logra utilizando dos vectores de tamanio 'n'. En los niveles pares de la recursión Mergeamos desde A hacia B. En los niveles impares mergeamos desde B hacia A. Si la cantidad de niveles es en total impar hacemos una copia de mas para volver a tener los datos en el vector original pero lo que se ahorra es siempre mas de este gasto extra.

Otra mejora habitual consiste en no aplicar MergeSort si el tamanio del vector es mas chico que un número constante de elementos, por ejemplo 20, en ese caso se aplica un sort por inserción o cualquier otro. Como la cantidad de elementos es constante esto solo agrega una cantidad constante de tiempo al algoritmo y en general permite obtener una mejora en el rendimiento.

Un detalle importante del algoritmo de MergeSort es el siguiente: En el proceso de Merge debido a comparamos $A[i] \leq A[j]$ tenemos como consecuencia que cuando los elementos son iguales copiamos desde la lista izquierda primero y luego desde la derecha. Podríamos haber copiado desde la derecha primero y el resultado sería el mismo ya que si los números o lo que sea que estamos ordenando son iguales o tienen la misma clave no importa en que orden aparecen en el vector ordenado. Sin embargo frecuentemente ordenamos un conjunto de datos por mas de una clave, si los elementos del vector ya estaban ordenados por alguna otra clave es importante que al ordenarlo por otra clave cuando la nueva clave es igual mantengamos el orden de la clave anterior. Esto se logra precisamente copiando primero de la parte izquierda del vector al mergear. A los algoritmos de sort que preservan el orden original de los elementos que tienen claves iguales se los denomina **estables**. La estabilidad es una propiedad bastante deseable en los algoritmos de sort.

Por ultimo es conveniente analizar si es posible implementar el algoritmo de MergeSort 'in-situ', es decir sin utilizar espacio auxiliar de memoria. La

respuesta es que si es posible pero nadie sabe como hacerlo sin destruir por completo la eficiencia del algoritmo.

4.2.3 Resumen MergeSort

- $T(n) = \Theta(n \log n)$. Mejor, peor y caso medio son iguales.
- Es estable.
- Requiere de un vector auxiliar para realizar el sort.

4.3 El problema de la selección

Supongamos que tenemos un vector A de tamanio n originalmente desordenado. Definimos el *rango* de un elemento del vector como la posición que el elemento ocuparía si el vector estuviese ordenado. De esta forma el elemento mas chico tiene rango 1, el segundo rango 2, etc. Aunque no afecta en absoluto la resolución del problema supongamos tambien que todos los elementos del vector son distintos entre si. El problema de la selección consiste en encontrar el elemento cuyo rango es k. En particular a veces interesa obtener el elemento cuyo rango es $k = \frac{n}{2}$ que es la *mediana* del conjunto de datos. A este problema se lo suele llamar tambien problema de la mediana.

El problema de la selección puede resolverse en forma super-sencilla ordenando el vector A y devolviendo $A[k]$. Como vimos hasta el momento esto puede hacerse en $\Theta(n \log n)$. Sin embargo podemos conjeturar que ordenar el vector no es del todo eficiente porque obtenemos todos los rangos cuando en realidad solo nos interesa uno en particular. Intuitivamente la solución usando sort es ineficiente. Queremos encontrar un algoritmo que nos permita encontrar el elemento cuyo rango es k en tiempo menor que $\Theta(n \log n)$. Como veremos la solución a este problema no es para nada trivial.

4.3.1 *Cribas...*

Hemos introducido este problema debido a que presenta un caso muy particular dentro de los algoritmos de tipo Divide and Conquer. En los algoritmos Divide and Conquer dividimos el problema en m problemas mas chicos para luego resolverlos y combinar las soluciones de ambos. Una criba se presenta cuando $m = 1$. Las cribas se aplican a problemas en los cuales estamos interesados en obtener un elemento determinado dentro de un conjunto de elementos de tamanio n. Este tipo de algoritmos trabaja eliminando un cierto conjunto de elementos del conjunto sabiendo que ninguno de ellos es el elemento buscado, el algoritmo se aplica recursivamente hasta que queda un único elemento en el conjunto: el elemento buscado.

El algoritmo que vamos a describir funciona de la siguiente manera. Recibe un vector A,dos numeros p y r que indican desde donde hasta donde buscar en el vector (inicialmente $p = 1$ y $r = n$) y un numero k que es el rango que queremos encontrar en el vector. El algoritmo selecciona un elemento del vector que se denomina **pivot (p)**, decidir cual es el pivot es importante, pero por el momento supongamos que el pivot es elegido de alguna manera. Luego particiona el vector en tres partes: A[p..q-1] contiene todos los elementos menores que el pivot. A[q] es el pivot. A[q+1..r] contiene todos los elementos mayores que el pivot. Dentro de los vectores A[p..q-1] y A[q+1..p] los elementos pueden estar en cualquier orden. El ranking del pivot en cualquier fase del algoritmo es $q - p + 1$, si $k = q - p + 1$ entonces el pivot es el elemento buscado. Si $k < q - p + 1$ aplicamos el algoritmo recursivamente a $A[p..q-1]$ si $k > q-p+1$ aplicamos el algoritmo recursivamente a $A[q+1..p]$, esto queda ejemplificado en la ilustración.

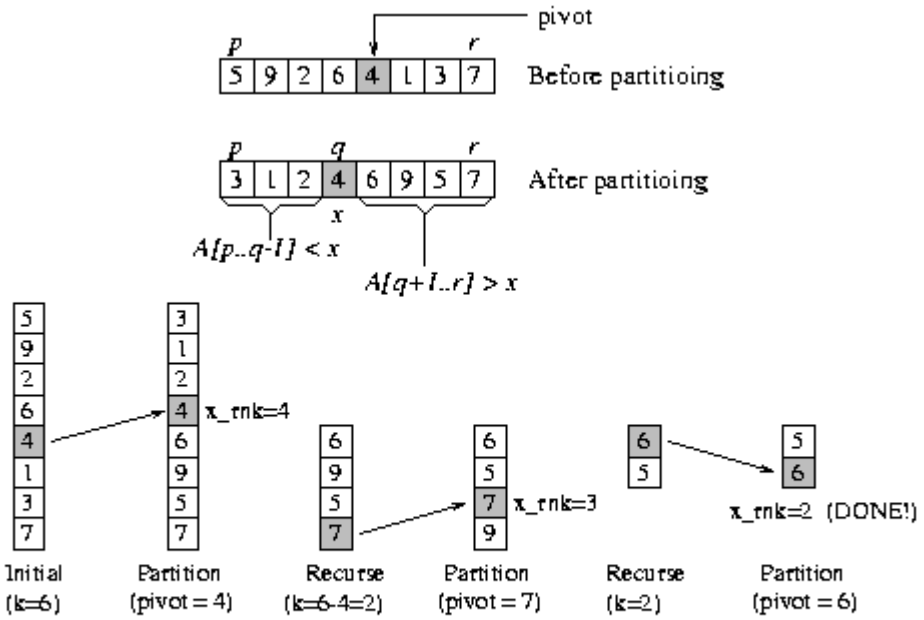


Figure 7: Selection Algorithm.

Notemos que el algoritmo satisface los principios que enunciabamos sobre una criba, elimina ciertos elementos y luego se invoca recursivamente con el resto. Cuando el pivot es el buscado tenemos suerte y eliminamos todos los demás elementos, sino seguimos buscando. Tenemos aún que resolver el problema de

Algorithm 17 Seleccion1(A,p,r,k). Selecciona de A[p..r] el 'k'esimo elemento

```
if p = r then
  return A[p]
else
  x ⇐ ElegirPivot(A, p, r)
  q ⇐ Particionar(A, p, r, x)
  x_rank ⇐ q − p + q
  if k = x_rank then
    return x
  else
    if k < x_rank then
      return Seleccion1(A,p,q-1,k)
    else
      return Seleccion1(A,q+1,r,k-q)
    end if
  end if
end if
```

seleccionar el pivote y ademas el de particionar el vector. Supongamos por el momento que ambas cosas pueden hacerse en $\Theta(n)$. Nos queda por averiguar que tan eficiente es nuestra criba, es decir cuantos elementos eliminamos cada vez. Si el pivote es el elemento mas chico o mas grande del vector eliminamos solamente un elemento, el mejor caso ocurre cuando el pivot es un elemento muy cercano al medio en cuyo caso eliminamos la mitad del vector y podemos escribir la siguiente recurrencia.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ T(\frac{n}{2}) + n & sino \end{array} \right\}$$

Resolviendo por el teorema mestro tenemos que el algoritmo es $\Omega(n)$, es decir que en el mejor caso insume una cantidad de tiempo lineal. Si el pivot que elegimos es malo de todas formas el algoritmo se va a mantener dentro del orden lineal, por ejemplo.

$$T(n) = 1\, si\, n = 1$$

$$T(n) = T(\frac{99n}{100}) + n\, sino$$

Tambien da como resultado $\Omega(n)$, por lo tanto podemos decir que el algoritmo es $\Theta(n)$

4.3.2 Eligiendo el pivote

Como vimos aun eligiendo el pivote malamente el algoritmo tiende a ser lineal, sin embargo cuanto mejor elijamos el pivote mejor va a funcionar. Otra de las cosas que sabemos es que el pivote debe ser elegido en $\Theta(n)$ para que valga la recurrencia que habíamos planteado anteriormente.

Como garantizar un buen pivote Uno de los problemas mas duros en el algoritmo de selección consiste en encontrar un pivote de forma tal que podamos asegurar que el pivote seleccionado siempre funcionará relativamente bien. Floyd, Tarjan, Pratt y Rivest encontraron un algoritmo que garantiza que $n/4 \leq q \leq 3n/4$, lo cual es mas que suficiente para aseverar la validez del pivote.

El algoritmo de Floyd-Pratt-Rivest-Tarjan

Primer paso: dividir en grupos de 5 En primer lugar se particiona el vector en grupos de 5 elementos. Por ejemplo A[1..5],A[6..10],A[11..15],etc. La cantidad de grupos es $m = \lceil \frac{n}{5} \rceil$ Todos los grupos tendran 5 elementos salvo el último que puede tener menos. Esto puede hacerse facilmente en tiempo $\Theta(n)$.

Segundo paso: computar la mediana de cada grupo En segundo lugar se computa la mediana de cada grupo. Como todos los grupos tienen tamanio constante=5 no necesitamos un algoritmo eficiente para encontrar la mediana, podemos aplicar burbujeo a cada grupo y luego elegir el elemento del medio de cada vector ordenado. Esto tarda una cantidad de tiempo constante $\Theta(1)$ y se repite $m = \lceil \frac{n}{5} \rceil$ veces, por lo tanto este paso insume tiempo $\Theta(n)$. Una vez calculadas las m medianas se las copia a un vector auxiliar B.

Tercer paso: Hallar la mediana de las medianas El tercer paso consiste en hallar la mediana de las m medianas anteriores, para esto tenemos que llamar al algoritmo $Seleccion1$ en forma recursiva de la forma $Seleccion1(B, 1, m, k)$ donde $m = \lceil \frac{n}{5} \rceil$ y $k = \lfloor \frac{m+1}{2} \rfloor$. El elemento resultante: la mediana de medianas es el que se elige como pivot. Debemos demostrar ahora que el pivot no deja menos de n/4 elementos a izquierda o derecha de el.

Teorema El rango del pivote x es por lo menos $n/4$ o a lo sumo $3n/4$ en el vector A

Demostración Vamos a probar que el rango de x es por lo menos $n/4$, la otra parte es exactamente simétrica. Para demostrar esto tenemos que probar que hay al menos $n/4$ elementos menores que x. Para evitar problemas causados por las funciones piso y techo supongamos que n el tamanio del vector es multiplo de 5 de forma tal que $m = n/5$. Debido a que x es la mediana de las medianas podemos afirmar que por lo menos la mitad de las medianas de los grupos son menores o iguales que x. Y en cada grupo de medianas (de tamanio 5) hay tres elementos que son menores o iguales a su mediana. Por lo tanto hay $3((n/5)/2) = 3n/10 \geq n/4$ elementos que son menores o iguales a x en el vector. La ilustración permite entender esta demostración en forma mas simple.

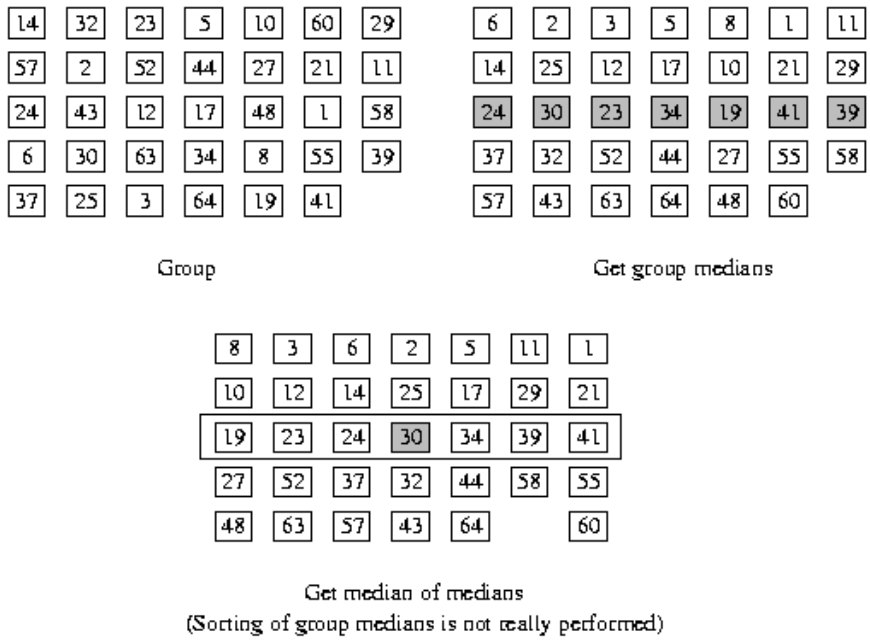


Figure 8: Choosing the Pivot. 30 is the final pivot.

Análisis de algoritmo Aplicando el algoritmo de Floyd-Rives-Tarjan-Pratt para calcular el pivote sabemos que por lo menos eliminamos a 1/4 de los elementos en cada llamada, sin embargo tenemos que recalcular el orden del algoritmo. La llamada recursiva a $Seleccion1(\ldots)$ se hace para no mas de 3n/4 elementos, pero para lograr esto dentro de $ElegirPivote$ tenemos que volver a llamar a $Seleccion1(\ldots)$ para el vector B (el que tiene las medianas de los grupos de 5) y sabemos que B tiene $n/5$ elementos. Todo lo demas como hemos mostrado se hace en $\Theta(n)$. El tiempo que insume el algoritmo lo podemos calcular resolviendo la siguiente recurrencia.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ T(n/5) + T(3n/4) + n & sino \end{array} \right\}$$

Esta es una recurrencia realmente rara que involucra una mezcla de fracciones (n/5) y (3n/4). Por lo que resulta imposible aplicar el teorema maestro, el método iterativo es tambien de resolución compleja por lo que sabiendo que pretendemos que el algoritmo sea lineal vamos a tratar de aplicar el metodo de sutitución para demostrarlo.

Teorema: Existe una constante c tal que $T(n) \leq n$.

Demostración por inducción en n Sea $n = 1$. $T(n) = 1$,por lo tanto $T(n) \leq cn$ si $c \geq 1$

Hipótesis inductiva: $T(n') \leq cn'$ para $n' < n$
Queremos probar que $T(n) \leq cn$
Por definicion sabemos que $T(n) = T(n/5) + T(3n/4) + n$
Como $n/5$ y $3n/4$ son ambos menores que n podemos aplicar la hipótesis inductiva obteniendo.

$$T(n) \leq c\frac{n}{5} + c\frac{3n}{4} + n = cn(\frac{1}{5} + \frac{3}{4}) + n$$
$$T(n) \leq cn\frac{19}{20} + n = n(\frac{19c}{20} + 1)$$

Por lo tanto $c \leq (19c/20) + 1$ Despejando c vemos que se cumple para $c \geq 20$. Como necesitabamos $c \geq 1$ y ahora necesitamos $c \geq 20$. Podemos decir que con $c = 20$. $T(n) \leq cn$ Por lo que queda demostrado.

A propósito: Esta demostración era un ejercicio de uno de los apuntes anteriores, espero que la resolución sea igual o muy parecida a la planteada por Ud.

Algunos podran preguntarse porque el algoritmo de *Floyd*, *etc* ... utiliza grupos de 5 elementos, ocurre que por la demostración anterior el algoritmo funciona para $n > 4$ y por eso eligieron $n = 5$.

4.3.3 Particionando

El otro problema que nos restaba resolver sobre el algoritmo "*Seleccion1*" es como particionar el vector una vez elegido el pivote. Este procedimiento lo vamos a dejar para mas adelante ya que no hay tiempo ni espacio suficiente en esta clase.

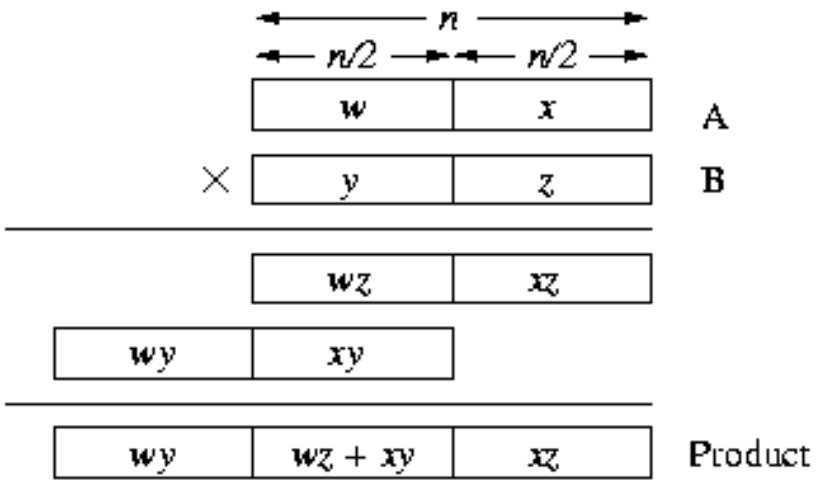
4.4 Problemas no tan simples: Como multiplicar dos números

El problema de multiplicar dos números consiste en encontrar un algoritmo eficiente que permita multiplicar dos números de gran cantidad de dígitos. Este tipo de problema se presenta hoy en muchos algoritmos criptográficos o de generación de numeros aleatorios que deben resolver multiplicaciones entre numeros de muchos dígitos. Sumar dos numeros no es problema ya que resulta sencillo hacerlo en tiempo $\Theta(n)$. Sin embargo el método tradicional para multiplicar dos numeros que todos hemos aprendido en la escuela primaria insume tiempo $\Theta(n^2)$, queremos averiguar si no hay un algoritmo mas eficiente. Sorprendentemente dicho algoritmo existe.

4.4.1 Dividir para conquistar la multiplicación

Trataremos de aplicar el paradigma de Dividir para Conquistar al problema de como multiplicar dos números para ver si podemos obtener un algoritmo que sea mejor que $\Theta(n^2)$.

Aplicando el paradigma de dividir para conquistar hacemos lo siguiente: sean dos numeros A y B de n dígitos cada uno (si tienen distinta cantidad de dígitos completamos con ceros). Dividimos el numero A en sus $n/2$ dígitos mas significativos:w y sus $n/2$ dígitos menos significativos:x. Hacemos lo mismo con B obteniendo y y z. De acuerdo a la partición que hicimos sabemos que $A = w * 10^{n/2} + x$ y $B = y * 10^{n/2} + z$. Una vez realizada la partición podemos realizar la multiplicación de acuerdo al siguiente esquema.



Como vemos podemos hacer el producto de la forma:

$$Sea\, m = n/2$$

$$mult(A, b) = mult(w, y)10^{2m} + (mult(w, z) + mult(x, y))10^m + mult(x, z)$$

La operación de multiplicar por 10^m consiste simplemente en shiftear el número m posiciones y por lo tanto no es realmente una multiplicación, las sumas implican sumar numeros de $n/2$ dígitos y cuesta por lo tanto $\Theta(n)$ cada una. Podemos expresar entonces el producto entre dos números como 4 productos de numeros de la mitad de dígitos de los originales mas una cantidad constante de sumas y shifts de costo $\Theta(n)$. Por lo tanto el costo del algoritmo se puede

calcular a partir de la siguiente recurrencia.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 4T(\frac{n}{2}) + n & sino \end{array} \right\}$$

Aplicando el teorema maestro vemos que $a = 4$ $b = 2$ $k = 1$ y $a > b^k$ por lo que estamos en el caso uno y el orden es $\Theta(n^{\log 4}) = \Theta(n^2)$. Desafortunadamente no hemos podido, aun, mejorar el algoritmo. Pese a no haber podido mejorar el algoritmo podemos apreciar un detalle importante: la parte crítica del algoritmo es la cantidad de multiplicaciones de números de tamanio $n/2$. El numero de sumas, siempre y cuando sea constante, no afecta el orden del algoritmo. Por lo tanto nuestro objetivo va a ser reeemplazar algunas multiplicaciones por una cantidad constante de sumas. Obviamente no podemos multiplicar mediante sumas sucesivas puesto que la cantidad de sumas dependería de n y no sería constante.

4.4.2 El método de Gauss

La clave para mejorar el algoritmo reside en un truco algebraico descubierto por Gauss. Gauss encontro lo siguiente.

$$C = mult(w, y)$$

$$D = mult(x, z)$$

$E = mult((w+x),(y+z)) - C - D = (wy + wz + xy + xz) - wy - xz = (wz + xy)$

Finalmente tenemos

$$mult(A, B) = C * 10^{2m} + E * 10^m + D$$

En total hacemos 3 multiplicaciones, 4 sumas y 2 restas de números de $n/2$ digitos, ademas se requieren algunos corrimientos. El tiempo del algoritmo aplicando el truco de Gauss es el siguiente.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ 3T(n/2) + n & sino \end{array} \right\}$$

Aplicando el teorema maestro nuevamente tenemos $a = 3$, $b = 2$ y $k = 1$. Lo cual nos da $T(n) = \Theta(n^{\log 3}) \simeq \Theta(n^{1.585})$.

Debemos analizar si esto es realmente una mejora. Este algoritmo arrastra una cantidad grande de factores constantes como el overhead debido a la recursión y las operaciones aritméticas adicionales, pero debido a nuestros conocimientos del orden de los algoritmos sabemos que a partir de un cierto n el nuevo algoritmo será mejor ya que el tradicional tiene una tasa de crecimiento mayor. Por ejemplo si suponemos que en el algoritmo de Gauss las constantes son 5 veces mayores que en el algoritmo simple tenemos ($5n^{1.585}$ versus n^2), y el n a partir del cual Gauss es mejor es 50, si las constantes fuesen 10 veces mayores tendriamos $n \geq 260$. En definitiva en general para multiplicar números de **gran** cantidad de dígitos el algoritmo mejorado resulta superior.

4.5 El problema de los puntos mas cercanos. Segunda parte

El problema de los números mas cercanos en el plano, recordemos consistía en dado un conjunto de puntos encontrar la pareja de puntos tal que su distancia es mínima.

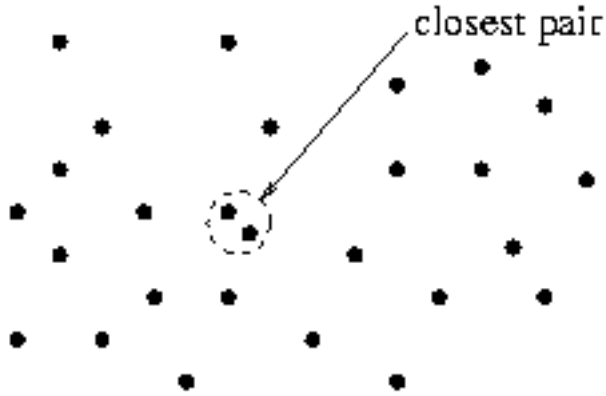


Figure 1: Closest Pair Problem.

Para este problema disponíamos de una solución iterativa de orden $\Theta(n^2)$, tal vez podamos aplicar nuestros conocimientos en algoritmos de tipo Dividir para Conquistar para resolver el problema en forma mas eficiente.Tenemos que aplicar las tres fases del paradigma: dividir, conquistar y combinar.

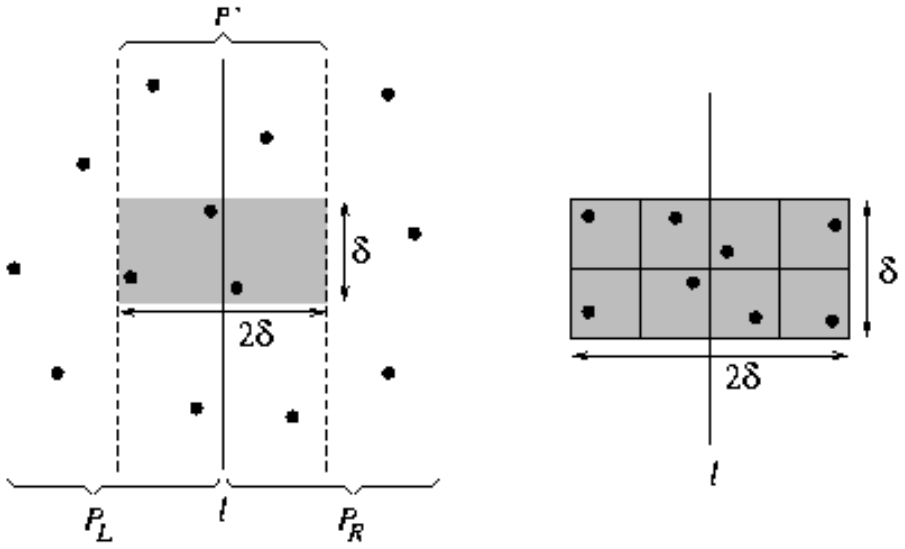
Dividir: Recordemos que los puntos se suponian almacenados en un vector P. Si P contiene pocos puntos, entonces simplemente aplicamos el algoritmo de fuerza bruta iterativo que conocíamos gasta el momento (Tarda una cantidad de tiempo constante). Si hay mas de n puntos entonces trazamos una linea vertical l que subdivide la lista P en dos conjuntos de aproximadamente el mismo tamanio, Pl y Pr.

Conquistar: Recursivamente aplicamos el procedimiento en ambas lisas, por lo que obtenemos δ_l y δ_r distancias mínimas para los dos conjuntos. De ambas distancias obtenemos $\delta = min(\delta_l, \delta_r)$.

Combinar: Lamentablemente δ no no es el resultado final, ya que podría darse que la linea l que hemos elegido pasa justo entre los dos puntos que estan a distancia mínima. Debemos chequear si no hay dos puntos, uno a cada lado

de la línea, cuya distancia sea menor a δ. En primer lugar observemos que no necesitamos chequear todos los puntos, si un punto esta a mayor distancia de l que δ entonces seguro no hay un vecino del otro lado de la línea que pueda formar una distancia menor que δ. Creamos una lista P' de puntos que están a menos de δ de cada lado de l. Determinamos entonces la distancia mínima para los puntos de P' que llamaremos δ' y devolvemos el mínimo entre δ y δ' como resultado. En realidad no devolvemos la distancia sino los dos puntos que estan a dicha distancia uno del otro.

El problema consiste en analizar como encontrar la distancia mínima en P', lo cual requiere un poco de astucia. Para optimizar el algoritmo queremos encontrar la distancia mínima en P' en tiempo $\Theta(n)$ vamos a demostrar que esto puede hacerse suponiend que los puntos de P' estan ordenados por la coordenada y, luego tendremos que resolver como ordenar los puntos.



Supongamos que los puntos de P' estan ordenados de acuerdo a su coordenada y. Consideremos un punto $P'[i]$ en la lista ordenada. Cual es el punto mas cercano a $P'[i]$. Podemos restringir la busqueda a los puntos que tienen índice j mayor que i ya que el proceso lo vamos a hacer para todos los puntos (si el vecino mas cercano esta arriba, entonces esto ya fue analizado cuando buscamos el vecino de dicho punto). Podríamos pensar que necesariamente el vecino mas cercano es $P'[i+1]$, pero esto no es verdad, dos puntos que tienen distancia mínima en cuanto a la coordenada y no necesariamente tienen distancia mínima en el plano, queremos saber que tan lejos tenemos que buscar hasta $P'[i+2]$?,

$P'[i+8]$? tal vez podamos acotar la búsqueda a una cantidad constante de puntos.

Resulta que podemos limitar la cantidad de puntos a buscar y ademas resulta que no nos hace falta analizar mas de 7 puntos en la lista P' para cada punto. Esto lo vamos a tener que demostrar.

Teorema: Si $P'[i]$ y $P'[j]$ $(i < j)$ son la dupla mas cercana en P' y la distancia entre ellos es menor que δ, entonces $j - i \leq 7$.

Demostración: Supongamos que $P'[i]$ y $P'[j]$ son la dupla mas cercana en P' y su distancia es menor que δ. Por estar en P' estan a distancia menor que δ de l. Por ser su distancia menor que δ sus coordenadas y difieren en menos de δ. Por lo tanto ambos puntos residen en un rectangulo de ancho 2δ y altura δ centrado en la línea l. Dividamos este rectángulo en 8 cuadrados del mismo tamanio de lado $\delta/2$. Observemos que la diagonal de cada uno de esto cuadrados tiene como longitud:

$$\frac{\delta\sqrt{2}}{2} = \frac{\delta}{\sqrt{2}} < \delta$$

Como cada cuadrado esta completamente a un lado de la linea l ningun cuadrado puede contener dos puntos de P ya que si los tuviera entonces ambos puntos estarían a distancia menor que δ y δ no sería la distancia mínima entre las distancias mínimas a ambos lados de P. Por lo tanto puede haber a lo sumo 8 puntos de P' en este rectángulo y por lo tanto $j - i \leq 7$.

Por lo tanto por cada $P'[i]$ solo debemos chequear una cantidad constante de números y por eso podemos buscar la distancia mínima en P' en $O(n)$.

El único asunto que nos queda pendiente es como ordenar los puntos de P', dado que queremos hacerlo en $O(n)$ nos vemos impedidos de usar un algoritmo de sort genérico que tardaria $O(n \log n)$ en general.

Hay una solución ingeniosa a este problema: Pre-ordenar los puntos. En particular almacenamos los puntos (redundantemente) en dos listas PX y PY ordenando por las coordenadas x y y respectivamente. Esto lo hacemos llamando a un algoritmo de sort dos veces **antes** de comenzar el algoritmo de dividir y conquistar.

Ya no necesitamos la lista P, cuando hacemos el split de P en Pl y Pr lo que hacemos es dividir PX y PY en Xl, Xr, Yl, Yr respectivamente. Esto es facil de hacer en $O(n)$. Dado que las listas originales estan ordenandas las listas resultantes tambien lo estan. Para armar la lista P' simplemente extraemos los elementos de Y que están a distancia δ de l y luego los copiamos a la lista Y' en donde luego buscamos.

Resumiendo Este es un resumen del algoritmo. Esta version devuelve la distancia mínima, es sencillo aplicar los cambios pertinentes para devolver los puntos a dicha distancia.

- **Presort:** Dada la lista P, hacemos dos copias PX y PY. Ordenamos PX por la coordenada x de los puntos y la lista PY por la coordenada y.
- **Parte Recursiva:** DistMin(X,Y)
  - **Condición de corte:** Si la cantidad de puntos es menor que 4 entonces resolvemos el problema por fuerza bruta y devolvemos la distancia mínima δ analizando todas las distancias posibles.
  - **Dividir:** Sino, sea l la mediana de las coordenadas x de PX. Dividimos ambas listas PX y PY por esta linea, manteniendo el orden, creando Xl, Xr, Yl, Yr.
  - **Conquistar:** δ_l = DistMin(Xl,Yl), δ_r = DistMin(Xr,Yr).
  - **Combinar:** $\delta = min(\delta_l, \delta_r)$.Creamos la lista Y' copiando todos los puntos de Y que estan a distancia menor que δ de l. Para i desde 1 hasta la longitud de Y' y para j desde $i+1$ hasta $i+7$ calcular la distancia entre $Y'[i]$ y $Y'[j]$. Sea δ' la distancia mínima entre dichas distancias. Devolver $min(\delta, \delta')$.

A partir de la descripción es fácil programar el algoritmo.

4.5.1 Análisis

Debemos analizar cuanto tiempo tarda el programa en su peor caso, en el caso promedio en análisis es muy complejo porque depende de cuantos puntos tiene P' en promedio y esto depende en cual es la distancia mínima esperada en cada sublista. En el peor caso podemos suponer que P' tiene a todos los puntos. La fase de pre-sort insume $O(n \log n)$. Para analizar la parte recursiva planteamos la recurrencia.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n \leq 3 \\ 2T(n/2) + n & sino \end{array} \right\}$$

Esta recurrencia ya es conocida y nos permite obtener que el algoritmo es $O(n \log n)$.

De esta forma obtenemos un algoritmo mas eficiente que el algoritmo de fuerza bruta que era $\Theta(n^2)$.

4.6 Resumen y puntos clave

Wow!, estos fueron muchos de temas. El propósito de esta clase fue explicar un paradigma muy utilizado en el disenio de algoritmos: "Dividir para conquistar", mediante varios ejemplos hemos visto como en algunos casos este paradigma

nos permite resolver un problema en forma eficiente. La forma básica de un algoritmo de tipo dividir para conquistar contiene los siguientes puntos.

- **Condici'eon de corte:** Controla en que momento se detiene la recursividad, usualmente consiste en aplicar algun método no muy eficiente cuando el problema alcanza un tamanio chico.
- **Dividir:** Se divide el problema en n sub-problemas mas chicos.
- **Conquistar:** Aplicando recursivamente el algoritmo se resuleven los problemas mas chicos.
- **Combinar:** Se combina la solución de los problemas resueltos para encontrar la solución al problema original.

El algoritmo de *MergeSort* es un algoritmo de sort interno importante, que ilustra claramente el paradigma.

El problema de la *Seleccion* o de la *Mediana* fue durante muchos anios de muy difícil resolución en forma eficiente, el algoritmo que explicamos aqui permite resolver el problema en forma eficiente utilizando un algoritmo de tipo dividir para conquistar. En este problema utilizamos una forma especial de algoritmos "Dividir para Conquistar" en la cual la cantidad de subproblemas es 1. A estos algoritmos se los conoce como *cribas*.

Otra aplicacion conocida del paradigma permite multiplicar dos numeros de gran cantidad de dígitos en forma mas eficiente que el metodo tradicional, aqui utilizamos un truco algebraico sugerido por Gauss para reducir el numero de multiplicaciones que debiamos realizar.

Por ultimo vimos como pudimos aplicar este paradigma al problema de encontrar los puntos a distancia mínima en un plano en forma mas eficiente que nuestra solución anterior.

4.7 Ejercicios

1. Dado un vector A de n numeros encontrar un algoritmo que permita encontrar el elemento mayoritario en A, es decir aquel que aparece con mayor frecuencia en el vector.
2. Una compania petrolera ha construido n torres de extracción de petroleo en la patagonia, la compania desea construir un oleoducto de dirección Norte-Sur que permita trasnportar el petroleo extraido por las torres. Para este problema vamos a suponer que el oleoducto esta formado por una unica linea recta. La empresa debe construir además tramos perpendiculares al oleoducto que lleguen a cada torre. Describir un algoritmo que permita

encontrar la mejor ubicación para el oleoducto de forma tal de minimizar la cantidad de kilometros a construir en tramos perpendiculares. El algoritmo debe ser lineal o mejor.

3. El método de Floyd, Rivest, Pratt y Tarjan toma grupos de 5 elementos para la selección del pivote. Indicar que ocurriría si se tomaran 7 o 3 elementos para dicha selección.

4. Dados dos vectores $X[n]$ y $Y[n]$ ordenados dar un algoritmo de orden $O(\log n)$ que permita calcular la mediana de los $2n$ elementos en conjunto.

Chapter 5

Algoritmos Aleatorizados

5.1 Algoritmos aleatorizados

Los algoritmos aleatorizados son aquellos que en algún momento de su ejecución utilizan la generación de un número aleatorio para tomar una determinada decisión. Sorprendentemente aleatorizar un algoritmo haciéndolo tomar decisiones al azar a menudo sirve para que el algoritmo sea mas eficiente. Los algoritmos aleatorizados se dividen en dos grandes familias: los algoritmos de tipo **Montecarlo** y los algoritmos de tipo **Las Vegas**.

5.1.1 Algoritmos tipo Las Vegas

Los algoritmos de tipo Las Vegas siempre resuelven correctamente un problema, sin embargo pueden (con baja probabilidad de ocurrencia) demorar mucho mas tiempo que en el caso medio para encontrar la respuesta.

5.1.2 Algoritmos tipo Montecarlo

Los algoritmos de tipo Montecarlo siempre son eficientes en cuanto a la cantidad de tiempo necesaria para encontrar la respuesta. Sin embargo en algunos casos (con baja probabilidad de ocurrencia) pueden producir una respuesta no correcta al problema.

5.2 Quick Sort

El algoritmo de Quick Sort es uno de los algoritmos de sort mas conocidos, en primer lugar es un algoritmo de tipo "Divide & Conquer" cuyo tiempo medio según demostraremos es $\Theta(n \log n)$, pero cuyo peor caso es $\Theta(n^2)$. Quick Sort es un algoritmo in-situ, es decir que a diferencia del algoritmo de Mergesort no necesitamos de espacio adicional en memoria para ordenar el vector ya que el proceso se lleva a cabo sobre el vector mismo.

Quick Sort es también un algoritmo aleatorizado ya que utiliza un generador de números al azar para tomar decisiones. Como es de esperar Quick Sort siempre produce un vector correctamente ordenado por lo que lo podemos clasificar como un algoritmo de tipo **Las Vegas**.

5.2.1 Fundamentos de Quick Sort

El funcionamiento básico de Quick Sort lo podemos describir mediante el esquema clásico de los algoritmos "Divide & Conquer".

Condición de corte: Si la cantidad de elementos del vector a ordenar es 0 o 1 entonces return.

Dividir: Seleccionar un elemento x al azar del vector que llamaremos **pivote**. Esta es la parte aleatorizada del algoritmo.

En base al pivote dividimos el vector en tres partes. A[1..q-1], A[q] y A[q+1..n].

Conquistar: Recursivamente invocar Quick Sort para $A[1..q-1]$ y $A[q+1..n]$

Combinar: No es necesario realizar ningún proceso extra ya que A[1..n] queda ordenado.

Algorithm 18 QuickSort(A,p,r). Ordena el vector A desde p hasta r

```
if r ≤ p then
  return
end if
i ⇐ random(p..r)
Swap(A[i],A[p])
q ⇐ Particionar(p, r, A)
QuickSort(p,q-1,A)
QuickSort(q+1,r,A)
```

Análisis En primer lugar es bastante obvio que el algoritmo ordena el vector, es decir que podemos afirmar que es **correcto**. El costo en tiempo depende bastante del pivote elegido, si el pivote fuese un número muy chico o muy grande llamaríamos recursivamente al algoritmo para practicamente todos los números, si el pivote en cambio es un numero medio las particiones se balancean y el rendimiento es mejor. [1]. Ademas para poder proceder al análisis del algoritmo necesitamos un algoritmo que realice la partición, este algoritmo también era solicitado por el algoritmo de *Seleccion*1 que ya estudiamos. Pasemos a ver, pues, como realizar el solicitado proceso de partición.

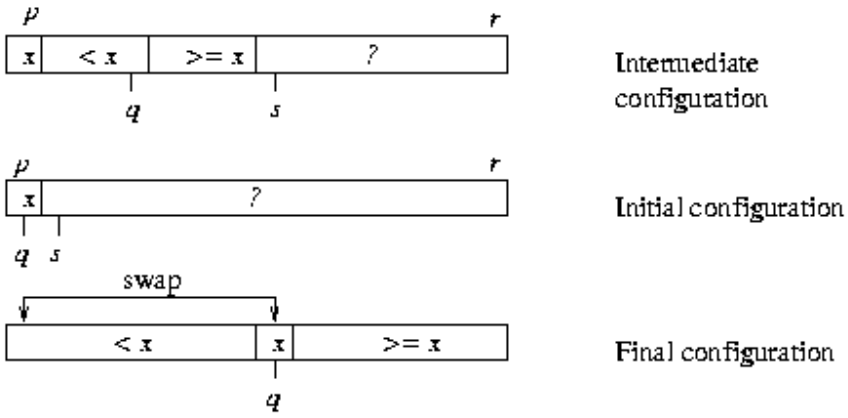
[1]Las llamadas recursivas pueden ser vistas como un árbol, si el pivote esta en el medio el árbol es balanceado, si en cambio el pivote es un extremo una de las ramas del árbol sera mucho mas extensa que la otra y necesitaremos mas llamadas recursivas

5.2.2 Algoritmo para particionar el vector

El algoritmo que presentamos trabaja suponiendo que se recibe un vector A, dos indices p y r que indican desde donde hasta donde particionar y ademas supone que el Pivote es el elemento $A[p]$, o sea el primer elemento del vector. Es por esto que en el algoritmo de Quick Sort se Swapea el pivote con $A[p]$ antes de particionar.

El algoritmo mantiene un invariante a lo largo de su ejecución que indica que el vector esta dividido en 4 segmentos indicados por p, q, s y r. De la forma:

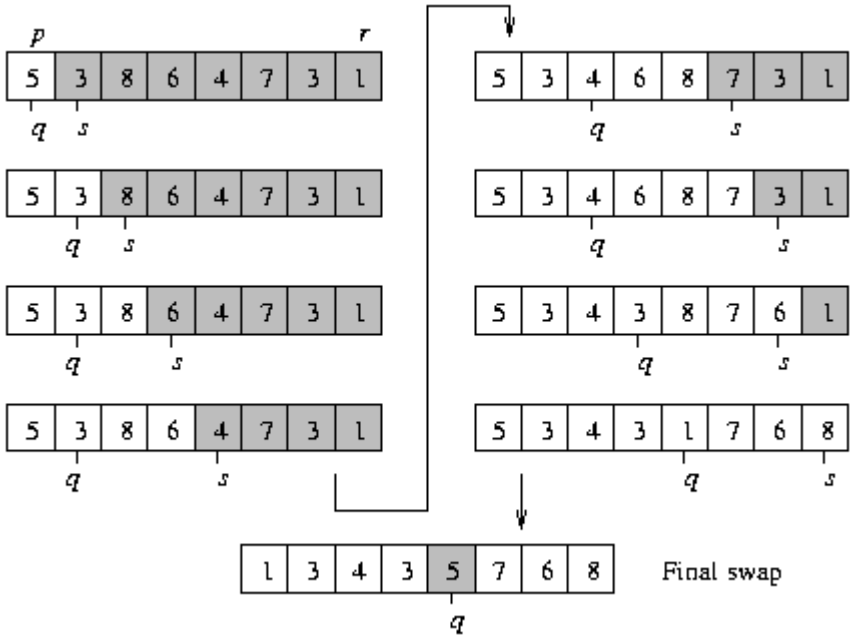
1. $A[p] = x$ es el pivote.
2. $A[p+1..q]$ contiene elementos que son menores que x
3. $A[q+1..s-1]$ contiene elementos que son mayores o iguales que x
4. $A[s..r]$ contiene elementos desconocidos.



El algoritmo que realiza la particion es el siguiente.

Algorithm 19 Particionar(p,r,A). Particiona el vector A[p..r] con pivote A[p]

$x \Leftarrow A[p]$
$q \Leftarrow p$
for $s = p + 1$ to r **do**
  if $A[s] < x$ **then**
    $q \Leftarrow q + 1$
    $Swap(A[q], A[s])$
  end if
end for
$Swap(A[p], A[q])$
$returnq$



En la figura vemos un ejemplo de funcionamiento del algoritmo de particion.

Análisis En primer lugar es evidente que el algoritmo funciona en forma correcta, además esta es solo una implementación, hay varias formas de realizar la partición y todas ellas son mas o menos iguales en eficiencia. El algoritmo tiene un único ciclo que recorre todos los elementos por lo que se trata claramente de un algoritmo de orden lineal.

5.2.3 Análisis de *QuickSort*

En general el tiempo necesario por el procedimiento *QuickSort* esta dado por la recurrencia.

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ T(q-1) + T(n-q) + n & sino \end{array} \right\}$$

Y como vemos el análisis depende del valor de q

Análisis del peor caso

El peor caso ocurre cuando el pivote es uno de los extremos del vector (extremo en cuanto a su valor: mínimo o máximo), suponiendo por ejemplo que el pivote es el elemento mas chico del vector tendremos que q, la posición del pivote es 1.Aplicando el método iterativo a la recurrencia.

$$\begin{aligned} T(n) &= T(0) + T(n-q) + n \\ T(n) &= 1 + T(n-1) + n \\ T(n) &= T(n-1) + (n+1) \\ T(n) &= T(n-2) + n + (n+1) \\ T(n) &= T(n-3) + (n-1) + n + (n+1) \\ T(n) &= T(n-4) + (n-2) + (n-1) + n + (n+1) \\ &\vdots \\ T(n) &= T(n-k) + \sum_{i=-1}^{n-3} n - i \\ T(n) &= O(n^2) \end{aligned}$$

Análisis del caso medio

Para analizar el caso medio debemos interpretar a $T(n)$ como el tiempo promedio del algoritmo *QuickSort* para un vector de tamanio n. El algoritmo tiene n opciones equiprobables para elegir el pivot. Por lo tanto la probabilidad de elegir un elemento es $1/n$. La recurrencia que tenemos que plantear es:

$$T(n) = \left\{ \begin{array}{ll} 1 & si\, n = 1 \\ \frac{1}{n} \sum_{q=1}^{n} \left(T(q-1) + T(n-q) + n\right) & sino \end{array} \right\}$$

Esta no es una recurrencia comun, por lo que no podemos aplicar el Teorema Maestro, el metodo iterativo es posible pero resulta complicado. Entonces vamos a intentar demostrar que el caso medio es $\Theta(n \log n)$.

Teorema: Existe una constante c tal que $T(n) \leq cn \ln n$ para todo $n \geq 2$. Notemos que hemos reemplazado *log* con *ln*, esto simplifica nuestra demostracion sin afectar el orden, de hecho cualquier logaritmo sirve.

Demostración: La demostración es por inducción en n. Para el caso básico $n = 2$ tenemos:

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

$$T(n) = \frac{1}{2}((T(0) + T(1) + 2) + (T(1) + T(0) + 2) = \frac{8}{2} = 4$$

$$4 \leq 2 \cdot c \ln 2 \Rightarrow c \geq 4/(2 \ln 2) \Rightarrow c \geq 2.88$$

Hipotesis inductiva Asumimos que $n \geq 3$, entonces para cualquier $n' \leq n$ vale que $T(n') \leq cn' \ln n'$. Queremos probar que esto es válido para T(n). Expandiendo la definición de T(n) y sacando el factor n afuera de la sumatoria tenemos.

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

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

Observemos que si dividimos la sumatoria en dos sumatorias ambas suman los mimso valores $T(0) + T(1) + \ldots + T(n-1)$, una desde 0 hasta $n-1$ y la otra desde $n-1$ hasta 0. Entonces podemos escribir la sumatoria como $2\sum_{q=0}^{n-1} T(q).T(0)$ y $T(1)$ no responden a esta formula por lo que los vamos a tratar en forma especial. Aplicando esta sustitucion y aplicando la hipótesis inductiva a la sumatoria restante lo cual podemos hacer porque $q \leq n$ tenemos.

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

$$T(n) = \frac{2}{n}(1 + 1 + \sum_{q=2}^{n-1}(cq \ln q)) + n$$

$$T8n) = \frac{2c}{n}(\sum_{q=2}^{n-1}(cq \ln q)) + n + \frac{4}{n}$$

Esta sumatoria no la hemos visto nunca hasta el momento, vamos a mostrar la solucion.

$$S(n) = \sum_{q=2}^{n-1} q \ln q \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}$$

Reemplazando

$$T(n) = \frac{2c}{n}(\frac{n^2}{2} \ln n - \frac{n^2}{4}) + n\frac{4}{n}$$

$$T(n) = cn \ln n - \frac{cn}{2} + n + \frac{4}{n}$$
$$T(n) = cn \ln n + n(1 - \frac{c}{2}) + \frac{4}{n}$$

Queremos que esta expresion sea menor o igual que $cn \ln n$ Cancelando $cn \ln n$ a ambos lados de la desigualdad vemos que se cumple si.

$$n(1 - \frac{c}{2}) + \frac{c}{n} \leq 0$$

Esto se cumple si $c = 3$, como ademas necesitabamos $c \geq 2.88$ queda demostrado.

Resumen Hasta el momento el peor caso del algoritmo $QuickSort$ es $O(n^2)$, el caso medio es $\Theta(n \log n)$. Aunque no lo hemos demostrado el peor caso no ocurre muy a menudo. Para valores grandes de n el tiempo puede considerarse $\Theta(n \log n)$ con una probabilidad alta. Para degenerar en el orden cuadrático el algoritmo tendria que elegir el pivote en forma defetuosa en cada uno de los pasos recursivos. Las elecciones malas del pivote son poco probables por lo que una secuencia de malas elecciones es aún menos probable. El algoritmo de $QuickSort$ es ademas un algoritmo de sort $in - situ$.

5.2.4 La ventaja del algoritmo aleatorizado

Existen versiones de $QuickSort$ que no eligen el pivote en forma aleatoria sino que por ejemplo toman siempre el primer elemento del vector como pivote. Esta técnica tiene como desventaja que, por ejemplo, si el vector ya se encontraba ordenado o bien se encontraba rebatido el algoritmo se encuentra automaticamente en el peor caso y adquiere orden cuadrático. Por eso es muy aplicable la elección aleatoria ya que ocasiona una ventaja fundamental **Ninguna disposición original de los datos ocasiona el peor caso**, esto es fácil de ver ya que la ocurrencia del peor caso no depende de los datos de entrada sino, de la elección del pivote que es aleatoria.

Hemos dado con el principio clave de los algoritmos de tipo Las Vegas: el tomar una decisión aleatoria para indpendizar el funcionamiento del algoritmo de los datos de entrada ocasionado que el peor caso ocurra con muy baja probabilidad (n malas elecciones de un numero aleatorio).

5.2.5 Resumen $QuickSort$

- Es un algoritmo de tipo Divide & Conquer.
- Es un algoritmo aleatorizado de tipo Las Vegas.
- El peor caso es $\Theta(n^2)$ con muy baja probabilidad.

- El caso medio que es altamente probable es $\Theta(n \log n)$.
- Ordena el vector $in - situ$
- No es estable.

5.3 El problema de los puntos mas cercanos en el plano. Parte 3.

El problema de los puntos mas cercanos en el plano, recordemos, consistía en encontrar los dos puntos que se encuentran mas cercanos (a distancia mínima) dentro de un conjunto de n puntos que se almacenan en un vector $A[1..n]$. En clases anteriores vimos una solución iterativa que utilizaba fuerza bruta para seleccionar cual era la dupla mas cercana analizando todas las posibles parejas de puntos, este algoritmo era de orden cuadrático. Luego vimos un algoritmo de tipo Divide & Conquer que con una aproximación mas inteligente permitia resolver el problema en $n \log n$ A continuación veremos como podemos plantear un algoritmo de tipo Las Vegas aún mas inteligente que el anterior para resolver el problema en forma mas eficiente.

5.3.1 Algoritmos aleatorizados incrementales

Una sub-división de los algoritmos aleatorizados son los algoritmos incrementales, en este caso vamos a presentar un algoritmo tipo Las Vegas incremental para resolver el problema de la distancia mínima. La idea es la siguiente, los puntos van a ser insertados uno a uno en un conjunto inicialmente vacío y por cada punto que insertamos vamos a chequear si no es necesario actualizar cual es la distancia mínima.

5.3.2 El algoritmo

El funcionamiento del algoritmo dependerá del orden en que se inserten los puntos, algunos ordenamientos serán particularmente malos ya que necesitaremos actualizar la distancia mínima por cada puunto insertado lo cual nos insumirá tiempo $O(n^2)$. Sin embargo al insertar los puntos en orden aleatorio el tiempo esperado del algoritmo sera $O(n)$. No lo vamos a demostrar pero la probabilidad de que el algoritmo insuma mas de $O(n \log n)$ es extremadamente chica.

Para simplificar asumamos que los puntos han sido normalizados para pertenecer al cuadrado [0,1]. (Esta suposición es solo para facilitar la explicación del algoritmo). En primer lugar permutamos en forma aleatoria los puntos, y sea $(p1, p2 \ldots pn)$ la permutación resultante. Inicialmente ponemos el primer punto en el conjunto y la distancia mínima al haber solo un punto en el conjunto es ∞. Uno por uno insertamos los puntos $p2, p3, etc.$ Sea P_{i-1} el conjunto

$\{p1, \ldots, p_{i-1}\}$ y sea δ la distancia mínima en P_{i-1}. Sea δ' la distancia mínima entre p_i y cualquier punto de P_{i-1} si $\delta' < \delta$, entonces actualizamos la distancia.

La gran pregunta es como encontrar el punto mas cercano a p_i, sería demasiado lento considerar todos los puntos en P_{i-1}. En purmer lugar observemos que solo necesitamos considerar puntos de P_{i-1} que estan a distancia δ de P_i, ya que ningún otro punto puede afectar la dupla mas cercana actual. Declaramos que esto se puede hacer en tiempo constante si los puntos son almacenados en una estructura apropiada.

Grilla de Buckets La estructura de datos que vamos a utilizar para guardar el conjunto de puntos se denomina **grilla de buckets** En particular subdividimos el cuadrado unitario [0,1] en una grilla de cuadraditos de lado δ. Dentro de cada cuadradito mantenemos una lista enlazada de los puntos de P_{i-1} que pertenecen al cuadradito, esta lista se denomina *bucket*.

Para determinar a que bucket corresponde un punto observemos que si dividimos el intervalo [0,1] en subintervalos de longitud δ tendremos $d = \lceil 1/\delta \lceil$ subintervalos. Si asumimos que indexamos los subintervalos de la forma $0, 1, \ldots, d-1$ para determinar el subintervalo que contiene un cierto valor x dividimos x por δ y redondeamos hacia abajo al entero mas cercano es decir.

$$I_\delta = \lfloor x/\delta \rfloor$$

Por ejemplo si $\delta = 0.3$ entonces los subintervalos son $0 : [0, 0.3]$, $1 : [0.3, 0.6]$, $2 : [0.6, 0.9]$, $3 : [0.9, 1]$. El valor $x = 0.72$ es mapeado al intervalo $\lfloor 0.72/0.3 \rfloor = 2$. Extendiendo esto a dos dimensiones mapeamos el punto (x, y) al bucket $[I_\delta(x), I_\delta(y)]$. Cuando el algoritmo comienza inicialmente $\delta = \infty$ y existe un único bucket.

Para almacenar los buckets necesitamos una **hash table** ya que no podemos usar una matriz porque para δ muy chicos tendríamos mas entradas en la matriz de las que podríamos manejar. Este es un buen ejemplo de como manejar el costo espacial de algoritmo. Aquellos que no conozcan como funciona una hash-table pueden despreocuparse y suponer que los buckets estan almacenados en una matriz.

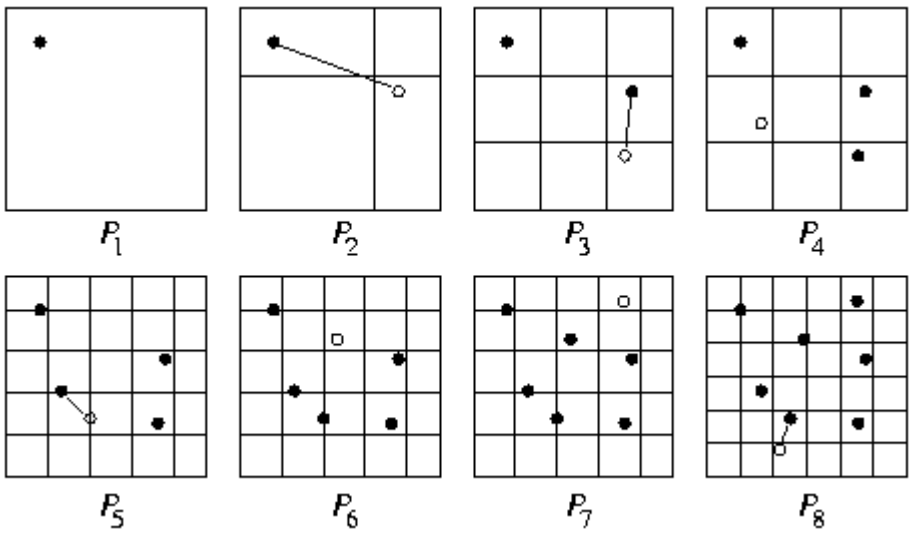
Un aspecto notable es que no podemos tener mas de 4 (cuatro) puntos en cada bucket (uno en cada esquina del bucket) ya que de lo contrario la distancia mínima no sería δ sino que sería menor.

Dado el punto a insertar p_i, para determinar el vecino mas cercano o bien esta en el bucket correspondiente a p_i o bien es uno de los 8 bucktes vecinos, por lo tanto a lo sumo revisamos 9 bucktes y cada uno de ellos tiene a lo sumo

4 puntos, por lo que la cantidad máxima de puntos a revisar es $4 \cdot 9 = 36$ y es constante.

Si no hay un punto a distancia menor que δ entonces la distancia mínima permanece inalterada e insertamos el nuevo punto en su bucket correspondiente. Si la distancia mínima es modificada debemos reconstruir la grilla para la nueva distancia mínima, esto insume $O(i)$ tiempo, ya que cada punto a ubicar insume una cantidad de tiempo contante.El algoritmo es el siguiente.

1. Permutar aleatoriamente los puntos. Sea $\{p1, p2, \ldots, pn\}$ la permutación resultante. Crear una grilla trivial de $1x1$ e insertar $p1$ en la grilla. Sea $\delta = \inf$
2. Para i desde 1 hasta n hacer
   (a) Sea $p_i = (X_i, Y_i)$. Seleccionar el bucket $[I_\delta(X_i), I_\delta(Y_i)]$ y sus 8 buckets vecinos de la grilla.
   (b) Si no hay puntos en ninguno de estos buckets entonces continuar el ciclo. Sino computar la distancia mínima y los (a lo sumo 36) puntos encontrados en los buckets. Sea δ' esta nueva distancia.
   (c) Si $\delta' < \delta$ entonces $\delta = \delta'$, destruir la grilla anterior y construir una nueva grilla a partir de $P_i = \{p1, \ldots, pi\}$. Sino simplemente agregar p_i a la grilla.
3. Dovolver δ



5.3.3 Analisis

Este es un algoritmo iterativo, por lo que para analizarlo no vamos a necesitar plantear, al fin, ninguna recurrencia.

Peor caso

En el pero caso cada punto que agregamos al conjunto nos obliga a actualizar la distancia mínima y por lo tanto a reconstruir la grilla, el tiempo se calcula entonces como.

$$T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$$

Mejor caso

En el mejor caso solo el segundo punto nos hace reconstruir la grilla y luego nunca mas actualizamos la distancia mínima, el tiempo es entonces.

$$T(n) = \sum i = 1^n O(1) = O(n)$$

Caso medio

Si realizamos una simulación del algoritmo veremos que la grilla es reconstruida una cierta cantidad de veces para los primeros puntos pero, a medida que el algoritmo procesa mas y mas puntos, la distancia mínima es cada vez mas firme y la reconstrucción de la grilla se vuelve muy poco frecuente. Para analizar el caso medio asumimos que todas las permutaciones de los puntos son equiprobables. Sea π_i la probabilidad de que el $i$ésimo punto ocasione un cambio en la distancia mínima. Entonces con probabilidad π_i la grilla es reconstruida en la $i$ésima iteración con un costo de $O(i)$. Con probabilidad $(1 - \pi_i)$ el $i$ésimo punto no afecta la distancia y la iteración insume $O(1)$. Para obtener el caso medio necesitamos el promedio ponderado del tiempo que insume cada ciclo de cuaerdo a su probabilidad es decir:

$$T(n) = \sum_{i=1}^{n} (i \cdot \pi_i + 1 \cdot (1 - \pi)) \leq \sum_{i=1}^{n} (i\pi_i) + n$$

Necesitamos saber cuanto vale π_i, para ello recurrimos a un truco bastante usado en el análisis probabilístico que consiste en analizar el algoritmo funcionando de atras para adelante. Obseremos que P_{i-1} es un conjunto de $i-1$ puntos, analizar que ocurre cuando un punto aleatorio es insertado en P_{i-1} es simétrico a analizar que ocurre cuando eliminamos un punto al azar de P_i. De aqui obtenemos esta observación.

Observación La probabilidad de que al insertar un nuevo punto en P_{i-1} se actualice la distancia mínima es igual a la probabilidad de que al eliminar un punto al azar de P_i estemos eliminando uno de los puntos que estan a distancia mínima.

Como hay exactamente dos puntos que participan de la distancia mínima y en total hay i puntos la probabilidad de que uno de los i puntos este en la dupla a distancia mínima es $2/i$ para $i \geq 2$. Luego $\pi_i = 2/i$ y calculamos.

$$T(n) = \sum i = 1^n i(2/i) + n = \sum_{i=1}^{n} 2 + n = 3n = O(n)$$

Por lo tanto el tiempo medio es O(n) y podemos afirmar que con una probabilidad muy alta este algoritmo es de orden lineal.

Conclusiones Aqui termina nuestro seguimiento del problema de la distancia mínima en el plano, hemos visto como a medida que estudiábamos distintas técnicas para el disenio de algoritmos podíamos resolver el problema en forma mas eficiente. Planteamos primero un algoritmo iterativo que usando fuerza bruta resolvía el problema en $O(n^2)$, luego mediante un algoritmo de tipo "Divide & Conquer" obtuvimos $O(n \log n)$, por último mediante un algoritmo aleatorizado resolvimos el problema en $O(n)$.

5.4 Testeando primalidad

Esta va a ser nuestra primera y única incursión dentro de los algoritmos de tipo numérico, estos algoritmos han adquirido gran importancia en los ultimos anios debido a que numerosos protocolos criptográficos y funciones de encripción de datos utilizan este tipo de algoritmos, en esta sección vamos a estudiar un algoritmo numérico particular que sirve para determinar si un número es primo o no.

Definición Un número es primo si y solo si solo es divisible por 1 y por sí mismo.

5.4.1 Algoritmo por fuerza bruta

Basandonos en la definición podemos construir un algoritmo bastante simple.

El algoritmo insume en el peor caso $n-1$ divisiones, es decir que es $O(n)$ pero si analizamos el orden del algoritmo en función del tamanio del número en bits podemos observar que la cantidad de bits de un número n es $k = \log(n)$ por lo

Algorithm 20 EsPrimo1(n). Devuelve true si n es primo

```
for k = 2 to n do
  if kmodn then
    return false
  end if
end for
return true
```

tanto $n = 2^k$, con lo cual el algoritmo es $O(n) = O(2^k)$ es decir que se trata de un algoritmo exponencial y claramente debe ser mejorado.

Teorema: Si n no es primo entonces n tiene un factor primo p tal que $p \leq \sqrt{n}$.

Demostracion: Supongamos que un número n no es primo y su factorización prima es $n = p1p2 \ldots pk > (\sqrt{n})^k$ lo cual es falso salvo que $k < 2$ es decir que n sea primo. Por lo tanto queda demostrado.

5.4.2 Algoritmo por fuerza bruta mejorado

El orden del algoritmo es ahora $O(\sqrt{n}) = O(\sqrt{2^k}) = O(2^{k/2}$ con lo cual pese a reducir notablemente la cantidad de divisiones a realizar seguimos teniendo un algoritmo de orden exponencial.

5.4.3 El pequenio teorema de Fermat

Para todos los primos, p **y enteros** a **tales que** $1 \leq a \leq p-1$ **vale que** $a^{p-1} = 1(modp)$

Por ejemplo $a = 5$, $p = 7$ $5^6 = 15625 = 2232 \cdot 7 + 1 = 1(mod7)$

La demostración del pequenio teorema de fermat nada tiene que ver con la materia, sin embargo la tenemos disponible si alguien quiere observarla por curiosidad.

Usando el teorema de Fermat, podemos especificar un algoritmo aleatorizado para testear si un número es primo.

5.4.4 Algoritmo aleatorizado tipo Montecarlo

Como vemos este algoritmo sirve para decir si un número no es primo pero no asegura que un número sea primo, esto ocurre porque el teorema de Fermat no funciona en ambas direcciones, que dos números n y a satisfagan el teorema no quiere decir que n sea primo.

Algorithm 21 EsPrimo2(n). Devuelve true si n es primo

```
n ⇐ random(1..n − 1)
if a^{n−1} ≠ 1(modn) then
  return false
else
  return true (aunque debería ser maybe!)
end if
```

Por ejemplo $n = 341$, $a = 22$, $2^{340} = (2^{10})^{34} = 1024^{34} = (3 \cdot 341 + 1)^{34}$ Pero $341 = 11 \cdot 31$

Claramente estamos ante un algoritmo aleatorizado (usa un número al azar para realizar el chequeo) y de tipo Montecarlo, ya que la aleatorización no afecta el tiempo de ejecución del algoritmo sino el resultado. Eventualmente nuestro algoritmo $EsPrimo2$ puede devolver como primo un número que no lo es. El orden del algoritmo es el orden que insume realizar a^{n-1}, como vimos en clases anteriores es $\Theta(\log n)$ es decir que pasamos de un algoritmo exponencial a uno sub-lineal (logarítmico).

Un detalle importante es calcular cual es la probabilidad de que un número que no es primo sea considerado primo por el algoritmo, no la vamos a calcular en este apunte pero es inferior a 0.5. Si se elige mas de un número al azar y se efectúa el chequeo varias veces esta probabilidad disminuye, de hecho la probabilidad de devolver un resultado erroneo se puede hacer tan chica como se desee.

Algorithm 22 EsPrimo2(n). Devuelve true si n es primo

```
for k = 1 to constante do
  n ⇐ random(1..n − 1)
  if a^{n−1} ≠ 1(modn) then
    return false
  end if
end for
return true
```

El orden del algoritmo es $\Theta(j \cdot \log n)$ es decir que sigue siendo un algoritmo de orden logarítmico. Cuanto mayor sea j mas tiempo va a tardar pero eso no afecta el orden del algoritmo.

Este comportamiento es típico de los algoritmos de tipo Montecarlo, en donde se puede disminuir la probabilidad de devolver un resultado equivocado a costas de insumir mayor tiempo en el algoritmo.

Numeros muy raros Pese a que hagamos el chequeo una cantidad enorme de veces el algoritmo aun tiene un problema, existen números no primos que satisfacen el teorema de Fermat para todo $1 \leq a \leq n-1$, estos números se conocen como "Números Carmiqueleanos", nuestro algoritmo es incapaz de distinguir un número primo de un número Carmiqueleano. Afortunadamente estos números son muy raros, hay solamente 255 en los primeros 100 millones de números enteros, por lo tanto podemos testear si el número es o no Carmiqueleano contra una tabla (Esto se hace en tiempo constante porque la tabla tiene longitud fija).

Los primeros números Carmiqueleanos

$$561, 1105, 1729$$

Nuestra tercera versión del algoritmo EsPrimo es muy similar al algoritmo utilizado hoy en día para chequear si un número es o no primo, los algoritmos tipo Montecarlo son ideales ya que reducen enormemente el tiempo de procesamiento y son bastante confiables en cuanto al resultado que producen. El algoritmo mas utilizado es el de "Miller-Rabin" que también detecta y elimina números Carmiqueleanos.

5.5 Resumen y puntos clave

Esta clase presento una familia de algoritmos muy particulares: los algoritmos aleatorizados, que en algún momento de su ejecución toman una decisión en base a un número random.

Los algoritmos de tipo **Las Vegas** aleatorizan el tiempo de ejecución del mismo, siempre producen un resultado correcto pero el mismo puede insumir gran cantidad de tiempo con una probabilidad muy baja. En estos algoritmos el objetivo de la aleatorización es independizar el fucionamiento del algoritmo de la disposición de los datos de entrada, de forma tal que el peor caso se vuelva muy poco probable. El algoritmo *QuickSort* es un ejemplo típico.

Los algoritmos de tipo **Montecarlo** aleatorizan el resultado que producen, son algoritmos muy rápidos pero que pueden dar un resultado equivocado con una probabilidad muy baja. La probabilidad de obtener un resultado erroneo puede disminuirse aumentando el tiempo del algoritmo. En estos algoritmos la aleatorización tiene por objetivo resolver en forma muy veloz un problema que si debe resolverse con total confianza insumiría demasiado tiempo. Los algoritmos de testeo de primalidad son los ejemplos mas notables de la aplicación de este tipo de algoritmos.

5.6 Ejercicios

1. El siguiente algoritmo creado por N.Lomuto particiona $A[p..r]$ en $A[p..i]$, $A[i+1..j]$ de forma tal que cada elemento de la primera región es menor o igual a $x = A[r]$ y cada elemento de la segunda región es mayor o igual que x.

Algorithm 23 Lomuto-Partition(A,p,r).

```
x ← A[r]
i ← p − 1
for j = p to r do
  if A[j] ≤ x then
    i ← i + 1
    Swap(A[i], A[j])
  end if
end for
if i < r then
  return i
else
  return i-1
end if
```

   (a) Validar el algoritmo

   (b) Analizar el tiempo de ejecución del algoritmo.

   (c) Analizar el tiempo de ejecución de $Quick - Sort$ utilizando el algoritmo de Lomuto para particionar.

   (d) Definir un algoritmo aleatorizado que intercambie $A[r]$ con un elemento elegido al azar de $A[p..r]$ y luego invoque al algoritmo de Lomuto. Analizar el algoritmo.

2. Mostrar que el mejor caso de $Quick - Sort$ es $\Omega(n \log n)$

3. Una forma interesante de mejorar el rendimiento de $Quick - Sort$ es aprovechando la eficiencia de $Insert - Sort$ para ordenar vectores que están **casi ordenados**. De esta forma se puede modificar el algoritmo de Quick-Sort para que no ordene particiones de menos de k elementos. Luego se llama a Insert-Sort sobre el vector casi ordenado para finalizar el proceso. Mostrar que este algoritmo es $O(nk + n \log(n/k))$. Indicar como debería elegirse k tanto en teoría como en la práctica.

Chapter 6

Sort

6.1 Algoritmos de Sort

El propósito de esta clase es revisar algunos algoritmos de sort ya vistos con anterioridad, estudiar algunos metodos de sort y realizar algunas conclusiones generales sobre métodos de sort. Para comenzar vamos a establecer algunas definiciones que serán útiles mas adelante.

Estabilidad Se dice que un algoritmo de sort es **estable** si para elementos que tienen la misma clave en el vector original mantiene el orden en el vector resultado. La estabilidad es una propiedad deseable en la mayoría de los algoritmos de sort y se vuelve necesaria cuando estamos ordenando por mas de una clave.

In-situ Un algoritmo de sort funciona "in-situ" cuando no requiere de espacio de memoria adicional (a excepción de un cierto espacio constante) para realizar el ordenamiento.

6.1.1 Algoritmos no recursivos

Los algoritmos de sort no-recursivos son los mas simples para programar y entender, hay muchos métodos pero tres de ellos son los mas conocidos: el burbujeo, la selección y la inserción.

Burbujeo

Es un algoritmo "in-situ", puede implementarse como un algoritmo estable y es de orden cuadrático $\Theta(n^2)$. El código y funcionamiento del algoritmo ya fue visto en clases anteriores, es un algoritmo que en general resulta demasiado lento ya que realiza una gran cantidad de intercambios entre los elementos del vector.

Selección

Este algoritmo es de tipo "in-situ". Lamentablemente no puede implementarse en forma estable sin grandes modificaciones. Es de orden cuadrático al igual que el burbujeo $\Theta(n^2)$. En general es mas eficiente que el burbujeo porque realiza menor cantidad de intercambios entre los elementos del vector.

Inserción

El isert-sort es un algoritmo "in-situ" que puede implementarse en forma estable, es tambien de orden cuadrático $O(n^2)$ pero en su mejor caso es de orden lineal $O(n)$. Pese a ser considerado un algoritmo lento es en general mejor que el burbujeo y la selección.

6.1.2 Algoritmos recursivos

Los algoritmos recursivos se basan en general en el paradigma "Divide & Conquer". Los algoritmos de este tipo mas populares son el *QuickSort*, el *MergeSort* y el *HeapSort*, los dos primeros ya fueron estudiados mientras que el tercero se ve en gran detalle en este apunte.

MergeSort

El algoritmo de MergeSort es un algoritmo estable de orden $\Theta(n \log n)$, lamentablemente no se puede implementar in-situ pues requiere de un vector auxiliar del mismo tamanio que el original para funcionar.

QuickSort

El algoritmos de QuickSort es un algoritmo in-situ pero no es estable, es un algoritmo aleatorizado de tipo Las vegas, en su peor caso es de orden cuadrático $O(n^2)$, sin embargo este caso se da con muy baja probabilidad. En la mayoría de los casos es $O(n \log n)$. Este algoritmo creado por "Hoare" es reconocido por ser en general el mas rápido de los algoritmos de sort, el truco del algoritmo consiste en que las comparaciones se hace siempre contra un elemento pivote que puede ser almacenado en un registro de la máquina, esto le da ventaja sobre algoritmos que son asintoticamente equivalentes como el HeapSort o el MergeSort.

6.2 HeapSort

El algoritmo de HeapSort se basa en la utilización de una estructura de datos especial denominada **"Heap"**.

6.2.1 Heaps

Un **Heap** es una estructura especial que permite implementar una cola con prioridades, las operaciones principales de las colas con prioridad se pueden realizar

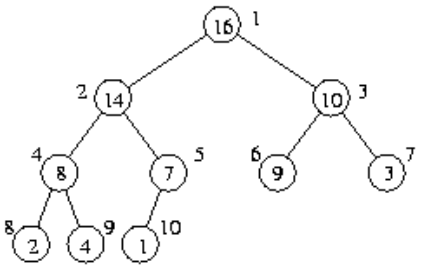
usando un Heap en $\Theta(\log n)$. Un Heap es un árbol binario completo a izquierda, lo cual quiere decir que todos los niveles del árbol excepto el último están completos y que el último nivel se completa de izquierda a derecha. Además los elementos se almacenan cumpliendo la propiedad de que la clave de un nodo siempre es mayor (o menor) que la clave de sus hijos. De esta forma la raiz del arbol es siempre el mayor elemento en todo el Heap.

Importante Un Heap debe cumplir dos propiedades:

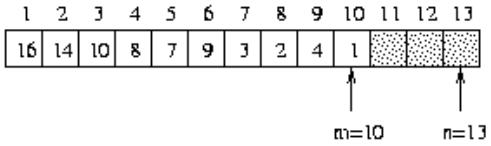
1. Ser un árbol binario completo a izquierda.
2. La clave de un nodo siempre es mayor que la clave de su(s) hijo(s).

Almacenamiento Un Heap tiene la interesante cualidad de poder ser almacenado en un vector sin ocasionar inconvenientes en el manejo de la estructura. Lo cual evita la necesidad de utilizar punteros. Esto es posible gracias a que un Heap siempre es un árbol completo a izquierda. El Heap se almacena en un vector $A[1..n]$, ademas se usa una variable que indica cuantos elementos tiene el Heap: m, de esta forma el heap es el sub-array $A[1..m]$.

Almacenamos el Heap en el vector copiándolo en el mismo nivel por nivel. El primer nivel tiene exactamente 1 elemento, el segundo 2, y asi sucesivamente hasta el último nivel que es el que puede no estar completo, la cantidad de elementos en el último nivel la conocemos por conocer la cantidad m de elementos en el Heap.



Heap as a binary tree



Heap as an array

Para acceder a los elementos del Heap en el array basta con efectuar algunas operaciones aritméticas básicas.

- Left(i) : return $2 \cdot i$
- Right(i) : return $2 \cdot i + 1$

- Parent(i) : return $\lfloor i/2 \rfloor$
- IsLeaf(i) : return $Left(i) > m$
- IsRoot(i) : return $i == 1$

Manteniendo los elementos en orden Para mantener la propiedad de que la clave de un nodo siempre es mayor que la clave de sus hijos existe una operación fundamental denominada $Heapify$. La idea es que dado un elemento del Heap que no está en orden y suponiendo que el resto del Heap si lo está se procede a colocar el elemento en cuestión en su posición correspondiente realizando interacambios con sus hijos si fuese necesario y luego invocando el procedimiento en forma recursiva. Es decir que el elemento fuera de orden es empujado hasta el lugar que le corresponde.

Algorithm 24 Heapify(A,i,m). Reordena el heap de m elementos a partir de A[i]

$l \Leftarrow Left(i)$
$r \Leftarrow Right(i)$
$max = i$
if $l \leq m$ and $A[l] > A[max]$ **then**
  $max \Leftarrow l$
end if
if $r \leq m$ and $A[r] > A[max]$ **then**
  $max \Leftarrow r$
end if
if $max \neq i$ **then**
  Swap(A[i],A[max])
  Heapify(A,max,m)
end if

Recomendamos realizar un seguimiento para ver como funciona este procedimiento. Ademas el procedimiento recursivo es intuitivo pero no es el mas eficiente de todos, la versión iterativa puede ser un poco mejor y se deja como ejercicio.

HeapSorting El algoritmo de HeapSort consiste de dos grandes partes: En primer lugar construir el Heap inicial, y luego extraer el mayor (o menor) elemento del Heap uno a uno.

Análisis de Heapify Un árbol binario tiene completo tiene $\log n$ niveles. La cantidad de trabajo que se hace en cada nivel es constante, por lo tanto Heapify es $O(\log n)$. No es $\Theta(\log n)$ ya que para las hojas por ejemplo el tiempo es constante: $O(1)$.

Construyendo un Heap El procedimiento Heapify se puede utilizar para construir un Heap de la siguiente manera, primero se empieza con un Heap completamente desordenado, luego se invoca Heapify para cada nodo comenzando desde el primer nivel de nodos no-hojas. El primer nodo no-hoja es $A[\lfloor n/2 \rfloor]$.

Algorithm 25 BuildHeap(A,n). Construye un Heap a partir de A[1..n]

```
for i = n/2 downto 1 do
  Heapify(A,i,n)
end for
```

Análisis de BuildHeap Como cada llamada insume $O(\log n)$ y se hacen $n/2$ llamadas el tiempo es $O((n/2)\log n) = O(n \log n)$ Luego veremos que en realidad es mucho mas rápido y tarda en realidad $O(n)$.

6.2.2 HeapSort

Una vez resueltos todos los asuntos concernientes al Heap podemos ver el algoritmo de HeapSort. La idea es remover el mayor elemento que es siempre la raiz del Heap, una vez seleccionado el máximo lo intercambiamos con el último elemento del vector, decrementamos la cantidad de elementos del Heap e invocamos Heapify a partir de la raiz.

Algorithm 26 HeapSort(A,n). Ordena el vector A de n elementos.

```
BuildHeap(A,n)
m ⇐ n
while m ≥ 2 do
  Swap(A[1],A[m])
  m ⇐ m − 1
  Heapify(A,1,m)
end while
```

Análisis

Como vemos hacemos $n-1$ llamadas a Heapify, cada llamada es $O(\log n)$. Por lo tanto el tiempo total es $O((n-1)\log n) = O(n \log n)$

BuildHeap revisado Para analizar el procedimiento BuildHeap en mayor detalle supongamos que $n = 2^{h+1} - 1$, donde h es la altura del árbol, es decir que trabajaremos con un árbol binario completo. (nota si h es la altura del árbol, el árbol tiene $h+1$ niveles).

El último nivel del árbol tiene 2^h nodos, todas las hojas residen en el nivel h.La raiz esta en el nivel 0.

Al llamar a Heapify el tiempo de ejecución depende de cuan lejos pueda ser llevado el elemento al ser empujado hacia abajo en el Heap, en el nivel mas bajo hay 2^h nodos, para ellos no invocamos a Heapify por lo que el trabajo es cero. En el siguiente nivel hay 2^{h-1} nodos, y cada uno puede descender a lo sumo un nivel. En el tercer nivel hay 2^{h-2} nodos y cada uno puede descender dos niveles. En general en el nivel j hay 2^{h-j} nodos y cada uno puede descender j niveles. Si contamos desde abajo hacia arriba nivel por nivel podemos calcular el tiempo total como.

$$T(n) = \sum_{j=o}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}$$

Factoreando el termino 2^h tenemos.

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

Esta es una sumatoria que no habiamos visto nunca hasta el momento, para resolverla hay que aplicar algunos trucos. Sabemos que.

$$\sum_{j=o}^{\infty} x^j = \frac{1}{1-x}$$

Derivando a ambos lados respecto de x y multiplicando por x.

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2}$$

$$\sum_{j=0}^{\infty} jx^{j} = \frac{x}{(1-x)^2}$$

Si reeemplazamos $x = 1/2$ obtenemos.

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2$$

Que es la sumatoria que necesitábamos, en nuestro caso no vamos hasta infinito pero como la serie infinita esta acotada la podemos usar para una aproximación muy válida.

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h \cdot 2 = 2^{h+1}$$

Como $n = 2^{h+1} - 1$ tenemos que $T(n) \leq n + q = O(n)$. Y claramente el algoritmo debe acceder por lo menos una vez a cada elemento del arreglo por lo que el tiempo de BuildHeap es $\Theta(n)$.

Esta es la segunda vez que un algoritmo con dos ciclos anidados termina resultando lineal si lo analizamos detalladamente, el secreto de BuildHeap reside en que el procedimiento es mas eficiente para los niveles mas bajos del árbol ya que alli realiza menos trabajo y en un árbol binario el 87.5 por ciento de los nodos del árbol residen en los tres ultimos niveles.

HeapSort HeapSort es un algoritmo de orden $\Theta(n \log n)$, que funciona "in-situ" pero no es estable.

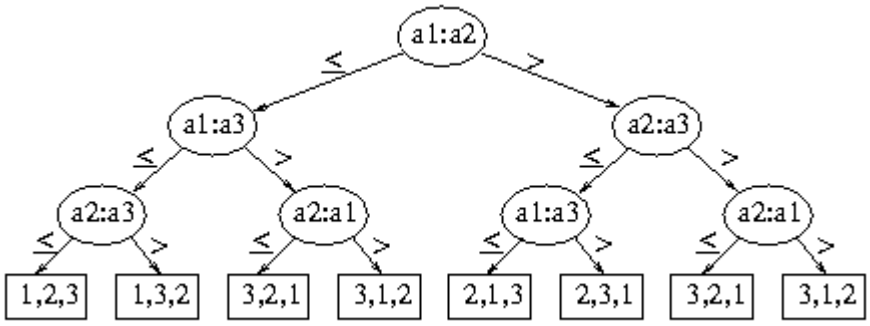
6.3 Límite inferior de los algoritmos de sort

Hasta ahora todos los algoritmos de sort que hemos analizado ya sea iterativos o recursivos se basan para ordenar en comparar elementos del vector entre si, si realizamos una abstracción podemos analizar en forma genérica cualquier algoritmo de sort basado en comparaciones.

Es lógico pensar que para ordenar un vector de n elementos hay una cierta cantidad de comparaciones que todo algoritmo debe realizar como mínimo para poder determinar el orden de los elementos del vector. Para realizar una asbtracción independiente de la técnica utilizada para ordenar vamos a representar a un algoritmo de sort genérico basado en comparaciones mediante un árbol de decisión.

Dada una secuencia de elementos $\{a1, a2, \ldots, an\}$ el orden entre los elementos puede determinarse usando las coparaciones $ai < aj$, $ai = aj$, $ai \leq aj$, $ai \geq aj$, $ai > aj$. De todas las comparaciones solamente una es suficiente para determinar el orden de un vector, por ejemplo podemos comparar $ai \leq aj$. En un arbol de decision cada nodo no-hoja representa una comparación a realizar entre dos elementos de la secuencia y las ramas son los caminos que se pueden seguir segun el resultado de la comparación. Las hojas del árbol de decisión representan el orden final del vector que se deduce a partir de las comparaciones realizadas.

Por ejemplo para tres elementos podemos realizar el siguiente árbol de decisión que permite determinar el orden de los elementos (ver figura).



Es de destacar que algunos de los nodos del árbol son inalcanzables, por ejemplo la cuarta hoja desde la izquierda indica que $a1 \leq a2$ y que $a1 > a2$ lo cual es imposible en cicunstancias normales. Esto explica porqué algunos algoritmos de sort son ineficientes, por realizar comparaciones de mas que resultan innecesarias.

Sea $T(n)$ la cantidad mínima de comparaciones que un algoritmo de sort debe realizar, observemos que $T(n)$ es exactamente la altura del arbol de decisión e indica cuantas comparaciones se hicieron hasta llegar a determinar cual es el ordenamiento de los elementos del vector. Un árbol binario como sabemos tiene a lo sumo $2^{T(n)}$ hojas distintas, que es la cantidad de resultados posibles del algoritmo de sort. A esta cantidad la llamaremos $A(n)$. El resultado del algoritmo es una cualquiera de las permutaciones posibles para n elementos, y hay $n!$ permutaciones posibles, por lo tanto el algoritmo debe ser capaz de generar $n!$ permutaciones posibles por lo que debe satisfacer.

$$A(n) = 2^{T(n)}$$

$$A(n) \geq n!$$

$$2^{T(n)} \geq n!$$

$$T(n) \geq \log n!$$

Para aproximar $n!$ se suele utilizar la aproximación de *Stirling*.

$$n! \geq \sqrt{2\pi n}(\frac{n}{e})^n$$

$$T(n) \geq \log(\sqrt{2\pi n}(\frac{n}{e})^n)$$

$$T(n) \geq \log\sqrt{2\pi n} + n\log n - n\log e \in \Omega(n\log n)$$

De acuerdo a lo anterior acabamos de probar que cualquier algoritmo de sort basado en la comparación de elementos es $\Omega(n \log n)$, con lo cual los algoritmos de MergeSort, QuickSort y HeapSort son asintoticamente óptimos, no se puede ordenar mediante comparaciones en un orden menor.

6.4 Algoritmos de sort lineales

Como sabemos mediante comparaciones no podemos ordenar una secuencia mas rapido que $\Omega(n \log n)$. Sin embargo algunos algoritmos pueden, realizando algunas hipótesis sobre los datos de entrada ordenarlos sin la necesidad de realizar comparaciones. Estos algoritmos solo funcionan en situaciones muy restrictivas por ejemplo enteros chicos, caracteres, notas de un examen, números de cuatro dígitos etc... Presentaremos dos algoritmos de sort que no realizan comparaciones, estos algoritmos como veremos son extremadamente rápidos.

6.4.1 Counting Sort

El algoritmo de counting sort se basa en la hipótesis de que los elementos son números enteros de rango $1..k$ el algoritmo funciona en orden $\Theta(n + k)$ .Si k es $\Theta(n)$ entonces el algoritmo resulta lineal.

La idea basica es determinar para cada elemento en el vector de entrdanda cual será su *rango* en el vector resultado. Una vez determinado el rango de cada elemento simplemente se lo copia en su posición y se obtiene el vector ordenado. La clave reside en determinar el rango de un elemento si compararlo contra los demás.

El método

El algoritmo utiliza tres vectores: $A[1..n]$ es el vector original, donde la clave de un elemento $A[j]$ es $A[j].key$. $B[1..n]$ es un vector de registros donde almacenaremos el resultado (claramente el algoritmo no funciona in-situ). $R[1..k]$ un vector de enteros, $R[x]$ es el rango de x en A donde $x \in [1..k]$.

El algoritmo es notablemente simple e ingenioso en primer lugar se construye R, esto se hace en dos pasos. En primer lugar inicializamos $R[x]$ como la cantidad de elementos de A cuya clave es igual a x. Esto se logra inicializando R en cero y luego por cada j de 1 hasta n incrementamos $R[A[j].key]$ en 1. Para determinar la cantidad de elementos menores o iguales a x reemplazamos luego $R[x]$ con la suma de los elementos en el vector $R[1..x]$.

Una vez armado R el vector contiene el rango de cada elemento, luego recorriendo el vector original se toma cada elemento y se accede a R para determinar su posicion en el vector resultado, luego de posicionar el elemento x en $B[R[x]]$ decrementamos $R[x]$ en una unidad.

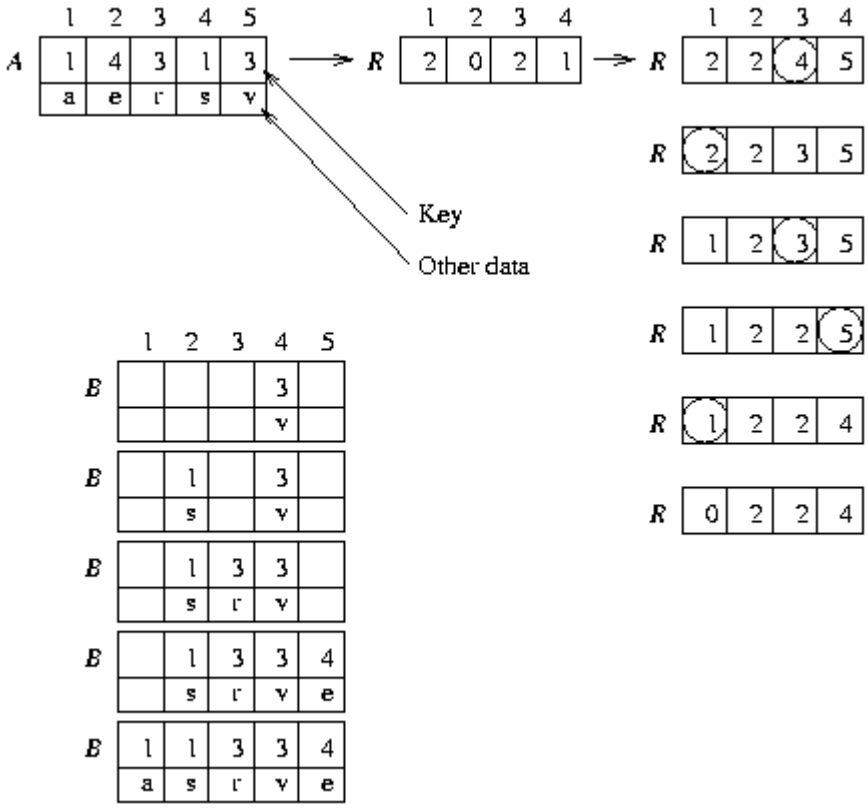
El algoritmo

Algorithm 27 CountingSort(A,B,n,k) Ordena A en B con n elementos y k posibles claves

```
for x = 1 to k do
   R[x] ⇐ 0
end for
for j = 1 to n do
   R[A[j].key] + +
end for
for x = 2 to k do
   R[x] ⇐ R[x] + R[x − 1]
end for
for j = n downto 1 do
   x ⇐ A[j].key
   B[R[x]] ⇐ A[j]
   R[x] ⇐ R[x] − 1
end for
```

Análisis

El algoritmo consta de cuatro ciclos no-anidados que insumen $O(k + n + (k - 1) + n)$ por lo que el orden del algoritmo es $O(n + k)$ si $kesO(n)$ entonces el algoritmo es $\Theta(n)$. La figura muestra un ejemplo del algoritmo, se recomienda realizar un seguimiento para convencerse de su funcionamiento.



El algoritmo de *CountingSort* es un algoritmo estable, esto se debe al ultimo ciclo que corre desde n hasta 1, si se hiciera desde 1 hasta n no seria estable. Para ordenar enteros de tamanio chico el algoritmo funciona en orden lineal y es claramente recomendable.

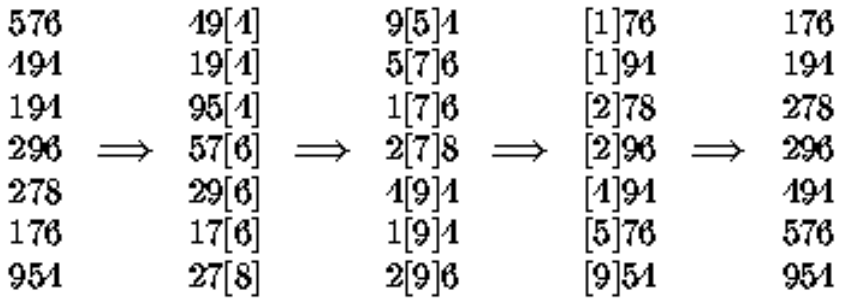
6.4.2 Radix Sort

La principal desventaja del algoritmo *CountingSort* es que solamente es aplicable, debido al costo espacial, para enteros de tamanio chico. Si tenemos enteros que pueden valer entre 1 y1000000 no vamos a querer contar con un vector auxiliar de un millón de posiciones. Radix sort provee un metodo para ordenar enteros grandes ordenando dígito por dígito. Una aplicacion típica de Radix-Sort es ordenar strings considerando que cada digito es un caracter (1 byte) del string. El algoritmo es extremadamente simple y funciona en base al mismo principio que usaban las maquinas que servian para ordenar tarjetas perforadas hace ya algunos anios. En primer lugar ordena el vector de acuerdo al ultimo dígito de cada clave, luego repite el proceso para los restantes dígitos usando

un algoritmo de sort **estable**. El algoritmo de *CountingSort* es ideal para esta parte.

Algorithm 28 RadixSort(A,n,d) Ordena A de n elementos con claves de d digitos

for $i = 1$ to d **do**
  Sort A en forma estable con respecto al iesimo digito.
end for



Análisis

El orden del algoritmo es $\Theta(dn)$, por lo que se trata de un algoritmo de orden lineal. Hay que tener cuidado ya que el algoritmo se aplica unicamente cuando el tamanio de los números a ordenar es fijo en cuanto a la cantidad de dígitos, si no fuera asi la cantidad de digitos de un numero n es $d = \log n$ y el orden pasa a ser $\Theta(n \log n)$. Por lo que equivale al *QuickSort*, *HeapSort*, aunque con un overhead mucho mayor que estos ultimos. *RadixSort* usando *CountingSort* es extremadamente eficiente para ordenar numeros de una cierta cantidad fija de dígitos o bien para ordenar strings de una cierta cantidad de caracteres. Ademaás es necesario contar con una buena cantidad de elementos a ordenar ya que los factores constantes que acompanian al algoritmo de RadixSort suelen ser grandes y las ventajas del método recién se hacen evidentes para colecciones de datos relativamente grandes.

6.5 Resumen y puntos clave

El propósito de esta clase ha sido resumir varios conceptos sobre algoritmos de sort que se habian visto a lo largo del curso y también aprender algunos métodos nuevos. En primer lugar hemos repasado los algoritmos ya conocidos, como por ejemplo el burbujeo, la selección, la inserción, el quicksort, y el mergesort. Luego estudiamos en detalle el HeapSort, un algoritmo basado en una estructura de datos especial. Para generalizar demostramos que ningún algoritmo de sort que

se base en la comparación de claves puede ordenar una secuencia en menos de $\Omega(n \log n)$. Para finalizar observamos que en situaciones especiales se pueden aplicar metodos de sort que no necesitan comparar claves para ordenar un conjunto de elementos. *CountingSort* y *RadixSort* son dos algoritmos de orden lineal que pueden aplicarse en algunas situaciones.

Algoritmo	Peor Caso	Caso Medio	Mejor Caso	Estable	In-Situ
Burbujeo	$O(n^2)$	$O(n^2)$	$O(n^2)$	Si	Si
Seleccion	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Si
Insercion	$O(n^2)$	$O(n^2)$	$O(n)$	Si	Si
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Si	No
QuickSort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	No	Si
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Si
CountingSort	$O(n)$	$O(n)$	$O(n)$	Si	No
RadixSort	$O(n)$	$O(n)$	$O(n)$	Si	No

6.6 Ejercicios

1. El siguiente algoritmo de sort fue creado por Howard.

Algorithm 29 STOOGE-SORT(A,i,j)

if $A[i] > A[j]$ **then**
  Swap(A[i],A[j])
end if
if $i + 1 \geq j$ **then**
  return
end if
$k \leftarrow \lfloor (j - i + 1)/3 \rfloor$
STOOGE-SORT(A,i,j-k)
STOOGE-SORT(A,i+k,j)
STOOGE-SORT(A,i,j-k)

   (a) Validar el algoritmo.

   (b) Dar una recurrencia para el peor caso de STOOGE-SORT.

   (c) Analizar el algoritmo y compararlo con los demás algoritmos estudiados, indicar si este algoritmo merece atención.

2. Indicar como implementar una cola FIFO usando Heaps, indicar como implementar una pila usando Heaps.

3. Dar un algoritmo $O(\log n)$ para el procedimiento Heap-Increase-Key(A,i,k). Que asigna a $A[i]$ el máximo entre $A[i]$ y k y además actualiza la estructura del Heap en forma acorde.

4. Un Heap n-ario es similar a los heaps estudiados pero cada nodo en lugar de tener 2 hijos puede tener hasta n.

   (a) Indicar como se representaría este tipo de Heap en un vector.

   (b) Indicar cual es la altura de un Heap d-ario con n elementos. Expresarlo en función de n y d.

   (c) Indicar como implementar Extract-Max en un Heap d-ario, analizar el orden del algoritmo en función de d y n.

   (d) Indicar como implementar Insert en un Heap d-ario, analizar el orden del algoritmo en función de d y n.

Chapter 7

Grafos

7.1 Grafos

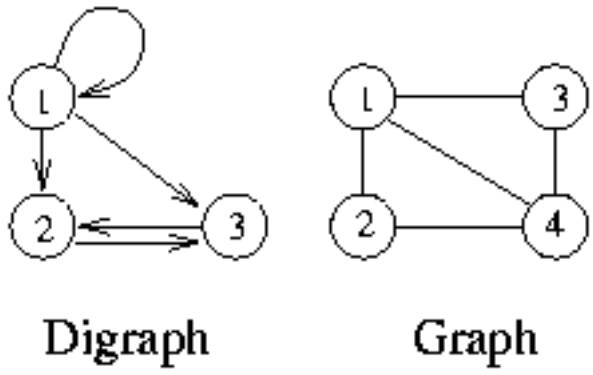
El proposito de esta clase es prestar atención a aquellos algoritmos relacionados con grafos. Basicamente un grafo es un conjunto de "nodos" o "vértices" interconectados por un conjunto de "aristas". Los grafos son estructuras discretas muy importantes ya que permiten modelar un sinnumero de situaciones. Practicamente cualquier problema que presente una serie de objetos y relaciones inter-objeto puede representarse adecuadamente con un grafo. Las aplicaciones que involucran el uso de grafos son tan numerosas que no podríamos iniciar una lista sin que la misma quedara sumamente incompleta. Para comenzar vamos a revisar algunas definiciones sobre grafos.

7.2 Introducción a la Teoría de Grafos

7.2.1 Grafos y Grafos Dirigidos

Un **grafo dirigido** o **digrafo** $G = (V, E)$ consiste de una serie finita de vértices V (también llamados nodos) y una relación binaria sobre V que son las aristas E. Cada arista es un par ordenado de vértices.

En la figura mostramos un grafo dirigido (izquierda). Observemos que los "loops" estan permitidos por la definición. Múltiples aristas no son permitidas pero las aristas (v, w) y (w, v) son distintas. El grafo dirigido $G = (V, E)$ esta caracterizada por $V = \{1, 2, 3\}$ y $E = \{(1, 1), (1, 2), (2, 3), (3, 2), (1, 3)\}$.



En un**grafo no dirigido** o simplemente, grafo, $G = (V, E)$ las aristas estan representadas por un par no-ordenado de vértices **distintos**, por lo tanto los "loops" no estas permitidos. En la figura el grafo de la derecha es un grafo no dirigido caracterizado por $V = \{1, 2, 3, 4\}$ y $E = \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$.

7.2.2 Vértices adyacentes

Decimos que un vértice w es **adyacente** a un vértice v si existe una arista de v hacia w. En un grafo no-dirigido decimos que una arista es **incidente** en un vértice si el vértice es punto de llegada o partida de una arista. En un grafo dirigido decimos que una arista entra o sale de un vértice por lo que hablamos de aristas **entrantes** o **salientes**.

7.2.3 Grafos pesados

Un grafo es **pesado** si sus aristas estan rotuladas con valores numéricos lamados **pesos**. El significado del peso depende de la aplicación, por ejemplo puede ser la distancia entre los vértices, la capacidad de la arista, etc.

7.2.4 Grado de un vértice

En un digrafo el número de aristas que salen de un determinado vértice determina el **grado de salida** (Gr_s) del vértice, a su vez las aristas que entran al vértice determinan el **grado de entrada** (Gr_e) vértice . En un grafo no dirigido se habla simplemente del **grado** (Gr) de un vértice como la cantidad de aristas incidentes en dicho vértice. El **grado de un grafo** es el grado máximo de sus vértices.

$|E|$ indica la cantidad de aristas de un grafo.
$|V|$ indica la cantidad de vértices de un grafo.

Observación Para un grafo dirigido $G = (V, E)$

$$\sum_{v \in V} Gr_e(v) = \sum_{v \in V} Gr_s(v) = |E|$$

Observación Para un grafo no-dirigido. $G = (V, E)$

$$\sum_{v \in V} Gr(v) = 2|E|$$

7.2.5 Caminos y ciclos

En un grafo dirigido un **camino** es una secuencia de vértices $(v0, v1, \ldots, vn)$ en donde vale que (v_{i-1}, v_i) es una arista para $i = 1, 2, \ldots n$. La **longitud** del camino es la cantidad de aristas n del mismo. Decimos que w es **alcanzable** desde u si existe un camino donde $v0 = u, vn = w$. Todo vértice es alcanzable desde si mismo por un camino de longitud cero. Un camino es **simple** si todos sus vértices (excepto a veces el primero y el último) son distintos.

Un **ciclo** es un camino que contiene al menos una arista y para el cual vale que $vo = vn$, un ciclo es simple si, además, $v1, \ldots, vn$ son distintos. Un "loop" es un ciclo simple de longitud 1.

En un grafo no-dirigido definimos caminos y ciclos de la misma forma pero para un **ciclo simple** agregamos el requisito de que el ciclo visite al menos tres vértices distintos, esto sirve para eliminar el ciclo degenerado (v, u, v) que implicaría ir y venir siempre por la misma arista.

7.2.6 Ciclo Hamiltoniano

Un ciclo Hamiltoniano es un ciclo que visita todos los vértices de un grafo exactamente una vez. En un **camino Hamiltoniano** no hace falta regresar al vértice inicial.

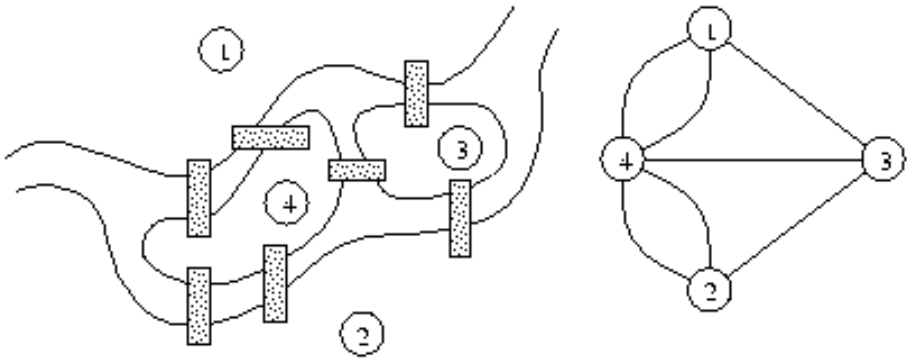
7.2.7 Ciclo Euleriano

Un ciclo Euleriano es un ciclo (no necesariamente simple) qye visita cada arista del grafo exactamente una vez. En un **camino Euleriano** no hace falta regresar al vértice inicial.

El problema de los puentes de Koningsberg

Uno de los problemas que motivó tempranamente el interés por la teoría de grafos es el *Problema del puente de Konigsberg*. Esta ciudad sobre el rio *Pregel* está unida por 7 (siete) puentes. El problema es si es posible cruzar los siete puentes sin pasar mas de una vez por cada puente. Euler mostró que tal cosa

era imposible enunciando las caracerísticas que debía tener un grafo para que exista un camino Euleriano.



Para que un grafo tenga un camino Euleriano todos salvo a lo sumo dos de los vértices deben tener grado par. En el grafo de Konigsberg los 4 vértices tienen grado impar.

7.2.8 Grafo acílico

Un grafo es acílico si no contiene ciclos simples.

7.2.9 Grafo conexo

Un grafo es conexo si todo vértice es alcanzable desde cualquier otro. Un grafo acíclico y conexo se denomina **árbol**, estos árboles se conocen tambien como **árboles libres** para destacar el hecho de que no poseen raiz. Un árbol es un grafo **minimamente conexo** ya que al eliminar una arista cualquiera del grafo el mismo deja de ser conexo. Además existe un único camino entre dos vértices de un árbol. El agregado de una arista a un árbol implica la creación de un ciclo.

7.2.10 Componentes conexos, árboles y bosques

La propiedad de vértice **alcanzable** es una relación de equivalencia entre los vértices, es decir que es reflexiva (un vértice es alcanzable desde sí mismo), es simétrica y es transitiva. Esto implica que la propiedad particiona los vértices del grafo en clases de equivalencia. Estas clases se denominan **componentes conexos**

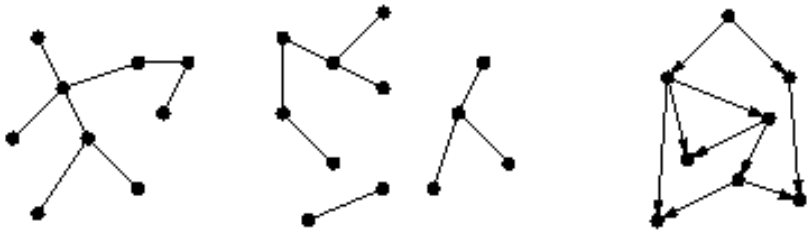
Un grafo conexo tiene un único componente conexo. Un grafo acíclico (no necesariamente conexo) consiste de una serie de árboles y se lo denomina **bosque**.

7.2.11 Grafos fuertemente conexos y componentes fuertemente conexos

Un grafo dirigido es **fuertemente conexo** si para cualquier par de vértices u y v, u es alcanzable desde v y v es alcanzable desde u. Los grafos fuertemente conexos particionan sus vértices en clases de equivalencia denominadas **componentes fuertemente conexos**

7.2.12 Grafo dirigido acíclico. (GDA)

Un grafo dirigido acíclico se denomina **GDA** (Grafo Dirigido Acíclico) y es distinto de un árbol dirigido.



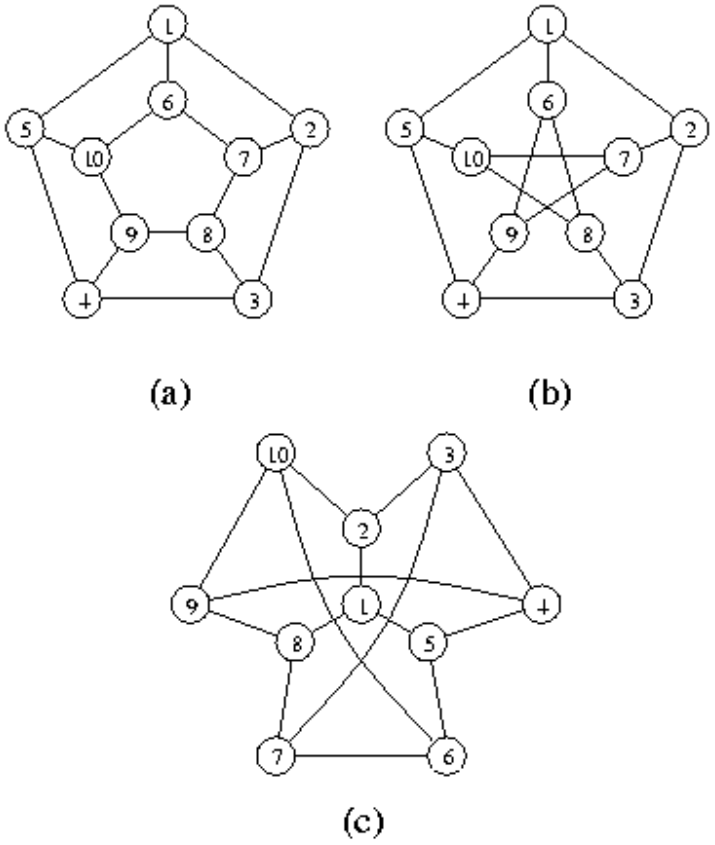
7.2.13 Grafos Isomorfos

Dos grafos $G = (E, V)$ y $G' = (E', V')$ son isomorfos si hay una función biyectiva $f : V \to V'$ tal que $(u, v) \in E$ si y solo si $(f(u), f(v)) \in E'$. Los grafos isomorfos son esencialmente "iguales" con la excepción de que sus vértices tienen distintos nombres.

Determinar si dos grafos son isomorfos no es tan sencillo como parece. Por ejemplo consideremos los grafos en la ilustración. Claramente (a) y (b) se parecen mas entre sí que con respecto a (c) sin embargo esto es solo una ilusión óptica. Observemos que los tres casos todos los vértices tienen grado 3 y que hay ciclos simples de longitud 4 en (a) pero que en (b) y (c) los ciclos simples mas chicos son de longitud 5. Esto implica que (a) no puede ser isomorfo a (b) ni a (c). Resulta que (b) y (c) sí son isomorfos. Uno de los isomorfismos

posibles es el que damos a continuacion, la notación $(u \to v)$ significa que el vértice u en el grafo (b) es mapeado al vértice v en el grafo (c).

$\{(1 \to 1), (2 \to 2), (3 \to 3), (4 \to 7), (5 \to 8), (6 \to 5), (7 \to 10), (8 \to 4), (9 \to 6), (10 \to 9)\}$



7.2.14 Subgrafos y grafo inducido

Un grafo $G' = (E', V')$ es un subgrafo de $G = (E, V)$ si $V' \subseteq V$ y $E' \subseteq E$. Dado un subconjunto $V' \subseteq V$ el **subgrafo inducido por V'** es el grafo $G' = (V', E')$ donde:

$$E' = \{(u, v) \in Etalqueu, v \in V'\}$$

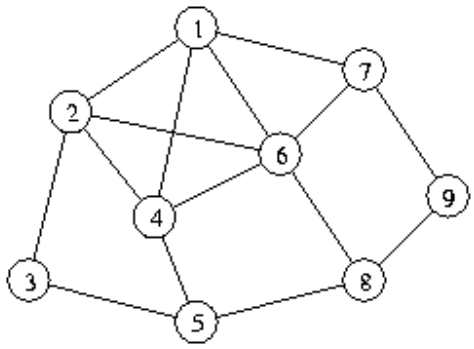
Es decir que se toman todas las aristas de G que inciden sobre vértices de V'.

7.2.15 Grafos completos. Cliques.

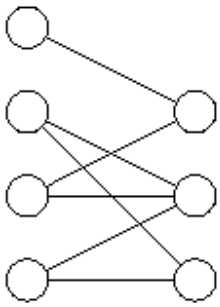
Un grafo no-dirigido que tiene el número máximo posible de aristas se denomina **grafo completo**. Los grafos completos se suelen denotar con la letra K. Por ejemplo K_5 es el grafo completo con 5 vértices. Dado un grafo G un subconjunto de vértices $V' \subseteq V'$ forma un **clique** su ek subgrafo inducido por V' es completo. En otras palabras si todos los vértices de V' son adyacentes unos con otros.

7.2.16 Conjunto independiente

Un subconjunto de vértices V' forman un **conjunto independiente** si el subgrafo inducido por V' no tiene aristas. Por ejemplo en la figura (a) el subconjunto $\{1, 2, 4, 6\}$ es un clique y el $\{3, 4, 7, 8\}$ es un conjunto independiente.



(a)



(b)

7.2.17 Grafo bipartito

Un grafo **bipartito** es un grafo no-dirigido en donde los vértices pueden ser particionados en dos conjuntos $V1$ y $V2$ de forma tal que todas las aristas van de un vértice de $V1$ a un vértice de $V2$. El grafo en la figura (b) es un grafo bipartito.

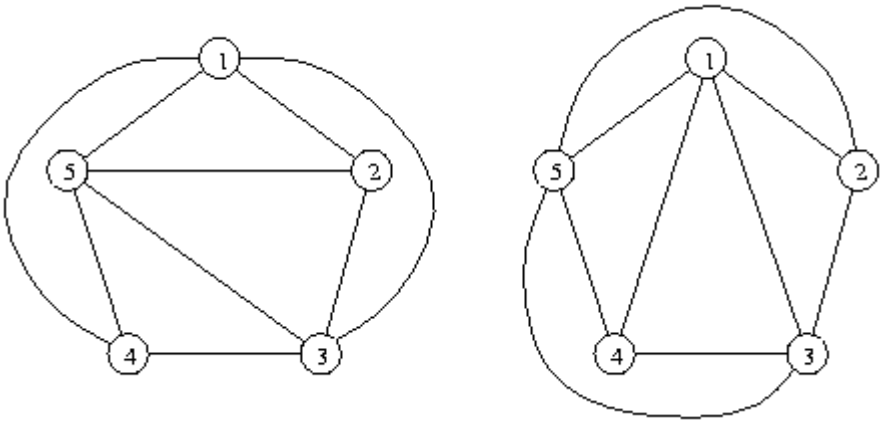
7.2.18 Grafo complemento, Grafo transpuesto

El **complemento** de un grafo $G = (E, V)$ denotado G^{-1} es un grafo sobre el mismo conjunto de vértices pero en el cual las aristas han sido complementadas. La **reversa** de un grafo dirigido denotado G^R es un grafo dirigido sobre el mismo conjunto de vértices en donde la dirección de las aristas ha sido cambiada, a este grafo se lo denomina tambien **grafo transpuesto** G^T

7.2.19 Grafos planares, Formas planares

Un grafo es **planar** si puede ser dibujado en el plano de forma tal que no se cruce ningún par de aristas. Los grafos planares son una familia importante de los grafos ya que se usan en muchas aplicaciones desde los sistemas de información geográfica hasta circuitos y modelado de sólidos.

En general hay muchas formas de dibujar un grafo planar en dos dimensiones. Por ejemplo en la próxima figura mosramos dos dibujos esencialmente diferentes del mismo grafo. Cada uno de estos dibujos se denomina **forma planar**. Los **vecinos** de un vértice son los vértices adyacentes al mismo. Una forma planar esta determinada para el ordenamiento antohorario de los vecinos de cada uno de los vértices. Por ejemplo en la forma de la izquierda los vecinos del vértice uno en sentido antohorario son $\{2, 3, 4, 5\}$ pero en la forma de la derecha el ordenamiento es $\{2, 5, 4, 3\}$ y por eso ambas formas son diferentes.



Caras

Un hecho importante acerca de las formas planares de un grafo es que subdividen al grafo en regiones denominadas **caras**. Por ejemplo en la figura de la izquierda la figura triangular limitada por los vértices $\{1, 2, 5\}$ es una cara. Siempre existe una cara denominada **cara externa** que rodea a todo el grafo. Las formas planares de las figuras tienen por ejemplo 6 caras en ambos casos. Dos formas planares siempre tienen el mismo número de caras, esto se desprende de la **fórmula de Euler**

Fórmula de Euler

Una forma planar de un grafo conexo con V vértices, E aristas y F caras satisface:

$$V - E + F = 2$$

En los ejemplos ambos grafos tienen 5 vértices y 9 aristas y por lo tanto de la formula de Euler deben tener $F = 2 - V + E = 2 - 5 + 9 = 6$ caras.

7.2.20 Tamanios

Al referirnos a grafos y digrafos vamos a utilizar que $n = |V|$ y $e = |E|$. Los tiempos de ejecución de los algoritmos que operan con grafos suelen depender de n y e por lo que se bueno ver que relación existe entre ambos números.

Observación Para un grafo dirigido $e \leq n^2 = O(n^2)$. Para un grafo no dirigido $e \leq \begin{pmatrix} n \\ 2 \end{pmatrix} = n(n-1)/2 = O(n^2)$

7.2.21 Grafos dispersos

Decimos que un grafo es disperso cuando e es mucho mas chico que n^2. Para el importante caso de los grafos planares $e = O(n)$. En la mayoría de las aplicaciones los grafos muy grandes tienden a ser dispersos. Esto es importante al diseniar algoritmos que utilicen grafos porque, cuando n es realmente grande un tiempo de respuesta del orden de $O(n^2)$ a menudo es inaceptable.

7.3 Representación de Grafos

Los algoritmos que trabajan sobre grafos deben contar con una estructura de datos apropiada para la representación de datos, vamos a describir dos formas de describir grafos ya sea dirigidos o no dirigidos. Por ejemplo sea el grafo dirigido $G = (E, V)$ con $n = |V|$ y $e = |E|$. Asumimos que los vértices de G son $\{1, 2, \ldots, n\}$.

7.3.1 Matriz de adyacencias

Es una matriz de nxn definida para $1 \leq v, w \leq n$ donde.

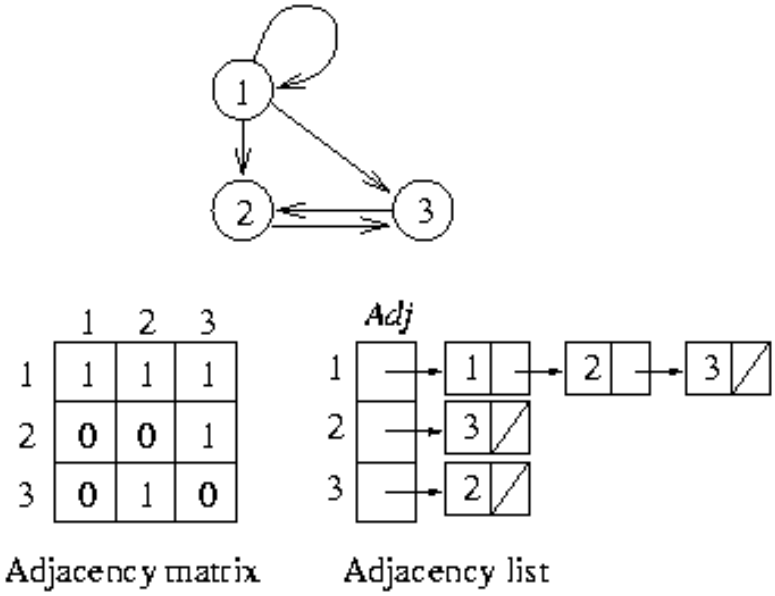
$$A[v, w] = \{ \begin{array}{ll} 1 & si (v, w) \in E \\ 0 & sino \end{array}$$

Si el grafo dirigido tiene pesos podemos almacenar los pesos en la matriz. Por ejemplo si $(v, w) \in E$ entonces $A[v, w] = W(v, w)$ (el peso de la arista (v, w)).

Si $(v, w) not \in E$ entonces en general $W(v, w)$ no necesita ser definido pero a menudo se inicializa en un valor especial como por ejemplo -1 o ∞ [1]

7.3.2 Lista de Adyacencias

Es un vector $Adj[1..n]$ de punteros donde para $1 \leq v \leq n, Adj[v]$ apunta a una lista enlazada que contiene los vértices que son adyacentes a v (los vértices que son alcanzables desde v usando una sola arista). Si las aristas tienen pesos los pesos pueden almacenarse en las listas.

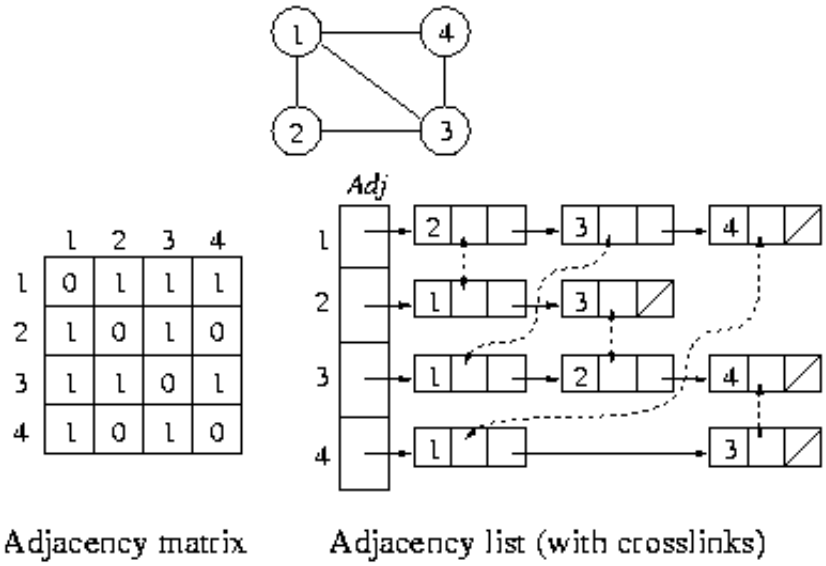


Los grafos no-dirigidos se pueden representar de la misma forma pero estaríamos guardando cada arista dos veces. En particular representaríamos la arista (v, w) como el par de aristas $(v, w) y (w, v)$.

Esto puede causar algunas complicaciones, por ejemplo un algoritmo que marca aristas de un grafo deberia tener cuidado de marcar dos aristas en lugar de una sola si el grafo es no-dirigido ya que las dos aristas en realidad representan la misma.

Al almacenar la **lista de adyacencias** puede resultar poco eficiente recorrer todas las listas por lo que suelen incluirse **cross links** como se muestra en la ilustración.

[1] ∞ obviamente es un número que es mayor que el peso máximo posible, o el valor máximo almacenable en una variable del tipo que se esta usando



Una matriz de adyacencias requiere $\Theta(n^2)$ [2] espacio, una lista de adyacencias requiere $\Theta(n+e)$. Para grafos dispersos la lista de adyacencias es mas eficiente. En la mayoría de las aplicaciones tanto la matriz de adyacencias como la lista de ayacencias son representaciones útiles para un grafo.

7.4 Algoritmos para Grafos

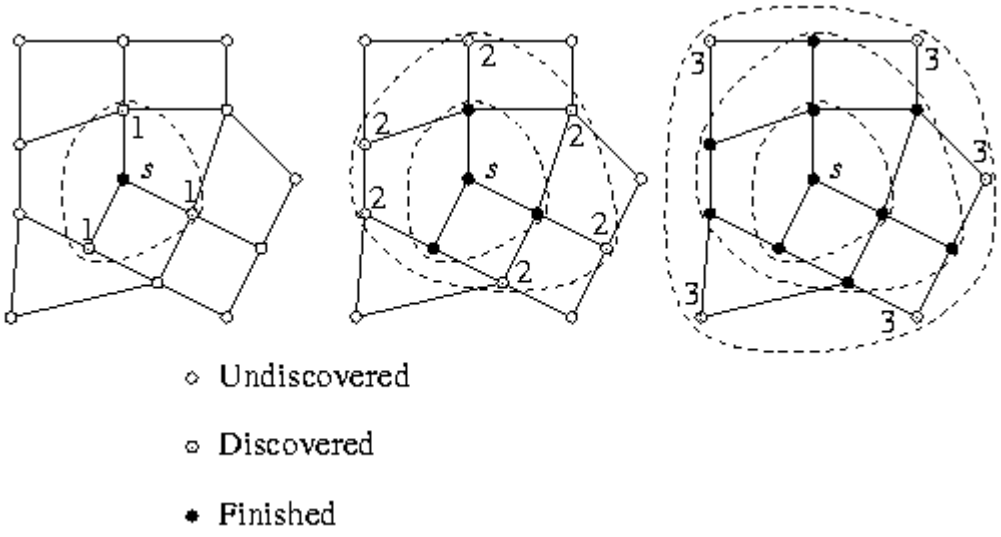
Una vez repasadas algunas definiciones y estudiado distintas formas de representar un grafo debemos empezar a analizar los algoritmos utilizados para operar con grafos, vamos a empezar con un algoritmo muy necesario que es aquel que permite recorrer todos los vértices de un grafo. El objetivo del algoritmo es recorrer todos los vértices del grafo,pasando sola una vez por cada vértice. Esta tarea que era muy simple en un árbol [3] resulta mas dificultosa en un grafo debido a la natural falta de estructura de los mismos. Hay dos técnicas para recorrer un grafo, el algoritmo **BFS** (Breadth-first-search) y **DFS** (Depth first search).

7.4.1 BFS

El algoritmo BFS se basa en recorrer el grafo "a lo ancho primero", Dado un grafo $G = (V, E)$ BFS comienza con un vértice cualquiera s y "descubre" todos los vértices que son adyacentes a s. En un momento cualquiera del algoritmo existe una "frontera" de vértices que han sido descubiertos pero no procesados.

[2] Estamos usando tambien esta notacion para el costo en espacio!
[3] Recordemos los algoritmos para recorrer un árbol: pre-order, post-order e in-order

Inicialmente todos los vértices excepto el *origen* son pintados de color blanco, lo cual quiere decir que aun no han sido observados. Cuando un vértice es descubierto se lo pinta de color gris (y pasa a ser parte de la forntera), cuando un vértice ya fue procesado (se descubrió toda su frontera) se lo pinta de color negro.



Algorithm 30 BFS(G=(E,V), s). Recorre el grafo G a partir del vértice s

```
int color[1..size(V)]
cola Q
for each u in V do
   color[u] ⇐ blanco
end for
color[s] ⇐ gris //Inicializamos el origen.
encolar(Q, s)
while Q not empty do
   u ⇐ desencolar(Q)
   for each v in Adj[u] do
      if color[v]==blanco then
         color[v] ⇐ gris
         encolar(Q, v)
      end if
   end for
   color[u] ⇐ negro
end while
```

BFS hace uso de una *cola FIFO* donde los elementos son obtenidos en el mismo orden en que fueron insertados. Mantienen tambien el vector $color[u]$ que contiene el color del vértice u (blanco=no observado, gris= observado pero no procesado, negro= procesado). El funcionamiento del algoritmo puede requerir realizar algunos seguimientos para poder comprenderlo en totalidad.

Distancia mínima usando BFS

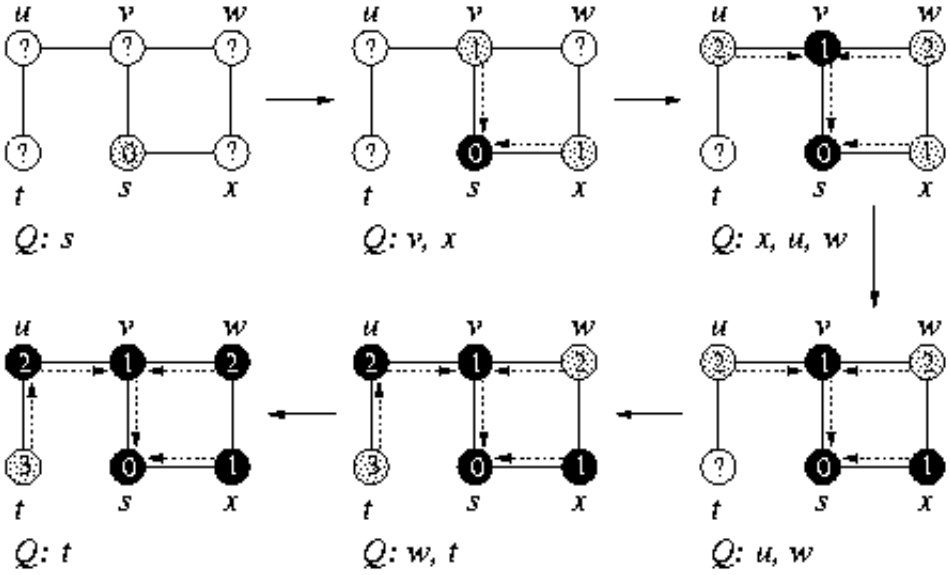
BFS suele ser muy utilizado para calcular la distancia mínima desde un vértice dado hasta cualquier otro de un grafo no-dirigido, esto es muy sencillo de hacer porque BFS recorre los vértices en orden de distancia "creciente" a partir del origen. Esto se logra realizando las siguientes modificaciones al algoritmo.

Algorithm 31 BFS(G=(E,V), s). Recorre el grafo G a partir del vértice s

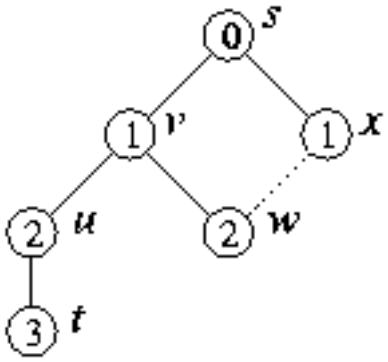
```
int color[1..size(V)]
cola Q
int d[1..size(V)] //vector de distancias.
int pred[1..size(V)] //predecesor de cada vértice
for each u in V do
   color[u] ⇐ blanco
   d[u] ⇐ INFINITO
   pred[u] ⇐ NULL

end for
color[s] ⇐ gris //Inicializamos el origen.
d[s] ⇐ 0
encolar(Q, s)
while Q not empty do
   u ⇐ desencolar(Q)
   for each v in Adj[u] do
      if color[v]==blanco then
         color[v] ⇐ gris
         d[v] ⇐ d[u] + 1
         pred[v] ⇐ u
         encolar(Q, v)
      end if
   end for
   color[u] ⇐ negro
end while
```

Hemos agregado un vector de distancias que para cada vértice v indicara a que distancia esta dicho vértice del vértice origen s, además para saber cual es

el camino que debemos hacer para obtener dicha distancia almacenamos en cada vértice cual es el vértice anterior al mismo en el camino mínimo que parte de s y llega a dicho vértice. Esto es valido debido al **principio de optimalidad** que dice que si un camino es mínimo entonces todos sus sub-caminos también lo son.



En la figura podemos observar que los punteros predecesores de BFS definen un arbol invertido. Si invertimos las aristas obtenemos un árbol desordenado denominado **Árbol BFS de G para s**.Las aristas de G que forman el árbol se denominan **aristas del árbol** y el resto de las aristas se denominan **aristas de cruce**. En la figura podemos observar el arbol BFS para nuestro ejemplo.



Análisis

El tiempo de ejecución de BFS es muy similar al tiempo de ejecución de muchos algoritmos que recorren grafos. Si $n = |V|$ y $e = |E|$. Observamos que la inicialización requiere $\Theta(n)$. El ciclo que recorre el grafo es el núcleo del algoritmo. Como nunca visitamos un vértice dos veces la cantidad de veces que ejecutamos en el ciclo *while* es a lo sumo n (es exactamente n si todos los vértices son alcanzables desde s). El número de iteraciones dentro del ciclo interior es proporcional a $grado(u) + 1$[4]. Sumando para todos los vértices tenemos que el tiempo insumido es:

$$T(n) = n + \sum_{v \in V} (grado(u) + 1 = n + \sum_{v \in V} grado(u) + n = 2n + 2e = \Theta(n + e)$$

Para un grafo dirigido el análisis es esencialmente el mismo.

7.4.2 DFS

Una de las mayores dificultades al diseniar algoritmos para grafos es la falta de estructura de los mismos. Al ordenar un vector, por ejemplo, vimos que era fácil particionar el problema para aplicar un algoritmo de tipo "Divide & Conquer". Sin embargo no es en absoluto claro como dividir un grafo en subgrafos. Lo importante de cualquier estrategia para recorrer grafos es la habiliad para imponer algun tipo de estructura al grafo. En BFS vimos como podemos ver al grafo como un árbol (el árbol BFS) mas un conjunto de aristas (aristas de cruce). Los árboles son objetos mucho mas estructurados que los grafos, por ejemplo los arboles se pueden subdividir elegantemente en sub-arboles para los cuales el problema es resuelto en forma recursiva. En un grafo aun nos quedaría el problema de como manejar las aristas que no forman parte del arbol BFS. El

[4] El $+1$ es debido a que si $grado(u) = 0$ necesitamos una cantidad de tiempo constante para incializar el ciclo

algoritmo **DFS** (Depth first search), tiene como propiedad inetersante que las aristas no-arboladas definen una estructura matematicamente manejable.

Consideremos el problema de buscar un tesoro en un castillo. Para resolver podríamos usar la siguiente estrategia. Cada vez que entramos a una habitación del castillo pintamos un graffiti en la pared para saber que ya estuvimos alli [5]. Sucesivamente viajamos de habitación en habitación mientras lleguemos a habitaciones en las cuales nunca estuvimos. Una vez que volvemos a una habitación en la cual ya estuvimos probamos con una puerta diferente a la usada anteriormente. Cuando todas las puertas han sido utilizadas volvemos a la habitación anterior y repetimos el procedimiento.

Notemos que este procedimiento es descripto en forma recursiva. En particular cuando entramos a una nueva habitación estamos comenzando una nueva búsqueda. Este es el principio de funcionamiento del algoritmo DFS.

Algoritmo DFS Supongamos que tenemos un grafo dirigido $G = (V, E)$ (el mismo algoritmo funciona para grafos no dirigidos pero la estructura resultante es distinta). Vamos a usar cuatro vectores auxiliares. Como antes se mantiene un color por cada vértice, blanco significa vértice no descubierto (habitación sin graffiti en la pared). Gris significa vértice descubierto pero no procesado (tiene graffiti pero aun no usamos todas sus puertas). Negro significa vértice ya procesado. Al igual que en BFS tambien almacenamos los punteros al vértice predecesor apuntando hacia atras hacia el vértice a partir del cual descubrimos el vértice actual. Tambien vamos a asociar dos números a cada vértice que denominamos **time stamps**. Cuando descubrimos un vértice u por primera vez guardamos un contador en $d[u]$ y cuando terminamos de procesar un vértice guardamos un contador en $f[u]$. El propósito de estos contadores sera explicado mas adelante.

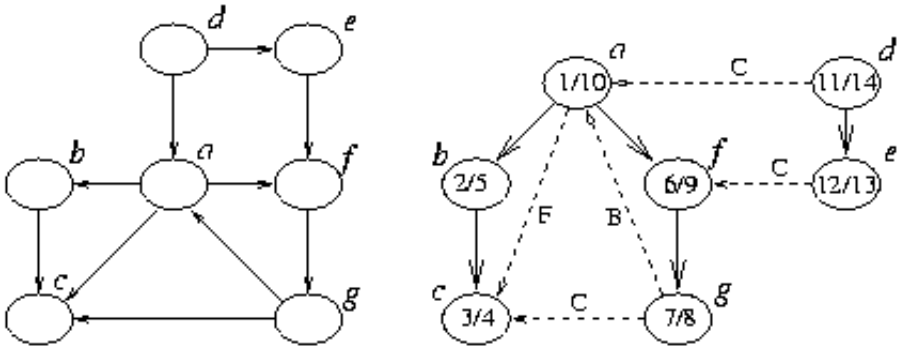
Algorithm 32 DFS(G). Recorre el grafo G en orden DFS.

```
for each u in V do
   color[u] ⇐ blanco
   pred[u] ⇐ NULL
end for
time ⇐ 0
for each u in V do
   if color[u]==blanco then
      DFSVisit(u)
   end if
end for
```

[5] Por favor no intenten esto en sus casas

Algorithm 33 DFSVisit(G,u).

$color[u] \Leftarrow gris$
$d[u] \Leftarrow time + 1$
for each v in Adj(u) **do**
  if color[v]==blanco **then**
    $pred[v] \Leftarrow u$
    DFSVisit(v)
  end if
end for
$color[u] \Leftarrow negro$
$f[u] \Leftarrow time + 1$



Análisis

El tiempo de ejecución de DFS es $\Theta(n + e)$. Esto es bastante mas difícil de observar con respecto al análisis de BFS debido a la naturaleza recursiva del algoritmo. Normalmente una recurrencia es una buena forma de calcular el tiempo de ejecución del algoritmo pero esto no ocurre aqui porque no tenemos noción del tamanio de cada llamada recursiva. En primer lugar observemos que ignorando las llamadas recursivas DFS corre en $O(n)$. Observemos que cada vértice es visitado exactamente una vez en la búsqueda y que por lo tanto $DFSVisit()$ es llamado exactamente una vez por cada vértice. Podemos analizar cada llamada individualmente y sumar sus tiempos de ejecución. Ignorando el tiempo necesario para las llamadas recursivas podemos ver que cada vértice u puede ser procesado en $O(1 + Grado_{salida}(u))$. Por lo tanto el tiempo total del procedimiento es.

$$T(n) = n + \sum_{uinV} (Grado_{salida}(u) + 1) = n + \sum_{uinV} Grado_{salida}(u) + n = 2n + e = \Theta(n + e)$$

Clasificación de aristas luego de DFS

El algoritmo DFS impone en el grafo una estrutura de árbol (en realidad una colección de árboles). Este árbol no es mas que el árbol de recursión donde la arista (u, v) surge cuando porcesando el vértice u llamamos a $DFSVisit(v)$. Las aristas que forman parte del árbol de recursión son llamadas **aristas del árbol DFS**. Para grafos dirigidos las restantes aristas del grafo se pueden clasificar de la siguiente manera.

Aristas de retroceso (u, v) donde v es un antecesor (no necesariamente propio) de u en el árbol. Por lo tanto un "loop" es considerado una arista de retroceso.

Aristas de avance (u, v) donde v es un sucesor de u en el árbol.

Aristas de cruce (u, v) donde u y v no son ni antecesor ni sucesor entre si. Aun mas, la arista podría conectar dos árboles distintos.

No es difícil clasificar las aristas de un árbol DFS analizando el valor del color de cada vértice y/o considerando los time-stamps.

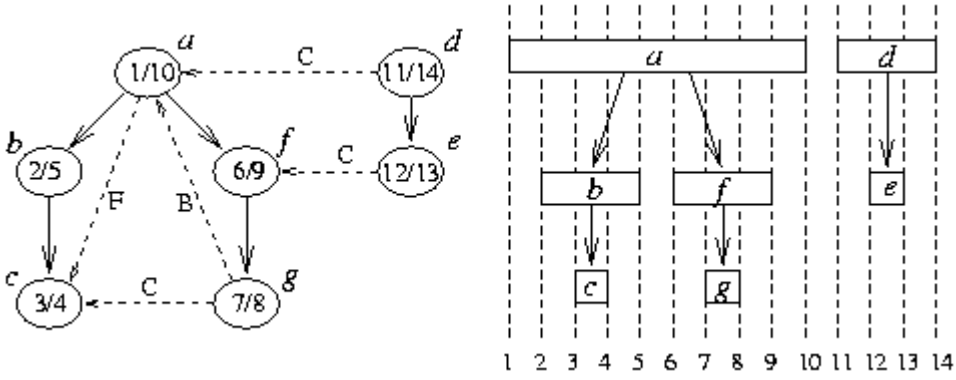
En un grafo no-dirigido no hay distinción entre aristas de avance y de retroceso por lo que las llamaremos aristas de retroceso en forma uniforme. Ademas no pueden existir aristas de cruce.

Estructuras a partir de los Time-Stamps Los time-stamps tambien imponen al grafo una estrutura interesante, en algunos libros esta estructura se conoce como estructura de intervalos.

Teorema: (Teorema de los intervalos). Dado un digrafo $G = (V, E)$ y un arbol DFS para G y dos vértices cualesquiera $u, v \in V$.

- u es un sucesor de v si y solo si $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u es un antecesor de v si y solo si $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u no tiene relación con v si y solo si $[d[u], f[u]]$ y $[d[v], f[v]]$ son disjuntos.

Notación: [x,y] es un intervalo perteneciente a los números naturales.



Ciclos Los time-stamps dados por el algoritmo DFS permiten determinar muchos datos importantes sobre un grafo dirigido o no-dirigido. Por ejemplo podemos determinar si el grafo contiene o no ciclos. Esto lo podemos hacer gracias a los siguientes lemas.

Lema: Dado un digrafo $G = (V, E)$ consideremos cualquier arista $(u, v) \in E$. Si la arista es parte de un árbol DFS, es una arista de avance o es una arista de cruce entonces $f[u] > f[v]$. Si la arista es de retroceso entonces $f[u] \leq f[v]$.

Demostración: Para un árbol o arista de avance o retroceso la demostración surge directamente del teorema de los intervalos. (Por ejemplo para una arista de avance (u, v) v es un sucesor de u, y por lo tanto el intervalo de v esta contenido dentro del de u implicando que $f[u] > f[v]$ pues $f[v]$ fue rotulado antes que $f[u]$.). Para una arista de cruce (u, v) sabemos que los intervalos son disjuntos. Cuando procesamos u, v no fue blanco (sino (u, v) sería una arista del árbol), eso implica que v fue inspeccionado antes que u, y como los intervalos son disjuntos v tiene que haber finalizado antes que u.

Lema: Dado un digrafo $G = (V, E)$. G tiene un ciclo si y solo si existe una arista de retroceso.

Demostracion: ($\Leftarrow$) Si hay una arista de retroceso (u, v) entonces v es un predecesor de u y siguiendo las aristas del árbol DFS de v hacia u obtenemos un ciclo. ($\Rightarrow$) Supongamos que no hay aristas de retroceso. Por el teorema anterior cada uno de los tipos de aristas restantes tienen la propiedad de que van de vértices con rotulo f mayor que el valor del vértice hacia el cual se dirigen. Por lo tanto siguiendo cualquier camino el tiempo disminuye monotonamente implicando que no puede haber un ciclo.

Precaución: No hay relación alguna entre la cantidad de aristas de retroceso y el número de ciclos en un grafo.

Ordenamiento Topológico

Consideremos un GDA (grafo directo acíclico). El cual es muy utilizado en aplicaciones en las cuales se muestra precedencia entre objetos (tareas, eventos, procesos, etc). En este tipo de aplicaciones una arista (u, v) implica que la tarea u debe finalizarse antes de empezar la tarea v.

Un **ordenamiento topológico** de un GDA es un ordenamiento lineal de los vértices del grafo de forma tal que para cada arista (u, v) u aparezca antes que v en el ordenamiento. En general hay varios ordenamientos topológicos validos para un GDA.

Obtener el ordenamiento topológico es sencillo utilizando DFS. Por el lema anterior para cada arista (u, v) en un GDA el tiempo f de u es mayor que el tiempo f de v. Por lo tanto es suficiente devolver los vértices ordenados en orden decreciente de tiempo f. Esto puede hacerse en forma muy sencilla al mismo tiempo que se utiliza el algoritmo DFS. En el siguiente algoritmo hemos modificado el algoritmo DFS para que solamente realice el ordenamiento topológico del grafo dejando de lado todo lo demás.

Algorithm 34 TopSort(G). Realiza el ordenamiento topológico del grafo G.

```
Require: G no debe tener ciclos.
  for each u en V do
    color[u] ⇐ blanco
  end for
  L ⇐ new(LinkedList)
  for each u en V do
    if color[u]=blanco then
      TopVisit(u)
    end if
  end for
  return L
```

Algorithm 35 TopVisit(u,G).

```
  color[u] ⇐ gris
  for each v in Adj(u) do
    if color[v]=blanco then
      TopVisit(v)
    end if
  end for
  Append u a la cabeza de L
```

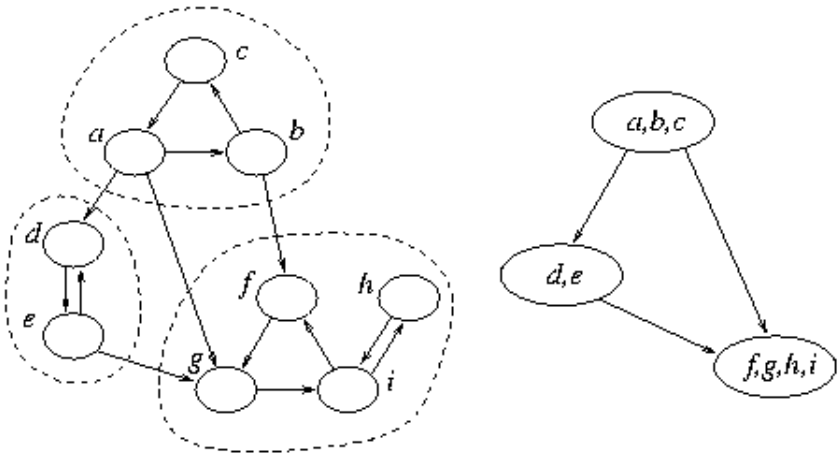
Análisis Al igual que el algoritmo DFS *TopSort* es de orden $\Theta(n+e)$

Componentes fuertemente conexos

Un problema importante realtivo a los grafos dirigidos es su conectividad, cuando este tipo de grafos son utilizados para modelar redes de comunicaciones o redes de transporte es deseable saber si dichas redes son completas en el sentido de que cualquier nodo es accesible desde cualquier otro. Esto se da cuando el digrafo que modela la red es **fuertemente conexo**.

Es deseable tener un algoritmo que determine si un grafo es fuertemente conexo. De hecho vamos a resolver un problema un tanto mas difícil que consiste en calcular los **componentes fuertemente conexos** cfc de un digrafo. En particular vamos a particionar el grafo en subconjuntos de vértices de forma tal que el subgrafo inducido por cada subconjunto es fuertemente conexo. Estos subconjuntos deberán ser lo mayor posibles en tamanio.

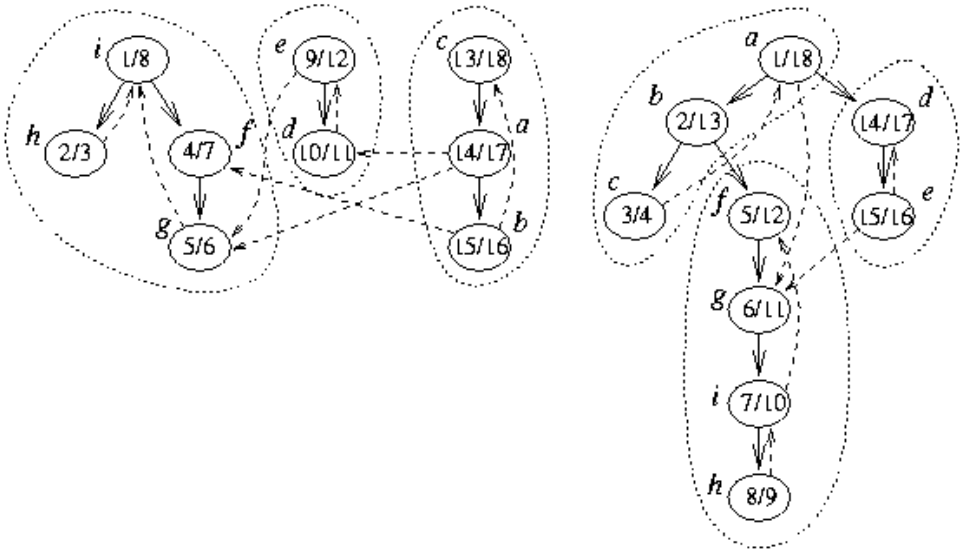
Si mergeamos los vértices de cada componente fuertemente conexo (cfc) en un **super vértice** y si juntamos dos supervértices (A, B) si y solo si existen vértices $u \in A$ y $v \in B$ tales que $(u, v) \in E$ el digrafo resultante llamado **digrafo componente** es necesariamente acíclico. Si fuera cíclico entonces los dos cfc's podrían formar un cfc de mayor tamanio.



Hemos introducido este tema debido a que el algoritmo que presentamos como solución es un verdadero ejemplo de disenio. Es extremadamente simple, eficiente e ingenioso, a tal punto que resulta difícil ver como funciona. Vamos a

dar algunas pautas intuitivas sobre el funcionamiento del algoritmo aunque, no vamos a llegar a demostrar formalmente su corrección.

Consideremos para entrar en tema el recorrido DFS del grafo ilustrado (izquierda). Observemos que cada cfc es un sub-arbol del bosque DFS. Esto no siempre es verdad para cualquier DFS pero siempre existe una forma de ordenar el DFS de forma tal que esto sea verdad.



Supongamos que conocemos el grafo componente por adelantado. [6], supongamos ademas que tenemos un ordenamiento topológico rebatido del grafo componente. Es decir que si (u, v) es una arista en el grafo comoponente entonces v aparece antes que u en el ordenamiento. Ahora corramos DFS pero cada vez que necesitamos un nuevo vértice para iniciar la búsqueda tomamos el próximo vértice disponible de la lista rebatida del ordenamiento topológico. Como consecuencia interesante de este procedimiento cada árbol en el bosque DFS será un cfc (magia?).

Revelando la magia Claramente una vez que DFS empieza dentro de un cfc debe visitar cada vértice del componente (y posiblemente otros) antes de finalizar. Si no empezamos en orden topologico inverso la búsqueda puede "escaparse" hacia otros cfc colocándolos en el mismo árbol DFS. En la figura de la

[6]Esto es ridiculo porque necesitariamos conocer los cfc y este es el problema que queremos resolver, pero permitanme la disgresion por ahora.

derecha, por ejemplo, la búsqueda empezóen el vértice a, no solo se visita dicho componente sino tambien otros componentes. Sin embargo visitando componentes en orden inverso de acuerdo al ordenamiento topológico impedimos que la búsqueda se escape hacia otros componentes.

Esto nos deja con la intuición de que si de alguna manera pudiesemos ordenar el DFS entonces tendríamos un algoritmo sencillo para calcular los cfc, sin embargo no sabemos el aspecto que tiene el grafo componente. El truco es que podemos encontrar un ordenamiento de los vértices que tiene la propiedad necesaria sin realmente calcular el grafo componente.

Desafortunadamente es difícil explicar como funciona este criterio. Presentaremos el algoritmo y sepan disculpar la falta de formalidad en su explicación. Recordemos que G^R es el grafo transpuesto.

Algorithm 36 StrongComp(G)

Run DFS(G), computando $f[u]$ para cada vertice u
$R \Leftarrow Reverse(G)$ //R es el grafo transpuesto.
Ordenar los vertices de R (counting sort) en orden decreciente de $f[u]$
Run DFS(R) utilizando dicho ordenamiento de los vertices.
Cada arbol DFS es un CFC

Todos los pasos del algoritmo son sencillos de implementar, y trabajan en $O(n+e)$

7.5 Resumen y puntos clave

En esta clase hemos cubierto toda la teoría que vamos a necesitar en el curso sobre Grafos, los grafos son estruturas particularmente importantes en el modelado de ciertos problemas y requieren de gran cuidado al diseniar algoritmos que empleen grafos. Los algoritmos que hemos estudiado en esta clase son algoritmos para recorrer todos los vértices de un grafo. En primer lugar presentamos el algoritmo BFS que recorre el grafo "a lo ancho primero", una aplicacion directa del algoritmo BFS puede ser utilizada para hallar caminos minimos en un grafo no-dirigido.

Luego vimos el algoritmo DFS, el cual sirve para establecer una estructura muy particular en el grafo, la cual permite realizar una buena clasificacion de las aristas del grafo. Entre las aplicaciones directas de DFS vimos que sirve para determinar si un grafo tiene ciclos y para realizar un ordenamiento topológico de los vértices de un grafo dirigido acíclico. El problema de los componentes

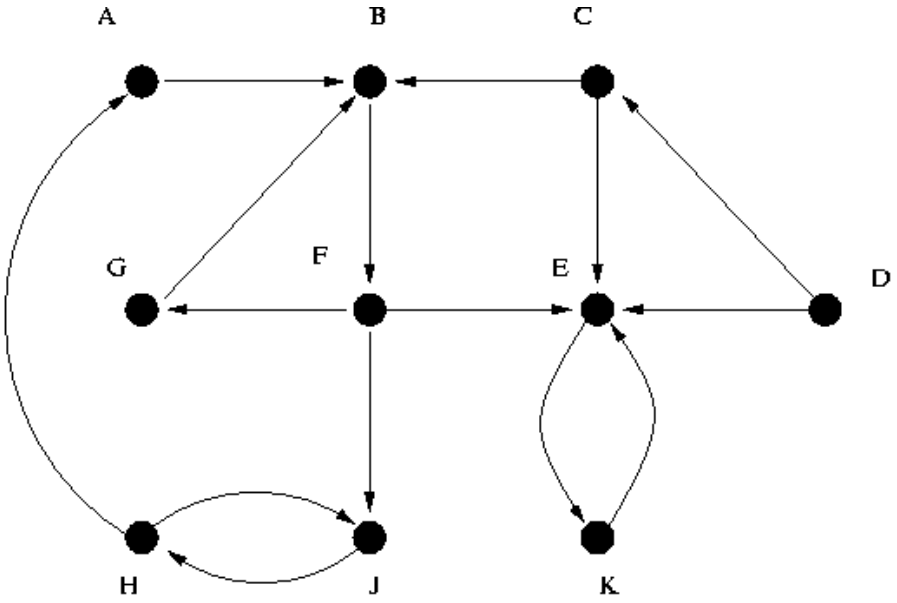
fuertemente conexos en un grafo sirve como ilustración de un algoritmo extremadamente elegante en cuanto a su disenio y nos permite observar que el manejo de grafos puede ser realmente delicado.

Con esto ha sido suficiente por ahora, pero hay muchos problemas con grafos que aun no hemos estudiado y que vamos a analizar en los temas subsiguientes, por eso es recomendable mantener estos temas presentes.

7.6 Ejercicios

1. Dar un algoritmo de orden lineal o mejor que dado un grafo conexo G devuelva un vértice de forma tal que la eliminación de dicho vértic no desconecte el grafo.

2. Dado el siguiente grafo.



   (a) Indicar todas las propiedades que pueda sobre el grafo.

   (b) Realizar una búsqueda BFS sobre el grafo a partir del nodo A.

   (c) Realizar una búsqueda DFS sobre el grafo a partir del nodo A. Dar el ordenamiento topológico del grafo, indicar si el grafo tiene ciclos.

   (d) Indicar cuales son los componentes fuertemente conexos de un grafo.

3. Dar un agoritmo de orden lineal que determine si un grafo es bipartito.

Chapter 8

Programación dinámica

8.1 Introducción a la programación dinámica

La programación dinámica es una técnica muy valiosa para el disenio de algoritmos, se aplica a problemas de optimización, es decir aquellos problemas en los cuales de un conjunto de soluciones posibles (que en general es un número grande) se debe elegir aquella que maximiza o minimiza un cierto parámetro.

Metodo de programacion dinamica

Programacion dinamica metodo

El desarrollo de un algoritmo de programación dinámica comprende los siguientes pasos.

1. Caracterizar la estructura de una solución óptima.
2. Recursivamente definir el valor de una solución óptima.
3. Calcular el valor de la solución óptima en forma bottom-up.
4. Construir la solución óptima a partir de la información calculada.

Los primeros tres pasos forman la base de una solución de programación dinámica a un problema. El paso 4 puede ser omitido si solo se desea el valor de la solución óptima, si se desea además saber cuál es la solución óptima el paso 4 es obligatorio.

8.1.1 Características de un problema de programación dinámica

Es interesante saber a que tipo de problemas podemos aplicarles el paradigma de programación dinámica al diseniar un algoritmo, no hay una respuesta exacta a esta pregunta pero si podemos dar algunas características que son comunes a

los problemas que se pueden resolver utilizando programación dinámica, dichas características son:

1. Subestructura-óptima
2. Superposición de subproblemas.

Subestructura óptima

Decimos que un problema exhibe subestructura óptima si una solución óptima al problema en cuestión contiene soluciones óptimas a los subproblemas involucrados.

Superposición de subproblemas

La segunda característica a observar en un problema para que el método de programación dinámica sea aplicable es que la solución de los distintos subproblemas implica la solución de sub-sub-problemas de forma tal que un sub-sub-problema es requerido para solucionar mas de un subproblema. Almacenando el resultado de los sub-sub-problemas en una tabla es posible evitar recalcularlos cada vez.

A continuación veremos algunos problemas clásicos que se resuelven por programación dinámica.

Problema de la multiplicacion de matrices

Multiplicacion en cadena de matrices

8.2 Multiplicación en cadena de matrices

Dada una secuencia de n matrices $\langle A_1, A_2, \ldots, A_n \rangle$ donde la cantidad de filas de A_i es A_{i-1} y la cantidad de columnas es $A_i = A_{j-1}$ (la cantidad de columnas de A_i debe ser igual a la cantidad de filas de A_{i+1} para poder multiplicarlas. Queremos calcular el producto.

$$A_1 \cdot A_2 \ldots \cdot A_n$$

De forma tal de utilizar la menor cantidad de multiplicaciones posibles, como el producto de matrices es asociativo resulta que hay varias formas de multiplicar la cadena de matrices. Por ejemplo cuatro matrices pueden multiplicarse de las siguientes formas.

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

La forma en la cual asociamos la cadena de matrices puede tener un impacto dramático en la cantidad de multiplicaciones que tenemos que hacer, por ejemplo con tres matrices $\langle A_1, A_2, A_3 \rangle$ de dimensiones $10x100$, $100x5y5x50$. Si hacemos $((A_1 A_2)A_3)$ tenemos que realizar $10 \cdot 100 \cdot 5 = 5000 + 10 \cdot 5 \cdot 50 = 2500 = 7500$ multiplicaciones en total. Si en cambio hacemos $(A_1(A_2 A_3))$ hacemos $100 \cdot 5 \cdot 50 = 25000 + 10 \cdot 100 \cdot 50 = 50000 = 75000$ multiplicaciones. Por lo tanto la primera asociacion es diez veces mas eficiente que la segunda.

Notemos que para multiplicar $A \cdot B$ insumimos

$$Filas_A \cdot Columnas_A \cdot Columnas_B$$

multiplicaciones. El algoritmo que multiplica dos matrices es el siguiente.

Algorithm 37 Matrix-Multiply(A,B). Multiplica las matrices A y B

```
if columnas[A] ≠ filas[B] then
  error
else
  for i = 1 to filas[A] do
    for j = 1 to columnas[B] do
      C[i, j] ← 0
      for k = 1 to columnas[A] do
        C[i, j] ← C[i, j] + A[i, k] · B[k, j]
      end for
    end for
  end for
end if
return C
```

8.2.1 Aproximación por fuerza bruta

Antes de aplicar la técnica de programación dinámica es conveniente verificar si la solución por fuerza bruta es ineficiente. En este caso la solución por fuerza bruta consiste en un algoritmo iterativo que pruebe todas las formas posibles de asociar las matrices calculando cuantos productos son necesarios en cada caso. El algoritmo de esta forma luego de verificar todas las asociaciones puede decidir cual es la forma óptima de asociar las matrices para multiplicarlas.

Para saber si este algoritmo es eficiente necesitamos saber cuantas asociaciones posibles existen dado un conjunto de n matrices.

Cuando solo hay dos matrices hay una sola forma de asociarlas. Cuando tenemos n matrices hay $n-1$ lugares en donde podemos poner el par de parénteisis mas externos. Por ejemplo con seis matrices tenemos:

$$(A_1 A_2)$$

$$(A_1 A_2 A_3)$$

$$(A_1 A_2 A_3 A_4)$$

$$(A_1 A_2 A_3 A_4 A_5)$$

Cuando el paréntesis mas externo es puesto luego de la $k$esima matriz creamos dos sublistas a ser asociadas. Una con k matrices y otra con $n-k$ matrices. Entonces debemos resolver como asociar estas sublistas. Como ambas sublistas son independientes si hay L formas de asociar la sublista izquierda y R formas de asociar la lista derecha el total de asociaciones posibles es $L \cdot R$. Esto sugiere la siguiente recurrencia donde P(n) es la cantidad de formas de asociar n matrices.

$$P(n) = \left\{ \begin{array}{ll} 1 & si\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & si\ n \geq 2 \end{array} \right\}$$

Esta es una función muy conocida en cálculo combinatorio porque define los **numeros Catalanes** (cantidad de diferentes árboles binarios con n nodos). En particular en nuestro caso $P(n) = C(n-1)$.

$$C(n) = \frac{1}{n+1} \left(\begin{array}{c} 2n \\ n \end{array} \right)$$

Aplicando la aproximación de Stirling al cálculo del factorial podemos saber que $C(n) \in \Omega(4^n/n^{3/2})$. Como 4^n es exponencial el crecimiento de $P(n)$ es exponencial.

Dado que el crecimiento de $P(n)$ es exponencial la solución por fuerza bruta no es aplicable a menos que la cantidad de matrices sea extremadamente chica.

Resulta interesante observar como para n matrices cada forma de asociarlas define un árbol binario que indica el orden en el cual deben realizarse los productos, de alli que la cantidad de asociaciones sea la cantidad de árboles binarios posibles para $n-1$ nodos.

8.2.2 Solución por programación dinámica

Veamos como se aplican los cuatro pasos básicos de la técnica de programación dinámica a nuestro caso.

Estructura de una solución óptima

El primer paso al aplicar programación dinámica es caracterizar la estructura de una solución óptima. Para el problema del producto de la cadena de matrices podemos hacerlo de la siguiente manera.

Sea $A_{i..j}$ el resultado de multiplicar $A_i \cdot A_{i+1} \cdot \ldots A_j$. Una asociación óptima del producto de n matrices necesariamente implica dividir la cadena en dos subcadenas, asociar las sub-cadenas y luego multiplicarlas.

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n = A_k \cdot A_{k+1}$$

El costo de esta asociación óptima es el costo de A_k mas el costo de A_{k+1} mas el costo de multiplicar ambas matrices resultantes. La clave consiste en observar que se cumple el principio de **subestructura óptima** ya que si la asociación $A_k \cdot A_{k+1}$ es óptima entonces la asociación de A_k también debe serlo de lo contrario la solución no sería óptima. Analogamente la asociación de A_{k+1} tambien debe ser óptima.

Definición recursiva de una solución óptima

El segundo paso de un problema de programación dinámica es definir el valor de una solución óptima en forma recursiva.

Sea $m[i,j]$ la cantidad de multiplicaciones óptimas para realizar el producto $A_{i..j}$, para n matrices la cantidad de multiplicaciones mínimas es $m[1,n]$. Podemos definir $m[i,j]$ recursivamente de la siguiente manera. Si $i = j$ entonces la cadena tiene solo una matriz y no hace falta ninguna multiplicación por lo tanto $m[i,i] = 0$. Para calcular $m[i,j]$ supongamos que la asociación óptima de las n matrices es $A_k \cdot A_{k+1}$ entonces.

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

Donde p_{i-1} es la cantidad de filas de A_i, p_k es la cantidad de columnas de A_i y la cantidad de filas de A_j, p_j es la cantidad de columnas de A_j.

La ecuación recursiva que definimos asume que conocemos el valor de k, lo cual no es cierto. Sin embargo solo hay $j-1$ valores posibles de k, como la asociación óptima necesariamente utiliza alguno de estos valores podemos chequearlos todos para encontrar el mejor. Por lo tanto nuestra definición recursiva para la asociación óptima de n matrices es.

$$m[i,j] = \left\{ \begin{array}{ll} 0 & si\ i = j \\ min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & si\ i < j \end{array} \right\}$$

Los $m[i,j]$ nos dan los valores de las soluciones óptimas a los subproblemas.

Calculo de la solución óptima en forma botton-up

El tercer paso para diseniar un algoritmo de programación dinámica es poder diseniar una estrategia que construya la solución óptima de un problema a partir de las soluciones de los sub-problemas involucrados. Veamos como podemos hacer esto para el producto de la cadena de matrices.

Supongamos que tenemos 5 matrices: $\langle A_1, A_2, A_3, A_4, A_5 \rangle$.Los siguientes subproblemas ya los tenemos solucionados.

$$m[1,2] = A_{1..2}$$

$$m[2,3] = A_{2..3}$$

$$m[3,4] = A_{3..4}$$

$$m[4,5] = A_{4..5}$$

Ya que solamente implica multiplicar dos matrices. Luego podemos pasar a los subproblemas con 3 (tres) matrices. Para $m[1,3]$ tenemos que calcular el mínimo entre $((A_1 A_2)A_3)$ y $(A_1(A_2 A_3))$. Esto implica calcular lo siguiente:

$$m[1,3] = min\{m[1,2] + p_0 p_2 p_3; m[2,3] + p_0 p_1 p_3\}$$

Analogamente podemos calcular $m[2,4], m[3,5]$.

Pasamos ahora a los subproblemas de longitud 4, por ejemplo para calcular $m[1,4]$ k puede valer $1,2,3$ por lo tanto tenemos que hallar el mínimo entre:

$$m[1,4] = min \begin{pmatrix} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{pmatrix}$$

Notemos que $m[2,4], m[1,2], m[1,3]$ son subproblemas que ya resolvimos por lo que no necesitamos resolverlos de nuevo, simplemente si $m[i,j]$ es una tabla accedemos al valor correspondiente. Aplicamos la misma forma para $m[2..5]$ que es el otro subproblema de longitud 4.

Por último podemos resolver el problemad de longitud 5 que es el problema completo,para ello aplicamos la misma técnica que antes.

$$m[1,5] = min \begin{pmatrix} m[1,1] + m[2,5] + p_0 p_1 p_5 \\ m[1,2] + m[3,5] + p_0 p_2 p_5 \\ m[1,3] + m[4,5] + p_0 p_3 p_5 \\ m[1,4] + m[5,5] + p_0 p_4 p_5 \end{pmatrix}$$

De esta forma podemos observar como resolviendo los sub-problemas en el orden apropiado podemos ir llenando una tabla de forma tal que cada sub-problema sea solucionado utilizando la solución a los subproblemas anteriores. El orden en el cual debemos llenar la tabla es de acuerdo al tamanio de los subproblemas en nuestro caso fue.

$$m[1,1] \tag{8.1}$$

$$m[2,2] \tag{8.2}$$

$$m[3,3] \tag{8.3}$$

$$m[4,4] \tag{8.4}$$
$$m[5,5] \tag{8.5}$$
$$m[1,3] \tag{8.6}$$
$$m[2,4] \tag{8.7}$$
$$m[3,5] \tag{8.8}$$
$$m[1,4] \tag{8.9}$$
$$m[2,5] \tag{8.10}$$
$$m[1,5] \tag{8.11}$$

Conociendo la definición recursiva necesaria para calcular la solución óptima y el orden en que debemos resolver los subproblemas podemos escribir el siguiente algoritmo de programación dinámica.

Algorithm 38 Matrix-Chain-Order(n). Determina la asociación óptima de n matrices.

```
for i = 1 to n do
  m[i, i] ← 0
end for
for l = 2 to n do
  for i = 1 to n − l + 1 do
    j ← i + l − 1
    m[i, j] ← ∞
    for k = 1 to j − 1 do
      q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
      if q < m[i, j] then
        m[i, j] ← q
      end if
    end for
  end for
end for
```

Si realizamos un seguimiento del algoritmo podemos observar como se llena la tabla en el orden necesario para resolver el problema.

Calculo de la solución óptima

Hasta aqui el algoritmo permite calcular cual es el número de multiplicaciones mínimo para n matrices, sin embargo tambien necesitamos saber cual es la asociación que nos permite realizar el producto en dicho orden. Para ello basta con utilizar una tabla auxiliar que llamaremos $s[i,j]$ en donde si $s[i,j] = k$ quiere decir que para $A_{i..j}$ la asociación óptima es $A_k \cdot A_{k+1}$. De esta forma el algoritmo que calcula la cantidad de multiplicaciones puede calcular la tabla s con un sencillo cambio. Para calcular la asociación óptima tenemos que acceder a s en forma recursiva, el algroritmo final es el siguiente.

Algorithm 39 Matrix-Chain-Order(n). Determina la asociación óptima de n matrices.

```
for i = 1 to n do
    m[i, i] ← 0
end for
for l = 2 to n do
    for i = 1 to n − l + 1 do
        j ← i + l − 1
        m[i, j] ← ∞
        for k = 1 to j − 1 do
            q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j] then
                m[i, j] ← q
                s[i, j] ← k
            end if
        end for
    end for
end for
```

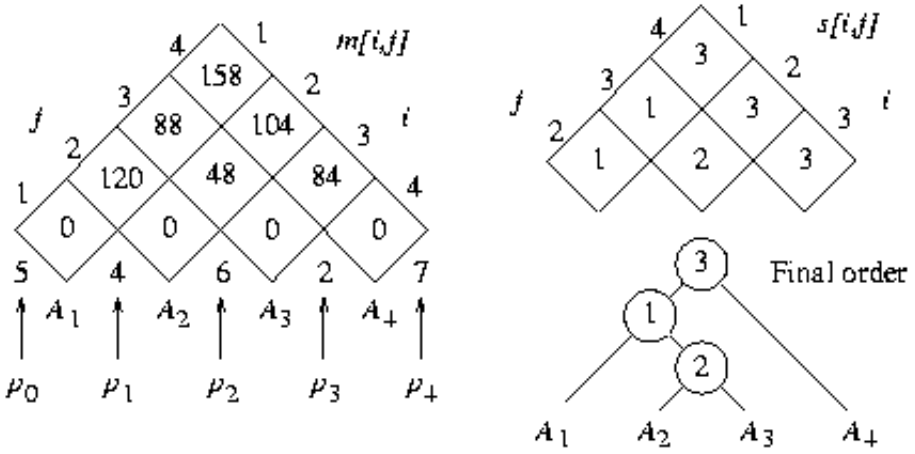
Algorithm 40 Matrix-Chain-Multiply(A,s,i,j).Usando s multiplica la cadena A_{i--j}

```
if j > i then
    x ← Matrix − Chain − Multiply(A, s, i, s[i, j])
    y ← Matriz − Chain − Multiply(A, s, s[i, j] + 1, j)
    returnMatrix − Multiply(X, Y)
else
    return A_i
end if
```

Ejemplo Por ejemplo supongamos que tenemos cuatro matrices.

$$\begin{array}{ll} A_1 & 5x4 \\ A_2 & 4x6 \\ A_3 & 6x2 \\ A_4 & 2x7 \end{array}$$

Las tablas se completan de la siguiente forma:



El algoritmo es un tanto rebuscado por lo que resultaría conveniente realizar un seguimiento hasta asgurarse de comprender su funcionamiento, la solución final del ejemplo es $((A_1(A_2A_3))A_4)$

8.3 El problema de todas las distancias mínimas en un grafo

Vamos a retornar a los algoritmos relacionados con la manipulación de grafos. Cuando estudiamos el algoritmo BFS vimos que podía ser utilizado para calcular la distancia mínima desde un vértice hasta cualquier otro en un grafo en el cual las aristas no tenian pesos. En esta sección vamos a realizar dos ampliaciones importantes al problema, en primer lugar vamos a calcular las distancias mínimas desde cualquier vértice hasta cualquier otro. En segundo lugar vamos a admitir que las aristas del grafo tienen pesos que en general representan distancias aunque podrían tener cualquier otro significado (por ejemplo costo del peaje etc).

8.3.1 Definición del problema

Sea $G = (E, v)$ un grafo dirigido con aristas pesadas. Si $(u, V) \in E$ es una arista de G entonces el peso de la arista es $W(u, v)$. Supongamos por el momento que las aristas siempre tienen pesos positivos, un peso positivo excesivamente grande $W(u, v)$ (equivalente a ∞) indica que no existe camino entre ambos vértices. Dado un camino $\pi = \langle u0, u1, \ldots uk \rangle$ el costo del camino es la suma de los pesos de las aristas.

$$costo(\pi) = W(u0, u1) + W(u1, u2) + \ldots + W(uk-1, uk) = \sum_{i=1}^{k} W(ui-1, ui)$$

El problema consiste en encontrar **todos** los caminos de costo mínimo desde un vértice hasta cualquier otro del grafo. Para este problema vamos a suponer que el grafo se encuentra representado por una matriz de adyacencias en lugar de la mas común en general lista de adyacencias [1].

Los datos de entrada estan almacenados en una matriz de $nx\ W$ que contiene los pesos de las aristas. Sea w_{ij} el elemento $W[i, j]$.

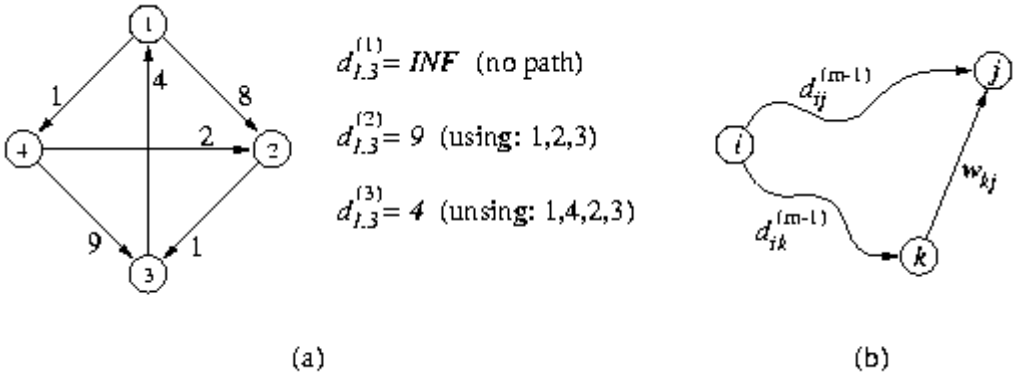
$$w_{ij} = \left\{ \begin{array}{ll} 0 & si\, i = j \\ W(i,j) & si\, i \neq j\, and\, (i,j) \in E \\ +\infty & si\, i \neq j\, and\, (i,j) not \in E \end{array} \right\}$$

El resultado del algoritmo debe ser una matriz de $nx\ D = d_{ij}$ donde $d_{ij} = \delta(i, j)$ el costo del camino mas corto desde i hasta j. Conocer cual es el camino mas corto implica el uso de una matriz auxiliar $pred[i, j]$. El valor de $pred[i, j]$ es un vértice que pertenece al camino mínimo entre i y j, es decir que $pred[i, j] = k$ implica que para ir desde i hasta j con costo mínimo hay que pasar por k, luego para seguir construyendo el camino hay que observar $pred[i, k]$ y $pred[k, j]$.

8.3.2 Solución por programación dinámica

Por el momento nos vamos a concentrar en calcular el costo de los caminos mínimos en lugar del camino en si. En primer lugar necesitamos descomponer el problema en subproblemas. Podemos por ejemplo adoptar el siguiente enfoque: para $0 \leq m \leq n-1$, definimos $D_{ij}^{(m)}$ como el costo del camino mínimo desde i hasta j usando a lo sumo m aristas. La idea es calcular $D^{(0)}, D^{(1)}, \ldots D^{(n-1)}$. Ya que ningun camino puede usarmas de $n-1$ aristas (en caso contrario deberia repetir un vértice), sabemos que $D^{(n-1)}$ es la matriz final. De esta forma hemos caracterizado la estructura de la solución óptima. Esto se ilustra en la siguiente figura.

[1] Para grafos dispersos la lista de adyacencias suele ser mas eficiente, pero en estos problemas es raro encontrar un grafo disperso ya que en general desde un vértice hay caminos hacia casi todos los demás vértices



El siguiente paso consiste en definir la solución en forma recursiva. Para computar las matrices de distancias el caso basico es $D^{(1)} = W$ dado que las aristas del digrafo son caminos de longitud 1.

$$D_{ij}^{(1)} = W_{ij}$$

Ahora queremos ver que es posible calcular $D^{(m)}$ a partir de $D^{(m-1)}$ para $m \geq 2$. Consideremos que $D_{ij}^{(m)}$ es la distancia mínima desde i hasta j usando a lo sumo m aristas. Hay dos casos.

Caso 1: Si el camino mínimo usa estritamente menos de m aristas entonces su costo es $D_{ij}^{(m-1)}$.

Caso 2: Si el camino mínimo usa exactamente m aristas entonces el camino usa $m-1$ aristas para ir desde i hasta algun vértice k y luego usa la arista (k, j) de peso w_{kj} para llegar hasta j. El camino desde i hasta k debe ser también óptimo (subestructura óptima), por lo que la longitud del camino resultante es $D_{ik}^{(m.1)} + W_{kj}$.

Como no sebmos el valor de k debemos hallar el mínimo probando con todos los valores posibles de k.

$$D_{ij}^{(m)} = min \left\{ \begin{array}{c} D_{ij}^{(m-1)} \\ min_{1 \leq k \leq n}(D_{ik}^{(m-1)} + w_{kj}) \end{array} \right\}$$

8.3.3 Primera solución

A continuación mostramos un algoritmo tentativo que utiliza esta regla recursiva.

Algorithm 41 Dist1(m,i,j). Calcula la distancia mínima desde i hasta j

if $m = 1$ **then**
  return $W[i, j]$
end if
$best \leftarrow \infty$
for $k = 1$ to n **do**
  $best \leftarrow min(best; Dist1(m-1, i, k) + W[k, j])$
end for
return best

Desafortunadamente el algoritmo es realmente lento, sea $T(m, n)$ el tiempo del algoritmo en un grafo de n vértices. El algoritmo hace n llamadas a si mismo con el primer parámetro de $m-1$, cuando $m = 1$ tenemos T(1,n)=1. Sino hacemos n llamadas recursivas a $T(m-1, n)$. Es decir:

$$T(m, n) = \left\{ \begin{array}{ll} 1 & si\, m = 1 \\ nT(m-1, n) + 1 & sino \end{array} \right\}$$

El tiempo total es $T(n-1, n)$, esto se resuelve en forma directa usando el método iterativo obteniendo como resultado $T(n) = O(n^n)$. Y lamentablemente esta función es una de las funciones de crecimiento mas rápido que conocemos, es super-exponencial y crece incluso mas rapido que $n!$. Esto ocurre porque el algoritmo simplemente intenta con todos los caminos posibles desde i hasta j y la cantidad de caminos posibles entre dos vértices de un grafo es exponencial.

Vemos este algoritmo para ilustrar la ventaja de la técnica de programación dinámica sobre los métodos simples o de fuerza bruta, el único paso que nos faltaba para obtener un algoritmo de programación dinámica era determinar como calcular la solución en forma bottom-up reutilizando los subproblemas ya resueltos.

El siguiente algoritmo implementa este paso, el procedimiento principal $ShortestPath(n, w)$ recibe la cantidad de vértices n y la matriz de pesos W. La matriz $D^{(m)}$ es almacenada en la tabla $D[m]$. Para cada m, $D[m]$ es una matriz de dos dimensiones, por lo que D es una matriz tridimensional. Inicializamos $D^{(1)}$ copiando W. Luego cada llamada a $ExtendPaths$ calcula $D^{(m)}$ a partir de $D^{(m-1)}$ usando la fórmula que habiamos mostrado antes.

El procedimiento $ExtendPaths$ consiste de tres ciclos anidados por lo que su orden es $\Theta(n^3)$. Es llamado $n-1$ veces desde el procedimiento principal y por lo tanto el tiempo total es $\Theta(n^4)$.

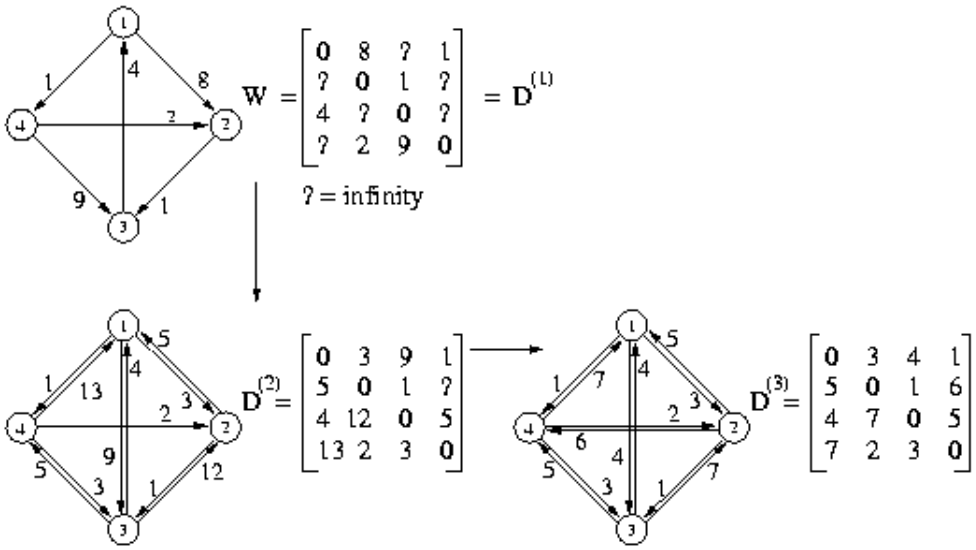
Algorithm 42 ShortestPath(n,W). Calcula todos los caminos mínimos.

```
array D[1..n-1][1..n][1..n]
copiar W en D[1]
for m = 2 to n − 1 do
   D[m] ← ExtendPaths(n, D[m − 1], W)
end for
return D[n-1]
```

Algorithm 43 ExtendPaths(n,d,W).

```
matrix dd[1..n][1..n]=d[1..n][1..n] // copiamos d en dd
for i = 1 to n do
   for j = 1 to n do
      for k = 1 to n do
         dd[i, j] ← min(dd[i, j], d[i, k] + W[k, j])
      end for
   end for
end for
return dd
```

El funcionamiento del algoritmo puede verse en la ilustración.



8.3.4 Algoritmo de Floyd-Warshall

El algoritmo de Floyd-Warshall surge en la decada del '60 y permite calcular todos los caminos minimos en $\Theta(n^3)$ lo cual mejora el orden de nuestro algoritmo anterior que era $\Theta(n^4)$. El algoritmo se basa en lo siguiente:

Estructura de la solución óptima El camino mínimo desde i hasta j puede ser o bien $W[i,j]$ o bien $d[i,k] + d[k,j]$ implicando que para ir desde i hasta j hay que pasar por j. A su vez d[i,k] y d[k,j] también deben ser óptimos.

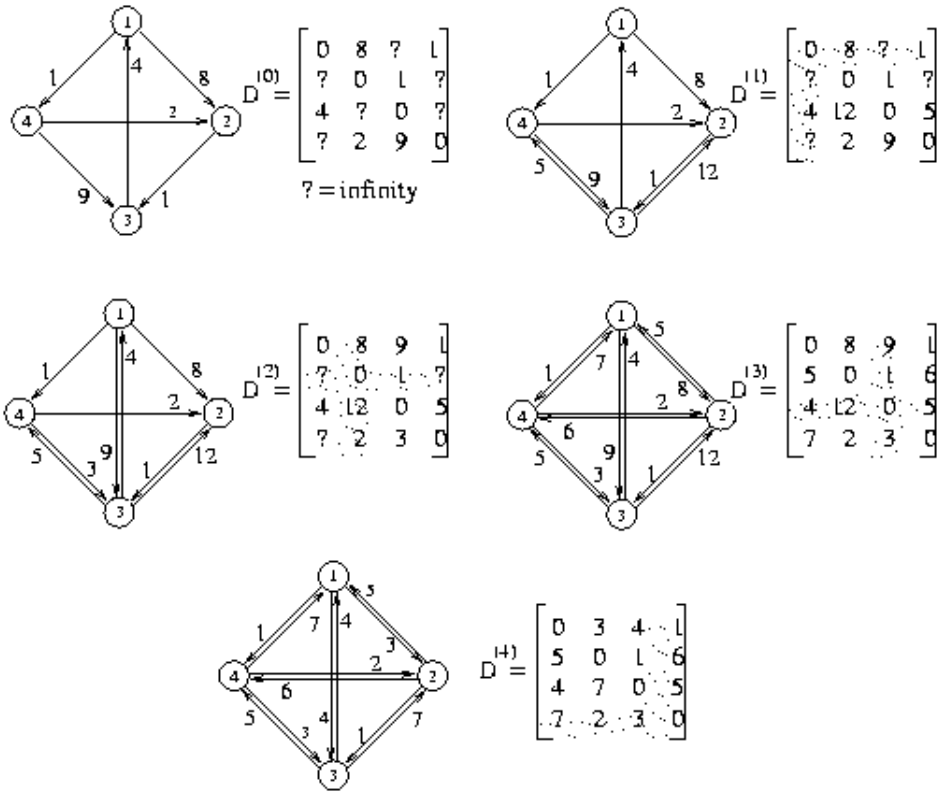
Algorithm 44 Floyd-Warshall(n,W)

```
array d[1..n][1..n]
for i = 1 to n do
  for j = 1 to n do
    d[i, j] ← W[i, j]
    pred[i, j] ← null
  end for
end for
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      if (d[i, k] + d[k, j] < d[i, j]) then
        d[i, j] ← d[i, k] + d[k, j]
        pred[i, j] = k
      end if
    end for
  end for
end for
return d
```

El algoritmo

Análisis

Claramente el orden del algoritmo es $\Theta(n^3)$, la clave esta en el orden en el cual se construye la solución, en primer lugar se recorre toda la matriz para el vértice intermedio 1, luego para el 2, etc. A continuación mostramos el algoritmo en un ejemplo.



Para calcular cuál es el camino mínimo hay que usar la matriz $pred[i,j]$ que también calcula el algoritmo, en forma recursiva se puede calcular cual es el camino mínimo de la siguiente manera.

Algorithm 45 Camino(i,j)

```
if pred[i, j] = null then
   output(i,j)
else
   Path(i,pred[i,j])
   Path(pred[i,j],j)
end if
```

8.4 El problema de la subsecuencia común de máxima longitud

Vamos a incursionar ahora en un algoritmo para manipulación de strings, uno de los problemas básicos es por ejemplo encontrar un string dentro de otro, este tipo de algoritmo debe ser utilizado por programas como el GREP y otros. En algunos casos sin embargo no interesa comparar dos strings en forma exacta sino en forma aproximada, esto tiene muhcas aplicaciones incluyendo varios aspectos importantes de la genética.

Subsecuencias Dadas dos secuencias $X = \langle X1, X2, \ldots Xn\rangle$ y $Z = \langle Z1, Z2, \ldots Zk\rangle$ decimos que Z es una subsecuencia de X si existe un orden estrictamente creciente de índices $\langle i1, i2, ..ik\rangle$ $(1 \leq i1 \leq i2 < \ldots < ik \leq n)$ tales que $Z = \langle Xi1, Xi2, \ldots, Xik\rangle$. Por ejemplo $Z = \langle AADAA\rangle$ es una subsecuencia de $X = \langle ABRACADABRA\rangle$. Notemos que el hecho de que existan elementos de X en el medio que no estan en Z no afecta el concepto de subsecuencia.

Subsecuencia común de longitud máxima Dados dos strings X y Y la **subsecuencia común de longitud máxima (SCM)** de X y Y es la subsecuencia de longitud máxima Z que es a su vez subsecuencia de X y Y.

Por ejemplo si $X = \langle ABRACADABRA\rangle$ y $Y = \langle YABBADABBADOO\rangle$. La subsecuencia común de máxima longitud es $Z = \langle ABADABA\rangle$

El problema de la SCM El problema (SCM) es el siguiente: dados dos strings X y Y determinar cual es la subsecuencia común de longitud máxima, si la misma no fuese única basta con calcular una cualquiera.

8.4.1 Solución por programación dinámica

La aproximación por fuerza brute al problema consistiria en computar todas las subsecuencias de un string y luego verificar si las mismas son subsecuencias del segundo string, de todas las subsecuencias comunes luego se elige la de mayor longitud. Sin embargo esto es claramente ineficiente ya que en un string de n caracteres existen 2^n posibles subsecuencias por lo que el algoritmo simple sería de caracter exponencial (o super-exponencial) lo cual es claramente ineficiente.

Para aplicar el método de programación dinámica necesitamos dividir el problema de la (SCM) en sub-problemas, esto lo vamos a hacer definiendo al prefijo X_i de una secuencia X como $\langle X0, X1, \ldots Xi\rangle$ .La idea es computar la (SCM) por cada posible par de prefijos. Sea $c[i, j]$ la longitud de la subsecuencia de longitud máxima de X_i y Y_j . Eventualmente el resultado final sera $c[m, n]$. Como veremos $c[i, j]$ puede calcularse si conocemos $c[i', j']$ para $i' \leq i$, $j' \leq j$ (pero no ambos iguales.Empecemos con algunas observaciones.

Caso básico $c[i,0] = c[j,0] = 0$. Si alguna de las subsecuencias es vacía entonces no hay (SCM).

Si los últimos caracteres son iguales Sea $x_i = y_j$. Por ejemplo $X_i = \langle ABCA \rangle$ y $Y_j = \langle DACA \rangle$. Como ambos terminan en "A" afirmamos que la (SCM) también deber terminar en A, ya que sino agregando A al final tendríamos una subsecuencia común de mayor longitud. Una vez que sabemos que A es parte de la (SCM) entonces podemos hallar la LCS de $X_{i-1} = \langle ABC \rangle$ y $Y_{j-1} = \langle DAC \rangle$. Entonces. Si $x_i = y_j$ entonces $C[i,j] = c[i-1, j-1] + 1$

Si los últimos caracteres son distintos Supongamos que $x_i \neq y_j$. En este caso o bien x_i no es parte dela SCM o bien y_j no es parte de la SCM o bien ninguno es parte de la SCM. En el primer caso la SCM de X_i e Y_j es la SCM de X_{i-1} e Y_j. En el segundo caso la SCM es la SCM de X_i y Y_{j-1}. Entonces. Si $x_i \neq y_j$ $c[i,j] = max(c[i-1,j], c[i,j-1])$. El tercer caso queda cubierto por un primer caso seguido de un segundo caso. La regla recursiva para calcular la solución óptima es entonces la siguiente:

$$c[i,j] = \begin{cases} 0 & si\ i = 0\ o\ j = 0 \\ c[i-1, j-1] + 1 & si\ i,j > 9\ y\ x_i = y_i \\ max(c[i, j-1]; c[i-1, j]) & si\ i,j > 0\ y\ x_i \neq y_j \end{cases}$$

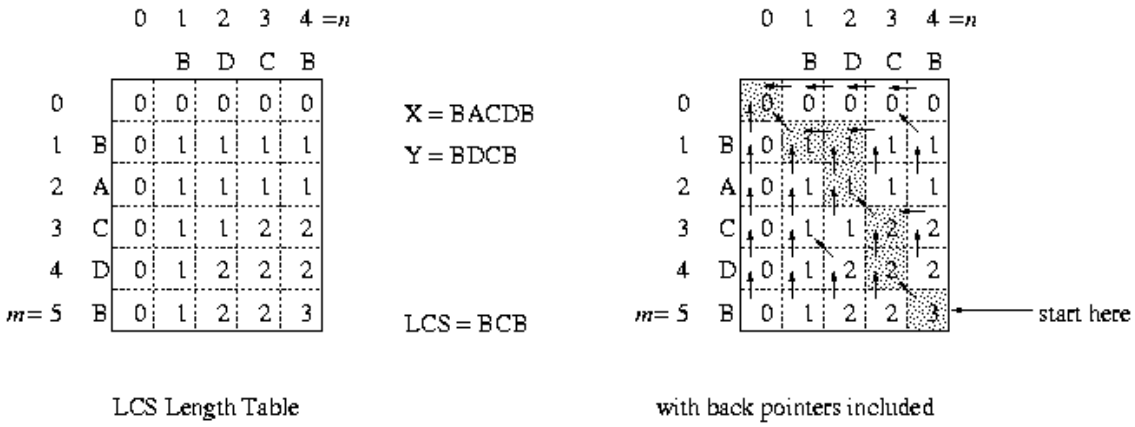
La tarea a continuación es implementar la fórmula recursiva en forma bottom-up reutilizando los subproblemas resueltos.

8.4.2 Análisis

El análisis es muy sencillo ya que el algoritmo consiste de dos ciclos de n y m iteraciones anidados, el orden es por lo tanto $O(mn)$. El funcionamiento del algoritmo puede resultar un tanto misterioso por lo que se recomienda realizar un seguimiento para observar como funciona. La ilustración es un ejemplo de como funciona el algoritmo.

Algorithm 46 LCS(char x[1..m],char y[1..n])

```
int c[0..m][0..n]
for i = 0 to m do
    c[i, 0] = 0
    b[i, 0] = SKIPX
end for
for j = 0 to n do
    c[0, j] = 0
    b[0, j] = SKIPY
end for
for i = 1 to m do
    for j = 1 to n do
        if x[i] = y[j] then
            c[i, j] ← c[i − 1, j − 1] + 1
            b[i, j] = ADDXY
        else
            if c[i − 1, j] ≥ c[i, j − 1] then
                c[i, j] ← c[i − 1, j]
                b[i, j] ← SKIPX
            else
                c[i, j] ← c[i, j − 1]
                b[i, j] ← SKIPY
            end if
        end if
    end for
end for
return c[m,n]
```



El cálculo de cual es la subsecuencia común de longitud máxima es simple utilizando los punteros almacenados en la matriz auxiliar $b[i, j]$

Algorithm 47 getSCM(char c[1..m],char y[1..n], int b[0..m,0..n])

```
SCM ← empty
i ← m
j ← n
while i ≠ 0 and j ≠ 0 do
  if b[i, j] = ADDXY then
    add x[i] to front of SCM
    i ← i − 1
    j ← j − 1
  end if
  if b[i, j] = SKIPX then
    i ← i − 1
  end if
  if b[i, j] = SKIPY then
    j ← j − 1
  end if
end while
return SCM
```

8.5 Resumen y puntos clave

La programación dinámica es una técnica que permite diseniar algoritmos eficientes (al menos mas eficientes que los de fuerza bruta) a problemas de op-

timización. Para que un problema sea solucionable mediante programación dinámica debe presentar dos características:

- Subsestructura óptima El problema debe poder dividirse en sub-problemas de forma tal que si la solución al problema es óptima la solución de los subproblemas que lo componen tambien debe serlo.
- Superposición de subproblemas Un problema debe poder resolverse a partir de la resolución de sus sub-problemas y asi sucesivamente en forma recursiva. Ademas un sub-problema debe formar parte de la solución de mas de un problema de mayor orden de forma tal que si se almacena el reusltado del sub-problema en una tabla el mismo pueda ser usado para evitar resolver dos veces el mismo problema.

La técnica de programación dinámica consiste en los siguientes pasos:

- Definir la estructura de la solución
- Hallar una fórmula recursiva para calcular la solución
- Implementar la solución recursiva en forma botton-up reutilizando los subproblemas ya resueltos.
- Encontrar la solución óptima

Como ejemplos de programación dinámica vimos cuatro algoritmos importantes, el primero de ellos sirve para multiplicar una cadena de matrices en forma óptima, el segundo y el tercero resuelven el problema de las distancias mínimas en un grafo, el cuarto resuelve un problema con strings que tiene muchas aplicaciones en la actualidad.

Comprender y dominar el método de programación dinámica requiere tiempo y dedicación pero permite encontrar soluciones eficientes donde de otra forma el costo para resolver el problema sería demasiado grande.

8.6 Ejercicios

1. Una empresa se encuentra organizando una fiesta para sus empleados, la empresa está organizada en forma jerárquica con una estructura en forma de árbol. La comisión de fiestas ha asignado a cada empleado un número real que indica el "carisma" de cada uno de sus empleados. Además para que las conversaciones sean mas fluidas y amenas se ha decidido que no pueden asistir a la fiesta un empleado y su superior inmediato juntos. La comisión desea maximizar la sumatoria de carisma total. Dar un algoritmo de programación dinámica que permita determinar quienes deben asistir a la fiesta.

Chapter 9

Algoritmos Golosos

9.1 Algoritmos golosos, principios.

El término "algoritmo goloso" es una traducción libre del inglés "greedy algorithms", en algunos textos se refiere a este tipo de algoritmos como algoritmos "ávidos". Los algoritmos golosos se utilizan para resolver problemas de optimización y son extremadamente sencillos , se basan en un principio fundamental:

Principio: Un algoritmo goloso siempre toma la decisión que parece mejor de acuerdo al estado actual del problema

Un algoritmo goloso funciona construyendo una solución en base a una secuencia de **decisiones golosas**, se dice que la decisión que toman estos algoritmos es golosa porque si bien son óptimas en el estado actual del problema no implica que lleven necesariamente a una solución óptima del problema.

Importante: Una solución localmente óptima no necesariamente lleva a una solución globalmente óptima.

Algunos algoritmos golosos son óptimos, en el sentido de que terminan generando una solución que es óptima para el problema planteado, otros no son óptimos pero sirven para encontrar una buena aproximación de la solución, por último en algunos casos un algoritmo goloso puede llevar a cualquier solución aleatoriamente alejada de la solución óptima.

Importante: Aplicar un algoritmo goloso a un problema en el cual la técnica no resulte propicia suele llevar a una solución que no es mas que una solución cualquiera del conjunto de soluciones posibles.

9.1.1 Fundamentos

Para que un problema pueda ser solucionable en forma óptima mediante un algoritmo goloso debe presentar dos características fundamentales:

- Subestructura óptima El problema debe poder ser descompuesto en varios sub-problemas de forma tal que si la solución del problema es óptima la solución de los subproblemas también lo es. (Este requisito también era necesario para el método de programación dinámica).
- Opción golosa Deber ser posible demostrar que una elección localmente óptima lleva a una solución globalmente óptima. Esta es la clave de los algoritmos golosos y en general se demuestra por inducción, en primer lugar se prueba que si existe una solución óptima entonces existe también una solución que parte de una decisión golosa. Luego para el subproblema restante por inducción se prueba que se puede llegar a una solución óptima.

En esta sección observaremos algunos problemas nuevos y otros no tanto que pueden ser resueltos utilizando un algoritmo goloso.

9.2 El problema de la selección de actividades

El problema de la selección de actividades es el siguiente: dado un un conjunto de n actividades $S = \langle S0, S1, \ldots, Sn \rangle$ y conociendo la hora de inicio y fin de cada actividad S_i, F_i se define que dos actividades son compatibles si no se superponen en el tiempo. El problema consiste en encontrar el conjunto de actividades compatibles de mayor cardinalidad.

9.2.1 Ejemplo

Actividad	Inicio	Fin
S0	3	6
S1	5	7
S2	0	4
S3	1	3
S4	4	5
S5	6	7

En nuestro ejemplo hay varios conjuntos de cantidad de actividades compatibles máxima, por ejemplo $\langle S2, S4, S1 \rangle$ o $\langle S3, S4, S5 \rangle$ o $\langle S3, S4, S1 \rangle$, etc Son soluciones óptimas.

9.2.2 Algoritmo goloso para la selección de actividades

El algoritmo goloso para resolver el problema se basa en lo siguiente:

En primer lugar ordenar las actividades por orden de finalización creciente.

Elegir la primera actividad.
Luego elegir la primera actividad compatible con la última elegida.
Repetir hasta que no queden actividades.

Algorithm 48 Greedy-Selector(S). Recibe un conjunto de S actividades.

Require: S debe estar ordenado por hora de finalizacion $S[i].f \leq S[i+1].f$
  $n \leftarrow length(S)$
  $A \leftarrow \{1\}$
  $j \leftarrow 1$
  for $i = 2$ to n **do**
    if $S[i]s \geq S[i].f$ **then**
      $A \leftarrow A \cup \{i\}$
      $j \leftarrow i$
    end if
  end for

Para nuestro ejemplo el algoritmo elegiría $\langle S3, S4, S1 \rangle$

9.2.3 Análisis

Es evidente que el algoritmo es eficiente, insume el tiempo necesario para el sort mas un tiempo que es $\Theta(n)$ para el proceso de selección. A continuación tenemos que demostrar que el algoritmo goloso para el problema de la selección de actividades es óptimo.

Teorema: El algoritmo $Greedy - Selector$ produce soluciones óptimas.

Demostración: Sea $S = \langle S1, S2, \ldots, Sn \rangle$ el conjunto de actividades a seleccionar ordenadas por tiempo de finalización de forma tal que la actividad 1 es la que finaliza mas temprano. Queremos mostrar que existe al menos una solución óptima que contiene la actividad $S1$.

Supongamos que $A \subseteq S$ es una solución óptima para el problema y ordenemos las actividades de A por hora de finalización. Supongamos ademas que la primera actividad de A es Sk. Si $k = 1$ entonces la solución contiene la actividad elegida por el algoritmo goloso. Si $k \neq 1$ queremos mostrar que existe otra solución óptima B que contiene $S1$. Sea $B = A - \{Sk\} \cup \{1\}$. Como $S1.f \leq Sk.f$ las actividades en B son compatibles, y como B tiene la misma cantidad de actividades que A que era una solución óptima entonces B es tambien una solución óptima. Por lo tanto siempre existe una solución óptima que contiene la decisión golosa.

Además una vez realizada la decisión golosa de la actividad el problema se reduce a encontrar un conjunto óptimo de actividades sobre el conjunto S' siendo S' las actividades de S compatibles con $S1$. Es decir que si A es una solución óptima a S entonces $A' = A - \{S1\}$ es una solución óptima a $S' = \{i \in S : si \geq s1.f\}$. Esto se debe a que si podemos encontrar una solución B' a S con mas actividades que A entonces agregando la actividad $S1$ a B' tendríamos una solución al problema original S con mas actividades que A y por lo tanto A no sería una solución óptima. Por inducción sobre la cantidad de decisiones tomadas por el algoritmo tenemos que la solución final es óptima.

9.3 El problema de los caminos mínimos (parte II)

El problema de los caminos mínimos en un grafo es un problema que ya habiamos estudiado y que consistía en encontrar todos los caminos mínimos en un grafo dirigido donde $W[i,j]$ indicaba el costo de utilizar la arista (i,j) en un camino. El costo de un camino era la sumatoria del costo de todas las aristas del camino. Para este problema habíamos visto dos algoritmos de programación dinámica que servian, uno era $\Theta(n^4)$ y el otro era el algoritmo de Floyd-Warshall que era $\Theta(n^3)$. A continuación trataremos el siguiente problema: dado un vértice v encontrar todos los caminos mínimos desde ese vértice hasta cualquier otro vértice del grafo. El algoritmo que vamos a presentar para este problema es un algoritmo goloso conocido como Algoritmo de Dijkstra.[1]

9.3.1 Algoritmo de Dijkstra para caminos mínimos

La estructura básica del algoritmo de Dijkstra consiste en mantener un **estimado** de la distancia mínima desde el origen s hasta cualquier otro vértice denominada $d[v]$. Este valor va a ser siempre mayor o igual a la distancia mínima desde s hasta v y además vamos a requerir que $d[v]$ siempre sea igual a la distancia de un cierto camino desde s hasta v (excepto cuando $d[v] = \infty$ indicando que no hay camino). Inicialmente $d[s] = 0$ y todos los demás vértices $d[v]$ se setean en ∞. A medida que el algoritmo progresa examina mas y mas vértices intentando actualizar $d[v]$ para cada vértice del grafo hasta que $d[v]$ converge al valor de la distancia mínima.

9.3.2 Mecanismo de relajación

El proceso por el cual se actualiza $d[v]$ se denomina **relajación**. Intuitivamente si uno puede ver que la solución aun no ha alcanzado un valor óptimo se la presiona un poco mas hacia el óptimo, en particular si se descubre un camino de

[1] El algoritmo de Floyd-Warshall de programación dinámica encuentra todos los caminos mínimos desde cualquier vértice hasta cualquier otro en $O(n^3)$, ahora se trata de encontrar todos los caminos mínimos desde solo un vértice hasta todos los demás lo cual debería poder hacerse en menos tiempo. El orden a batir es entonces $O(n^3)$.

s hacia v mas corto que $d[v]$ entonces es necesario actualizar $d[v]$, este principio es común en muchos algoritmos de optimización.

Consideremos una arista desde un vértice u hacia un vértice v cuyo peso es $W[u, v]$. Supongamos que hemos calculado el estimado para $d[u], d[v]$. Sabemos que hay un camino desde s hacia u de costo $d[u]$, usando este camino y agregando la arista (u, v) obtenemos un camino desde s hasta v de distancia $d[u] + W[u, v]$. Si este camino es mejor que el que ya teníamos entonces actualizamos $d[v]$, en caso contrario lo dejamos como está.

Algorithm 49 Relax(u,v). Relaja la arista (u,v)

```
if d[u] + W[u, v] < d[v] then
   d[v] ← d[u] + W[u, v]
   pred[v] ← u
end if
```

No es difícil observar que si realizamos relajaciones sucesivas en todas las aristas del grafo entonces $d[v]$ eventualmente converge la verdadera distancia mínima desde s hasta v. La inteligencia en un algoritmo de este tipo esta en realizar las relajaciones de forma tal que la convergencia sea lo mas rápida posible.

9.3.3 El algoritmo de de Dijkstra

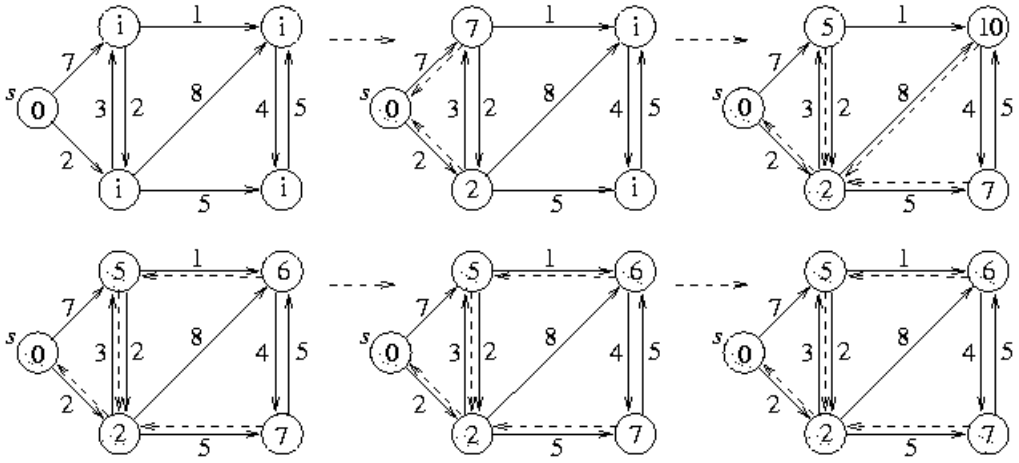
El algoritmo de Dijkstra se basa en la realización de sucesivas relajaciones. El algoritmo opera manteniendo un subconjunto de vértices $S \subseteq V$ para los cuales conocemos la verdadera distancia mínima $d[v]$. Inicialmente $S = Null$ y $d[s] = 0$ y $d[v] = \infty$ para los demas vértices. Uno por uno seleccionamos vértices de $V - S$ para agregarlos a S. El conjunto S puede implementarse como un vector de vértices coloreados, inicialmente todos los vértices son blancos y seteamos $color[v] = negro$ para indicar que $v \in S$.

Al seleccionar que vértice de $V - S$ se agregara a S entra en acción la decisión golosa. Para cada vértice $u \in V - S$ hemos computado la distancia estimada $d[u]$. La decisión golosa consiste en elegir el vértice u para el cual $d[u]$ es mínimo, es decir el vértice sin procesar que de acuerdo a nuestra estimación esta mas cercano a s.

Para hacer esta selección mas eficiente almacenamos los vértices de $V - S$ en una cola de prioridades (Un Heap por ejemplo), donde el valor clave de cada vértice u es $d[u]$. Este es el algoritmo de Dijkstra.

Algorithm 50 Dijsktra(G,w,s). Calcula todos los caminos minimos desde s.

for each u in V **do**
  $d[u] \leftarrow \infty$
  $color[u] \leftarrow blanco$
  $pred[u] \leftarrow null$
end for
$d[s] \leftarrow 0$
$pred[s] \leftarrow null$
$Q \leftarrow construircolacontodoslosvertices$
while Non-empy(Q) **do**
  $u \leftarrow Extract - min(Q)$
  for each v in $Adj[u]$ **do**
    $Relax(u, v)$
    $Decrease - key(Q, v, d[v]$
  end for
  $color[u] \leftarrow negro$
end while



9.3.4 Validación

Queremos ver ahora que el algoritmo de Dijkstra es correcto, es decir que calcula las distancias mínimas desde s el vértice origen hasta cualquier otro vértice. Sea $\delta(s, u)$ la distancia mínima de s hasta u.

Teorema: Cuando un vértice u es agregado a S, $d[u] = \delta(s, u)$.

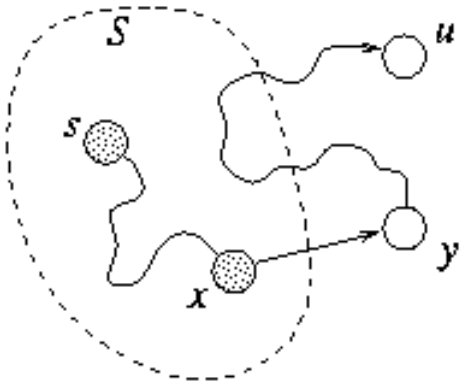
Demostración: Supongamos que en algun momento el algoritmo intenta por primera vez agregar un vértice u a S para el cual $d[u] \neq \delta(s,u)$. De nuestras observaciones sobre la relajación sabemos que $d[u] > \delta(s,u)$. Consideremos la situación anterior a la insercion de u. Consideremos el verdadero camino mínimo desde s hasta u. Como $s \in S$ y $u \in V - S$ en elgún momento el camino debe tomar por primera vez un salto hacia afuera de S. Sea (x,y) la arista que toma el camino donde $x \in S$ y $y \in V - S$. Podemos probar que $y \neq u$.

Como $x \in S$. Tenemos $d[x] = \delta(s,x)$ (porque u es el primer vertice que viola el principio). Entonces cuando aplicamos la relajación a x tendríamos que $d[y] = d[y] + W[u,x] = \delta(s,y)$. Por lo tanto $d[y]$ es correcto y por hipótesis $d[u]$ no es correcto entonces no pueden ser el mismo vértice.

Observemos ahora que como y parece en algun punto del camino desde s hasta u tenemos que $\delta(s,y) < \delta(s,u)$ y por lo tanto.

$$d[y] = \delta(s,y) < \delta(s,u) \leq d[u]$$

Por lo tanto y deberia ser agregado a S antes que u lo cual contradice que u es el próximo vértice que se agrega en S y por lo tanto el algoritmo es correcto.



9.3.5 Análisis

El algoritmo de Dijkstra es $O((n+e)\log n)$. Si se usan heaps de Fibonacci [2] en lugar de Heaps binarios standard el tiempo es entonces $O(n \log n + e)$.

[2] Los heaps de Fibonacci se estudiaran mas adelante en el curso

9.4 El problema del árbol generador mínimo

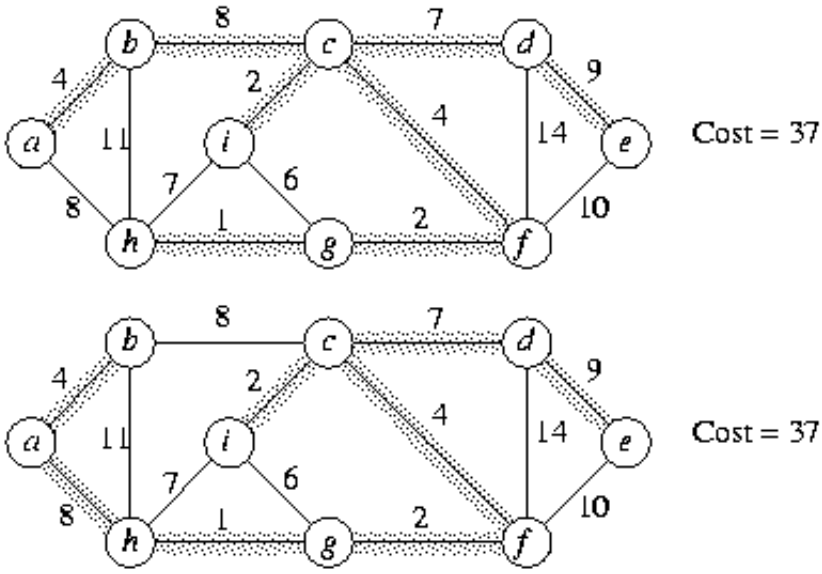
Un problema comun en redes de comunicaciones y disenio de circuitos es aquel que pretende conectar un conjunto de nodos mediante una red de longitud total mínima. (Siendo la longitud la sumatoria de las longitudes de los tramos que integran la red).

Se asume que la red es una red no-dirigida. Para minimizar la longitud de la red nunca será conveniente tener un ciclo ya que podemos sacar una de las aristas que forman el ciclo sin destruir la conectividad y disminuyendo la longitud de la red. El grafo en definitiva debe ser conexo, no dirigido y acíclico, es decir que debe ser un árbol.

El que tratamos se denomima problema del **árbol generador mínimo (AGM)**. Formalmente dado un grafo conexo no dirigido $G = (V, E)$ un árbol generador es un subconjunto de aristas $T \subseteq E$ que conectan todos los vértices sin formar ciclos. Asumiendo que cada arista (u, v) tiene asociado un costo $W(u, v)$ definimos le costo del árbol generador T como la sumatoria de las aristas que integran el árbol.

$$w(T) = \sum_{(u,v) \in T} W(u, v)$$

Un **árbol generador mínimo** (AGM) es un árbol generador de costo mínimo. Y puede no ser único, pero si todos los pesos son distintos entonces el árbol óptimo es único (esto es un tanto sutil y no lo vamos a demostrar). La figura muestra dos árboles generadores mínimos para el mismo grafo.



Presentaremos un algoritmo goloso para hallar el árbol generador mínimo de un grafo. Antes tenemos que hacer algunas consideraciones generales que afortunadamente son fáciles de probar.

Lema

- Un árbol con n vertices tiene exactamente $n-1$ aristas.
- Siempre existe un único camino entre dos vértices de un árbol.
- Agregar una arista cualquiera a un árbol crea un ciclo, eliminando cualquier arista de este ciclo se recupera el árbol.

Sea $G = (V, E)$ un grafo no dirigido conexo cuyas aristas tienen pesos. La intuición detrás de los algoritmos golosos para el (AGM) es simple, mantenemos un subconjunto de aristas A incialmente vacia y agregamos una arista a la vez hasta que A es el (AGM). Decimos que un subconjunto $A \subseteq E$ es viable si A es un subconjunto de aristas en **algún AGM**. Decimos que una arista $(u, v) \in E - A$ es **segura** si $A \cup \{(u, v)\}$ es viable. Es decir que la elección de (u, v) es una elección segura de forma tal que A aun puede ser extendido hasta formar un AGM

Un algoritmo goloso genérico opera agregando en forma repetida cualquier arista segura al AGM actual.

9.4.1 Algoritmo de Kruskal

El algoritmo de Kruskal trabaja agregando aristas a A en orden creciente de peso. (Las aristas menos costosas se agregan primero). Elige la arista mas económica y si la misma no produce un ciclo entonces la agrega al AGM. Cuando no quedan aristas por agregar (no hay mas o bien no se puede agregar ninguna) queda formado el árbol generador mínimo.

La unica parte intrincada del algoritmo es detectar la formación de ciclos. Para ello podríamos aplicar DFS pero eso llevaría demasiado tiempo, el detectar si el agregado de una arista causa un ciclo en A puede hacerse empleando una estructura de datos denominada **estructura Union-Find** [3] la estructura soporta las siguientes operaciones.

- Create-Set(u). Crea un conjunto conteniendo u
- Find-Set(u). Encuentra el conjunto que contiene un elemento u.
- Union(u,v). Realiza la unión de los conjuntos que contienen u y v en un unico conjunto.

Usando esta estructura podemos crear un conjunto por cada árbol del bosque A, de esta forma detectamos que se forma un ciclo si agregamos una arista cuyos vértices pertenecen al mismo conjunto.

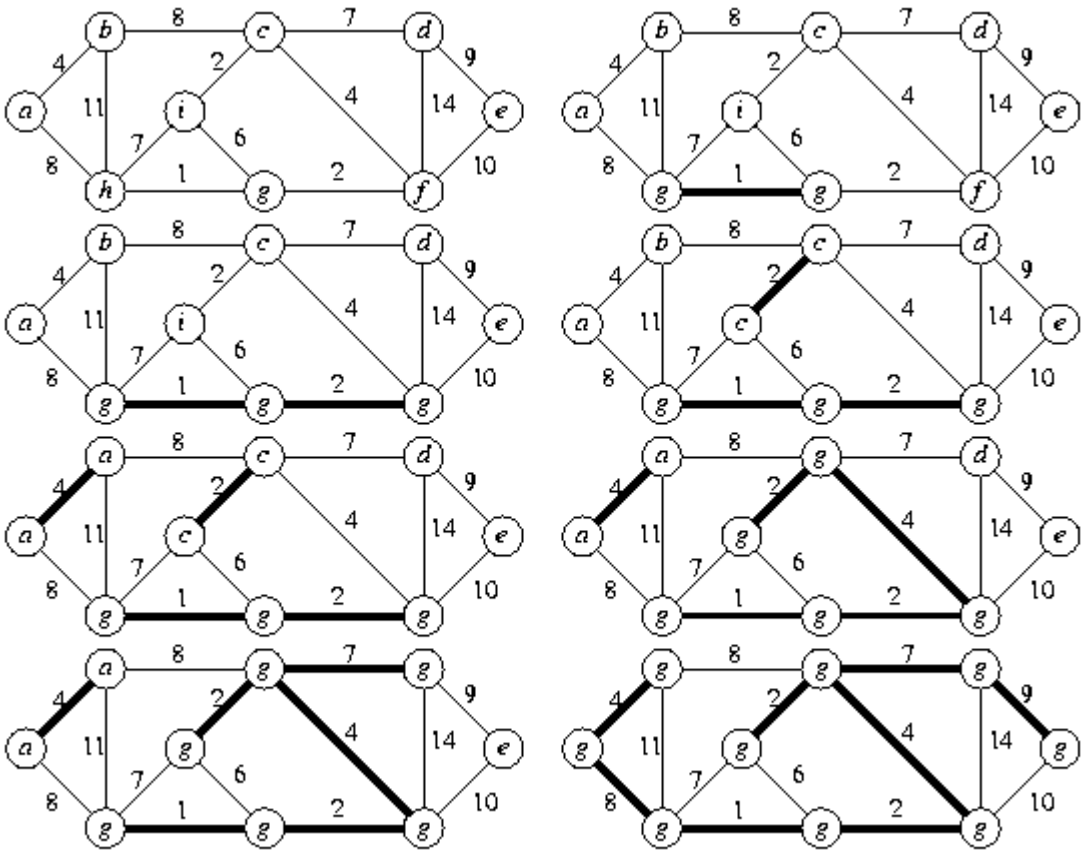
Veamos como podemos usar esta estructura como una caja negra en la cual cada operacion se hace en $O(\log n)$[4].

En el algoritmo de Kruskal los vértices del grafo son los elementos a almacenar en los conjuntos y los conjuntos serán vértices en cada árbol de A. El conjunto A puede guardarse como una lista simple de aristas.

[3] Esta estructura y su implementación se estudian mas adelante en el curso
[4] de hecho hay implementaciones aun mas rápidas

Algorithm 51 Kruskal(G,W).Calcula el AGM del grafo G

$A \leftarrow \{\}$
for each u in V **do**
  Create-Set(u)
end for
Sort E de acuerdo al peso w.
for each (u, v) de la lista ordenada **do**
  if Find-Set(u) $\neq$ Find-Set(v) **then**
    Add (u, v) to A
    Union(u,v)
  end if
end for
return A



9.4.2 Análisis

El algoritmo de Kruskal recorre todas las aristas del grafo y por cada arista realiza operaciones que son $O(\log n)$ por lo tanto el orden del algoritmo es $O((n+e)\log n)$

9.5 Resumen y puntos clave

Los algoritmos golosos permiten resolver problemas de optimización en forma extremadamente sencilla. En algunos problemas los algoritmos golosos generan una solución óptima pero en otros lamentablemente tal cosa no es posible.

Para que un algoritmo goloso sea óptimo deben darse dos condiciones.

- Subsestructura optima El problema debe poder dividirse en subproblemas de forma tal que si la solución al problema es óptima la solución de los subproblemas tambien debe serlo.
- Decisión golosa Debe ser posible demostrar que existe al menos una solución óptima que comienza con la decisión golosa. Por inducción debe poder probarse que mediante una repetición de sucesivas decisiones golosas se llega a un solución óptima.

Como ejemplos vimos el problema de la selección de actividades que es muy sencillo, el algoritmo de Dijkstra para caminos mínimos en un grafo y el algoritmo de Kruskal para generar el árbol generador mínimo en un grafo.

9.6 Ejercicios

1. Una máquina expendedora de pasajes de tren necesita un algoritmo para dar el vuelto usando la menor cantidad de monedas posibles.
   (a) Describir un algoritmo goloso para solucionar el problema para monedas de 5,10,25 y 50 centavos. Demostrar que el algoritmo es óptimo.
   (b) Suponer que las monedas son ahora de $c^0, c^1, c^2, \ldots, c^n$ centavos para $c > 1$ y $k \geq 1$ enteros. Demostrar que el algoritmo anterior es óptimo.
   (c) Indicar si el algoritmo es siempre óptimo independientemente del conjunto de monedas que se utilice, dar ejemplos en los cuales el algoritmo no es óptimo.

Chapter 10

Estructuras de datos avanzadas

10.1 Revisión de alguna estructuras de datos simples

El objetivo de esta clase es analizar diversas estructuras de datos desde el punto de vista de la eficiencia de las primitivas que dichas estructuras soportan. El objetivo de toda estructura de datos es modelar de forma simple y eficiente uno o varios aspectos de un problema. Uno de los paradigmas válidos a la hora de diseniar un algoritmo es modelar el problema en una estructura de datos apropiada de forma tal que las primitivas que soporta la estructura permitan resolver el problema en forma sencilla. Los evaluadores de expresiones que utilizan pilas o árboles son buenos ejemplos de la aplicación de este paradigma. Vamos a empezar revisando algunas estructuras de datos clásicas y luego vamos a estudiar otras mas novedosas.

10.1.1 Pilas

Una pila es una estructura dinámica en la cual la entrada y salida de elementos se hace en orden $LIFO$ (Last-in-first-out). Una pila (stack) soporta las siguientes operaciones.

- Create-Stack() Crea una nueva pila vacia. $O(1)$.
- Empty-Stack() Indica si la pila esta vacia. $O(1)$.
- Top() Devuelve el elemento en el tope de la pila sin sacarlo de la pila. $O(1)$.
- Push(e) Apila un elemento e. $O(1)$.
- Pop() Desapila un elemento y lo devuelve. $O(1)$.

La implementación de pilas es una tarea en extremo sencilla, las pilas suelen ser estructuras muy utilizadas en gran cantidad de algoritmos.

10.1.2 Colas

Una cola es una estructura dinámica en la cual la entrada y salida de elementos se hace en orden $FIFO$ (First-in-first-out). Una cola soporta las siguientes operaciones.

- Create-Queue() Crea una nueva cola vacia. $O(1)$.
- Enqueue(e) Agrega un elemento e a la cola. $O(1)$.
- Dequeue() Extrae un elemento de la cola. $O(1)$.

Las aplicaciones de las colas son muchas.

Colas dobles

En una cola doble se puede encolar o desencolar ya sea desde la cabeza o desde la cola de la cola. Es decir que se pueden manejar los elementos en orden $FIFO$ o en orden $LIFO$ Las colas dobles soportan las siguientes operaciones.

- Create-Queue() Crea una nueva cola vacía. $O(1)$.
- Enqueue-Head(e) Agrega un elemento e a la cabeza de la cola. $O(1)$.
- Dequeue-Head() Extrae un elemento de la cabeza de la cola. $O(1)$.
- Enqueue-Tail(e) Agrega un elemento e a la cola de la cola. $O(1)$.
- Dequeue-Tail() Extrae un elemento de la cola de la cola. $O(1)$.

10.1.3 Listas

Una lista enlazada es una colección dinámica de objetos de propósito general, en su uso mas básico una lista sirve para mantener una relación entre una cantidad variable de objetos, pero hay muchas variantes de listas especializadas en otro tipo de aplicaciones. Una lista enlazada simple provee.

- Create-List Crea una lista enlazada vacia. $O(1)$.
- Insert(e) Agrega un elemento e a la lista. $O(1)$.
- Search(e) Busca un elemento e en la lista. $O(n)$.
- Delete(e) Elimina el elemento e de la lista. $O(n)$.

Algunas variantes de las listas son las listas doblemente enlazadas, las listas ordenadas (en donde se inserta en forma ordenada), las listas circulares, listas dobles circulares, etc.

10.1.4 Arboles Binarios

Un árbol binario es una estructura demasiado popular como para realizar alguna observacion, debemos suponer que el árbol se mantiene ordenado (es decir que no puede insertarse un elemento en cualquier lado). Las operaciones que dependen de la **altura (h)** del árbol son:

- Create-Tree() Crea un árbol vacio. $O(1)$.
- Insert(e) Inserta el elemento e en el árbol. $O(h)$.
- Recorrer() Lista los elementos del árbol en algun orden (inorder, preorder o postorder). $O(n)$.
- Search(e) Busca un elemento e en el árbol. $O(h)$.
- Delete(e) Elimina el elemento e del árbol. $O(h)$.

Las aplicaciones de los árboles binarios son innumerables.

10.1.5 Arboles AVL

En un árbol binario cuando el orden en el cual se insertan los elementos es aleatorio la altura del árbol tiende a ser $h = \log n$, sin embargo si esto no ocurre un árbol binario puede degenerar en una lista enlazada siendo $h = n$, los árboles AVL tambien llamados $Red - Black - trees$ se encargan de mantener al árbol balanceado independientemente de los elementos que se inserten y del orden en que se hagan las inserciones. Un árbol AVL es un árbol binario que ademas de mantenerse ordenado se mantiene balanceado, cuando esto ocurre podemos decir que $h = \log n$ por lo tanto las operaciones que son las mismas que en un árbol binario son:

- Create-Tree() Crea un árbol vacio. $O(1)$.
- Insert(e) Inserta el elemento e en el árbol. $O(\log n)$.
- Recorrer() Lista los elementos del árbol en algun orden (inorder, preorder o postorder). $O(n)$.
- Search(e) Busca un elemento e en el árbol. $O(\log n)$.
- Delete(e) Elimina el elemento e del árbol. $O(\log n)$.

El costo de balancear el árbol es un factor constante que se agrega en cada operación, pero precisamente por ser constante no afecta el orden de las primitivas.

Los árboles AVL son herramientas útiles y bastante poderosas para resolver varias situaciones problemáticas. Por ejemplo son muy eficientes para implementar la parte interna del algoritmo de $ReplacementSelection$ para sort externo.

10.2 Análisis amortizado

El análisis amortizado es una herramienta importantísima para analizar el tiempo de ejecución de las primitivas de una determinada estructura de datos. Supongamos que para una operación Op conocemos el peor caso del tiempo de ejecución, muchas veces este peor caso no es demasiado importante y nos interesa mas conocer cual es el tiempo de ejecución **promedio** si efectuamos n operaciones Op. Sorprendentemente hay operaciones que son individualmente costosas pero que realizadas en serie son mas eficientes.

El análisis amortizado analiza el tiempo promedio de una operación que se supone se realiza n veces.

Las técnicas de analisis amortizado conocidas son las siguientes:

- Método de agregacion
- Método contable
- Método de los potenciales

Vamos a estudiar en esta clase los dos primeros que son los mas simples, el tercero requiere de algunas herramientas que no estan a nuestro alcance en este curso.

10.2.1 Método de agregacion

En el método de agregacion calculamos el peor caso de una serie de n operaciones como $T(n)$ y luego calculamos el costo amortizado de una operación como $T(n)/n$.

$$T(n) = Costo\, de\, n\, operaciones \Rightarrow T(op) = \frac{T(n)}{n}$$

Ejemplo Supongamos que agregamos a la estructura de datos pila la operación *Multipop* que se encarga de remover k elementos de la pila o bien se detiene si en algun momento la pila queda vacia. El algoritmo es:

Algorithm 52 Multipop(S,k). Elimina hasta k elementos de la pila S

```
while not Empty-Stack(S) and k ≠ 0 do
    Pop(S)
    k ← k − 1
end while
```

El costo de multipop depende de la cantidad de pops que se realicen, la cantidad de pops es el minimo entre k y la cantidad de elementos en la pila n, como pop es $O(1)$ el algoritmo Multipop es $O(min(k, n))$.

En el peor caso Multipop elimina n elementos de la pila (todos los elementos) y por lo tanto $Op = O(n)$.

Queremos analizar que ocurre si hacemos n operaciones $Push$, Pop y $Multipop$ en orden aleatorio para n un número grande. Como $Push$ y Pop son $O(1)$ y $Multipop$ es $O(n)$ podríamos decir que el costo de las n operaciones es $n \cdot O(n) = O(n^2)$ con lo que diríamos que $T(n) = O(n^2)$ y por lo tanto $T(n)/n = n$ sin embargo esto no es cierto.

Si analizamos el caso con mejor cuidado podemos ver que por tratarse de una pila un elemento puede ser Popeado a lo sumo tantas veces como fue pusheado, por lo tanto en n operaciones Pop, $Push$, $Multipop$ la cantidad de $Pops$ es menor o igual a la cantidad de $Pushs$, y esto incluye a la operación $Multipop$ que consiste en una cantidad entera de $Pops$, por lo tanto $T(n)$ nunca puede ser $O(n^2)$ sino que es a lo sumo $O(n)$, luego aplicando el método de agregación $T(n)/n = O(n)/n = O(1)$. Por lo tanto el costo amortizado de la operación $Multipop$ es $O(1)$.

Observación adicional La primitiva $Multipop$ es un ejemplo de una **agregación** a una estructura de datos. Una agregación consiste en implementar una primitiva nueva sobre una estructura de datos conocida con el proposito de resolver un problema específico. Muchas veces la solución de un problema se simplifica notablemente realizando una agregación a una estructura conocida. Ademas si la estructura esta bien diseniada en general el agregar una nueva primitiva es sencillo y se reutilizan muchas de las primitivas ya realizadas.

Ejemplo II Nuestro segundo ejemplo es una estructura de datos que simula un contador binario, esta estructura soporta solamente una operación denominada $Increment$, que incrementa el contador en una unidad. El contador esta implementado sobre un vector de k dígitos binarios la operación se realiza de la siguiente forma:

Algorithm 53 Increment(A). Incrementa el contador A.

```
i ← 0
while i < len(A) and A[i] = 1 do
    A[i] ← 0
    i ← i + 1
end while
if i < len(A) then
    A[i] ← 1
end if
```

Este algoritmo es exactamente el mismo que se utiliza en hardware para incementar un registro. Queremos medir el costo de la operación como la cantidad de bits que se cambian de estado, en el peor caso $A = 1111111$ y al incrementarlo hay que cambiar todos sus bits por lo que el costo es $O(k)$.

Si hacemos n operaciones, sin embargo, el costo no es $O(nk)$. Observemos un seguimiento del contador y de la cantidad de bits que se cambiaron en forma acumulada.

Valor	Representacion	Costo	Acumulado
0	000000	0	0
1	000001	1	1
2	000010	2	3
3	000011	1	4
4	000100	3	7
5	000101	1	8
6	000110	2	10
7	000111	1	11
8	001000	4	15

Observemos que luego de incrementar n veces el contador el costo acumulado nunca es mayor que $2n$, observemos tambien que el bit menos significativo cambia su estado $n/2$ veces, el segundo bit menos significativo cambia su estado $n/4$ veces la cantidad de bits que se cambian al incrementar el contador n veces es entonces.

$$\sum_{i=0}^{\log n} \frac{n}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Lo cual concuerda con nuestra observacion de que el costo acumulado nunca superaba $2n$. De lo anterior el peor caso de n operaciones es $T(n) = O(2n)$ por lo que aplicando el método de agregación el costo de la operación *Increment* es $T(2n)/n = O(1)$

10.2.2 Método contable

En el método de agregación suponíamos que todas las operaciones tenían un costo igual al realizarse n veces e igual a n por el costo de la operación, en el método de contabilidad se asigna a cada operación un valor que se denomina **costo amortizado**, y luego se realiza una simulación de n operaciones, cada vez que se realiza una operación se debe pagar lo que la operación cuesta y se cuenta para ello con el costo amortizado, si el costo amortizado no alcanza para pagar la operación hay que reajustar el costo y volver a comenzar. Si el costo sobra la estructura de datos recibe un cierto **crédito** que puede ser usado si otras operaciones no pueden pagar su costo. [1]. Veamos un ejemplo para ver de

[1] Todo esto debería resultarles verdaderamente raro, si asi ocurre no se asusten

que se trata este asunto.

Ejemplo 1 Volvamos al ejemplo de los n $Push, Pop, Multipop$. Asignemos los siguientes costos amortizados.

```
PUSH = 2
POP = 0
MULTIPOP = 0
```

Un pop o un multipop no pueden hacerse si antes no se hizo al menos un push, cuando se hace un push que insume $O(1)$ disponemos de 2 unidades, por lo que la estructura recibe 1 crédito. A medida que se hacen mas y mas operaciones se acumula un crédito por cada Push, cada pop insume $O(1)$ por lo que consume un crédito, $Multipop$ consume una cantidad de créditos variable pero podemos demostrar que siempre hay créditos suficientes para Multipop, por lo tanto como Pop es $O(1)$ y tiene costo amortizado cero $Multipop$ también debe ser $O(1)$.

El método contable es mas "fantástico" que el método de agregación y requiere mucha práctica.

10.2.3 Método de los potenciales

El método de los potenciales es mas complejo que los dos anteriores pero es tambien mas poderoso. La idea es definir una funcion potencial ϕ que mapea una estructura de datos a un numero real. ϕ mide la cantidad de esfuerzo que hemos aplicado a la estructura de forma tal de poder aprovecharlo en futuras operaciones, el concepto es análogo al usado en física cuando hablamos de energia potencial.

Es importante destacar que el potencial depende unicamente de la estructura de datos en si misma, no de las operaciones que han sido aplicadas sobre dicha estructura en el pasado. En algunas ocasiones sin embargo es posible modificar la estructura de datos de forma tal de llevar un registro en algun lugar de almacenamiento de las operaciones que le fueron aplicadas, en estos casos la función potencial podría depender de dicho lugar de almacenamiento.

El costo amortizado de la operación i se define como $c'(i)$ donde:

$$c'(i) = c(i) + \phi(D(i)) - \phi(D(i-1))$$

Siendo D_i la estructura de datos que resulta luego de la operación $i-1$ y Donde $c(i)$ es le costo verdadero asociado a la operación, por lo tanto:

$$\sum i = 1^n c'(i) = \sum_{i=1}^{n} c(i) + \phi(D(n)) - \phi(D(0))$$

Por lo tanto si ϕ es siempre no-negativa y $\phi(D(0)) = 0$ el costo amortizado:

$$\sum_{i=1}^{n} c'(i)$$

sirve como cota superior del costo verdadero.

Ejemplo Volviendo al ejemplo de *Multipop* definamos ϕ = Cantidad de elementos en el stack. Entonces:

$$c'(i) = c(i) + \phi(D(i)) - \phi(D(i-1))$$

$$= \left\{ \begin{array}{ll} 1+1=2 & POP \\ 1+(-1)=0 & POP \\ k+(-k)=0 & Multipop \end{array} \right\}$$

Por lo tanto $C'(i)$ esta acotado por 2 y ese es el límite del costo amortizado por operación.

El arte reside en encontra la función potencial en forma ingeniosa, por ejemplo que función deberiamos usar para el ejemplo del contador binario?

Resumen Debemos notar que el análisis amortizado es importante ya que en general el costo amortizado representa en forma mas práctica y real el costo de una primitiva y cuando de utilizar primitivas se trata debemos ser sumamente precisos en sus costos para no afectar en forma innecesaria el costo de los algoritmos que las utilizan.

10.3 Heaps Binomiales

Al estudiar el algoritmo de HeapSort estudiamos una estructura de datos muy importante denominada *Heap*, ahora vamos a extender esta estructura a otra de mayor envergadura denominada Heaps Binomial. Un Heap Binomial soporta las siguientes operaciones:

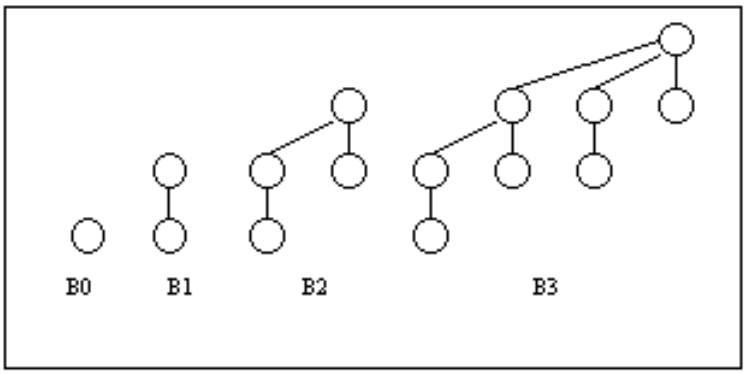
- Make-Heap Construye un heap nuevo vacío.
- Insert(H,x) Inserta el elemento x en el Heap H.
- Minimum(H) Devuelve el elemento minimo del Heap H.
- Extract-Min(H) Devuelve el minimo y ademas lo remueve del Heap H.
- Union(H1,H2) Realiza la union de dos Heaps en uno único.
- Delete(H,x) Elimina el elemento x del Heap H.
- Decrease-Key(H,x,k) Cambia el elemento x por el elemento k que es menor.

En un Heap comun las operaciones $Make - Heap$ y $Minimum$ son $O(1)$ y las restantes excepto la union son $O(\log n)$. La operación de union en un heap común es $O(n)$. Los Heaps Binomiales y los Heaps de Fibonacci tienen como objetivo realizar estas operaciones en forma mas eficiente.

Para construir un Heap binomial necesitamos previamente el concepto de **árbol binomial**.

10.3.1 Arboles Binomiales

Un árbol binomial es un árbol binario. Un árbol binomial de orden k esta formado por dos árboles binomiales de orden $k-1$ en donde la raiz del árbol de orden k es la raiz del árbol derecho de orden $k-1$. El gráfico muestra algunos árboles binomiales.



Propiedades Las siguientes propiedades son válidas en un árbol binomial de orden k.

1. B_k tiene B^k nodos.
2. La altura de B_k es k
3. B_k tiene exactamente $\begin{pmatrix} k \\ i \end{pmatrix}$ nodos en el nivel i.
4. La raiz de B_k tiene grado k y no hay otro nodo en B_k con grado mayor o igual que k
5. Si los hijos de B_k son rotulados de izquierda a derecha como $k-1, k-2, k-3, \ldots, 0$ entonces el hijo i es la raiz de un B_i

Todas las propiedades se pueden probar en forma simple por inducción.

10.3.2 Estructura de un Heap Binomial

Un **Heap binomial** es una estructura de datos que consiste en una lista de árboles binomiales (las raíces de los árboles forman una lista) con las siguientes propiedades.

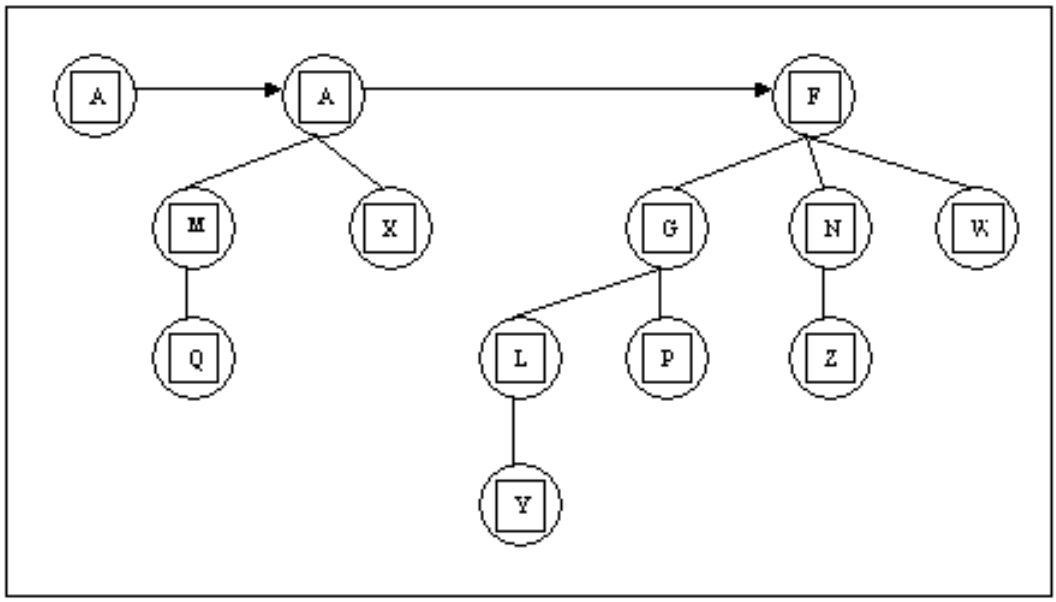
1. Cada árbol binomial esta en **Heap-order** es decir que la clave de un nodo en uno cualquiera de los árboles es mayor que la clave de sus hijos.
2. El heap no contiene dos árboles de igual orden.
3. La lista de árboles se mantiene en orden creciente de acuerdo al grado de la raiz.

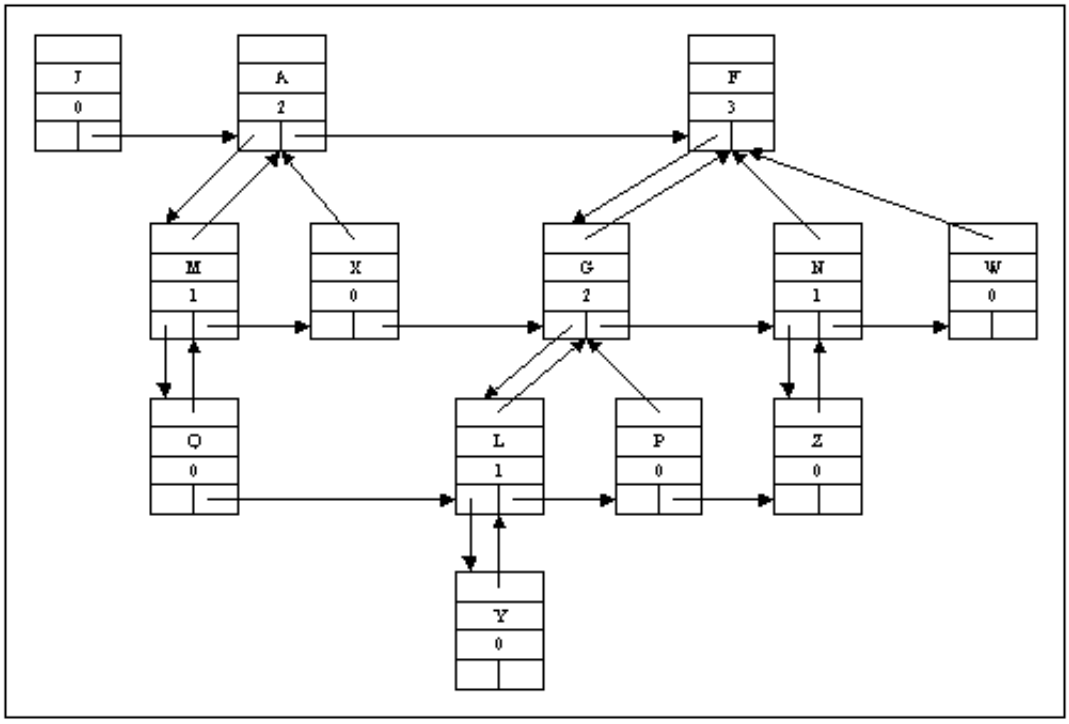
10.3.3 Implementación

Para implementar un Heap binomial se recurre a un nodo que presenta la siguiente estructura:

```
*  Puntero al nodo padre.
*  Valor de la clave.
*  Grado del nodo. (cantidad de hijos)
*  Puntero al hijo izquierdo.
*  Puntero al hermano derecho.
```

Esta estructura es suficiente para todos los nodos de un heap binomial, el puntero al hermano derecho en los nodos que son raiz del árbol binomial sirven para mantener la lista de árboles. La ilustración muestra un heap binomial y su implementación.


A
A
F
M
X
G
N
W
Q
L
P
Z
Y


J
0
A
2
F
3
M
1
X
0
G
2
N
1
W
0
Q
0
L
1
P
0
Z
0
Y
0

10.3.4 Implementación de las operaciones

Create-Heap

Para crear un nuevo Heap-Binomial hay que crear un nodo con los punteros y grado seteados apropiadamente, esto insume $\Theta(1)$.

Minimum

Se recorre la lista de raíces, esto funciona porque cada árbol binomial esta en orden de Heap, por lo tanto el elemento mínimo es alguna de las raices de los árboles binomiales. Cuando hay n nodos los tamanios de los árboles binomiales en el Heap corresponden a las potencias de 2 que aparecen en la expansión binaria de n. Por ejemplo para $n = 13 = 8 + 4 + 1$ tendremos un B_0, un B_2 y un B_3. Esto implica que a lo sumo hay $1 + \log n$ árboles binomiales por lo tanto este algoritmo insume $O(\log n)$.

Algorithm 54 Minimum(H). Devuelve la clave mínima de un Heap Binomial H.

```
y ← NULL
x ← head(H)
min ← ∞
while x ≠ NULL do
  if key[x] < min then
    min ← key[x]
    y ← x
  end if
  x ← sibling[x]
end while
return y
```

Union

Para hacer la union de dos heaps binomiales tomamos la lista de raíces de cada Heap y realizamos un **merge** de acuerdo al grado de cada raiz. Luego hacemos un merge de los árboles binomiales de igual grado.

En primer lugar presentamos un algoritmo que dados dos árboles binomiales de grado k representados por sus nodos raices construye un nuevo árbol binomial de grado $k + 1$ donde z es la raiz del árbol.

Como podemos ver Binomial-Link es $O(1)$. El algoritmo que realiza la Union es complejo pues se deben manejar varios casos, el algoritmo funciona en $O(\log n)$.

Algorithm 55 Binomial-Link(y,z). Genera un nuevo árbol binomial a partir de y y z

$p[y] \leftarrow z$
$sibling[y] \leftarrow child[z]$
$childz \leftarrow y$
$degree[z] \leftarrow degree[z] + 1$

Algorithm 56 Union(H1,H2). Realiza la union de dos heaps binomiales.

$h \leftarrow Make - Binomial - Heap()$
$head[H] \leftarrow Binomial - Heap - Merge(H1, H2)$
Liberar los objetos H1 y H2 pero no las listas
if $head[H] = NULL$ **then**
  return H
end if
$prev - x \leftarrow NULL$
$x \leftarrow head[H]$
$next - x \leftarrow sibling[x]$
while $next - x \neq NULL$ **do**
  if $degree[x] \neq degree[next - x]$ or ($sibling[next - x] \neq NULL$ and $degree[sibling[next - x]] = degree[x]$) **then**
    $prev - x \leftarrow x$
    $x \leftarrow next - x$
  else
    if $key[x] \leq key[next - x]$ **then**
      $sibling[x] \leftarrow sibling[next - x]$
      $Binomial - Link(next - x, x)$
    else
      if $prev - x = NULL$ **then**
        $head[H] \leftarrow next - x$
      else
        $sibling[prev - x] \leftarrow next - x$
      end if
      $Binomial - Link(x, next - x)$
      $x \leftarrow next - x$
    end if
  end if
  $next - x \leftarrow sibling[x]$
end while

Insert

Para insertar un elemento en un Heap-Binomial simplemente construimos un Heap vacío, lo inicializamos con el elemento a insertar y luego hacemos la Union entre el Heap creado y el Heap donde queremos insertar el elemento. Como solo agregamos tiempos constantes a la union este algoritmo es $O(\log n)$.

Extract-Min

Hay que encontrar la raiz con clave mínima (de la lista de raices). Remover esta raiz y su árbol asociado de la lista, tomando el árbol que estaba asociado a esta raiz y quitando la raiz obtenemos k árboles binomiales que ponemos en una lista enlazada. Luego hacemos la Union (primitiva de Union) par juntar esta lista con la lista anterior.

Decrease-Key

Se mueve la nueva clave hacia la raiz hasta que es $\geq$ que su padre.

Delete

Se decrementa la clave (decrease-key) hasta $-\infty$ de esa forma el nodo llega a la lista raiz, una vez alli usamos $Extract - Min$ para eliminarlo.

10.4 Heaps de Fibonacci

Los Heaps de Fibonacci son otra forma de implementar Heaps que soportan las mismas operaciones que los heaps binomiales, algunas de las operaciones de los heaps de Fibonacci pueden hacerse en menos tiempo que un Heap Binomial y por eso la estructura merece atención.

10.5 Resumen de Heaps

El siguiente cuadro resume el orden de las operaciones para las tres estructuras de Heap estudiadas.

Primitiva	Heap	Heap Binomial	Heap de Fibonacci
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
Union	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
Decrease-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

10.6 Union-Find

La estructura **Union-Find** es una estructura de datos que implementa una serie de **conjuntos disjuntos**. Las operaciones que se permiten son:

- Make-Set(x) Crea un nuevo conjunto que contiene el elemento x es decir $\{x\}$.
- S=Union(x,y) Toma dos conjuntos como parámetros y realiza la union , el resultado es un nuevo conjunto cuyo nombre es el nombre de algunos de los conjuntos que lo originaron , los conjuntos anteriores son destruidos.
- Find(x) Devuelve el *nombre* del conjunto que contiene al elemento x. El *nombre* en la mayoría de las implementaciones es un elemento representativo del conjunto[2] de forma tal que $Find(x) = Find(y)$ si x e y pertenecen al mismo conjunto.

Aplicación La estructura Union-Find ya habia sido mencionada en el algoritmo de Kruskal, y tiene varias aplicaciones importantes mas. Por ejemplo supongamos que $G = (V, E)$ es un grafo no dirigido. Decimos que el vértice v esta en el mismo *componente conexo* que el vértice w si hay un camino usando ejes de E que conecta v y w. Dado un par de vértices queremos saber si estan en el mismo componente conexo. Podemos resolver el problema de la siguiente forma:

Algorithm 57 Connected-Components(G)

for each v in V **do**
  $Make - Set(v)$
end for
for each (u, v) in E **do**
  if Find-Set(u) ¡¿ Find-Set(v) **then**
    $Union(u, v)$
  end if
end for

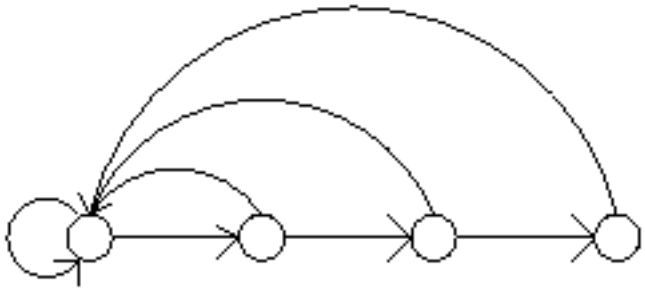
[2]Un elemento cualquiera sirve ya que los conjuntos son disjuntos

Algorithm 58 Same-Component(u,v)

```
if Find-Set(u)=Find-Set(v) then
  return TRUE
else
  return FALSE
end if
```

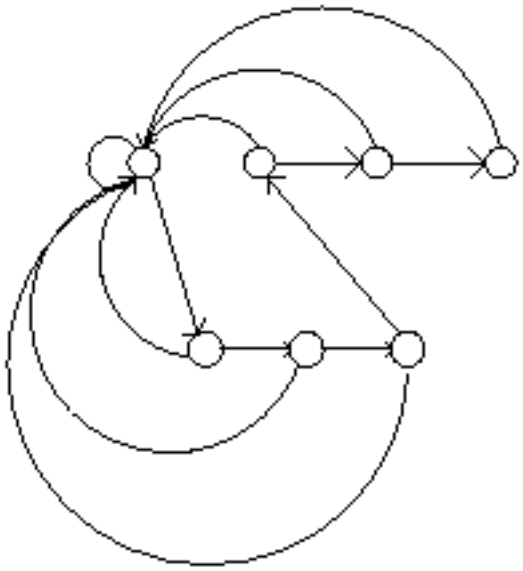
10.6.1 Implementación usando listas

En nuestra primera implementación (simple). Cada conjunto va a estar representado por una lista enlazada y el nombre del conjunto va a ser el primer elemento de la lista. Cada elemento de la lista tiene un puntero al primer elemento de la lista y al siguiente.



Las operaciones se implementan de la siguiente manera.

- Make-Set(x) Creamos una nueva lista que contiene x. $O(1)$
- Find-Set(x) Usamos el puntero a la cabeza de la lista desde x. $O(1)$
- Union(x,y) Hay que realizar la union de dos listas y actualizar los punteros a la cabeza (ver figura). Esta operación se denomina splice, el splice de $L1$ en $L2$ implica hacer que la cabeza de $L1$ apunte al primer elemento de $L2$ y el ultimo elemento de $L2$ apunte al elemento que antes era la cabeza de $L1$, esto engancha ambas listas. Ademas hay que actualizar todos los punteros a la cabeza de $L2$ para que apunten ahora a la cabeza de $L1$. $O(n)$.



Para m operaciones esta implementación requiere $\Theta(m^2)$.

10.6.2 Segunda Implementación usando listas

Para mejorar el tiempo necesario para realizar la operación de Union, agregamos un campo a cada nodo de la lista. Para los elementos a la cabeza de una lista el campo contiene la cantidad de elementos en la lista, para el resto de los elementos este campo no se usa.

Con esta implementacion las operaciones $Make-Set$ y $Find-Set$, no cambian.

Union(x,y) Para realizar el splice el mismo se hace de la lista que tiene menos elementos en la otra. De esta forma tenemos que actualizar menor cantidad de punteros, veamos como afecta esto el orden de la Union.

Realicemos un análisis amortizado para m operaciones,en el viejo algoritmo de Union esto insumía $\Theta(n^2)$.

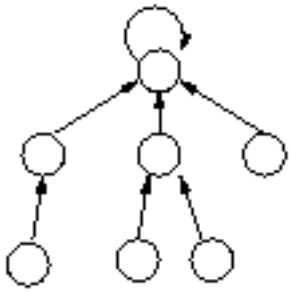
Consideremos cuantas veces cambia el nombre de un objeto (el puntero a la cabeza). Cada vez que el nombre de un objeto x cambia sabemos que x pertenecía a un conjunto de menos elementos de los dos que participan en la Union. La primera vez que x es actualizado el conjunto resultante pasa a tener

dos conjuntos, la segunda vez pasa a tener al menos 4 elementos (en caso contrario el conjunto de x no seria el mas chico). Por lo tanto por cada vez que se actualiza un puntero el tamanio del conjunto por lo menos se duplica. Como un conjunto puede tener a lo sumo n elementos la cantidad de veces que el nombre de un objeto puede ser actualizado es $\log n$. Como hay n objetos el costo total es $O(n \log n)$. Si consideramos m operaciones (Make-Set,Find-Set,Union) vemos que Make-Set y Find-Set son $O(1)$ por lo tanto el total es $O(m + n \log n)$.

10.6.3 Implementación usando Forests

En esta implementacion cada conjunto del grupo $Union-Find$ esta representado por un $Up-Tree$[3], la implementación cumple las siguientes propiedades.

1. Cada nodo de un árbol representa un elemento del conjunto.
2. Cada nodo apunta a su padre.
3. La raiz es padre de si mismo.
4. El nombre del árbol es su raiz.



Operaciones

- Make-Set(x) Crea un nuevo nodo con un puntero a si mismo.

[3]Cada nodo apunta a su padre

- Union(x,y) Apunta la raiz del árbol de menor altura a la raiz del árbol mas

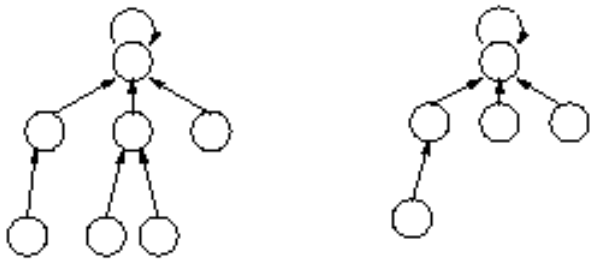


Figure 1: Before Union

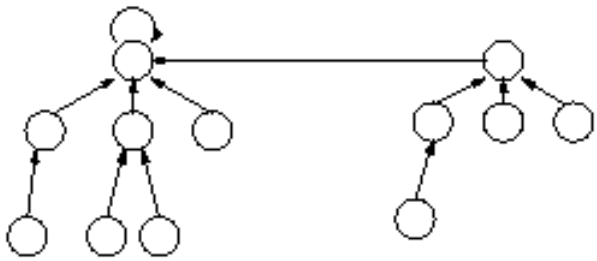


Figure 2: After Union

alto.

- Find(x) Desde el nodo x se siguien los punteros hasta la raiz del árbol que el nombre del conjunto.

Path-Compression Una optimización interesante es la **compresión de caminos**. Cada vez que hacemos un $Find$ del nodo x y el $Find$ recorre el árbol hasta la raiz xr, pasando por los nodos $x2, x3, \ldots xr$ hacemos que xr sea el nuevo padre de todos estos nodos. De esta forma cualquier $Find$ que hagamos para estos nodos va a tardar $O(1)$.

Algorithm 59 AL: Make-Set(x).

$p[x] \leftarrow x$
$rank[x] \leftarrow 0$

Algoritmos

Algorithm 60 Union(x,y)

LINK(Find-Set(x),Find-Set(y))

Algorithm 61 LINK(x,y)

if $rank[x] > rank[y]$ **then**
  $p[y] \leftarrow x$
else
  $p[x] \leftarrow y$
  if $rank[x] = rank[y]$ **then**
    $rank[y] \leftarrow rank[y] + 1$
  end if
end if

Algorithm 62 Find-Set(x), sin path compression

if $x \neq p[x]$ **then**
  $p[x] \leftarrow Find - Set(p[x])$
end if
return p[x]

Notemos que cada nodo tiene un campo **rank**. El "rank" de un nodo es una cota superior de la altura del nodo. Inicialmente es cero y solo se incremente en 1 cuando el nodo es una raiz y otra raiz de igual rango es linkeada como un nuevo hijo.

Análisis Puede probarse que si hacemos m operaciones $\{MakeSet, Union, Find\}$ y n de ellas son $Make - Set$ el tiempo total es $O(m\alpha(m,n))$, donde $\alpha(m,n)$ es la función inversa de Ackermann, que se define de la siguiente forma:

$$\alpha(m,n) = min\{i \geq 0 : A(i, \lfloor m/n \rfloor) > \log n\}$$

Esta definición depende de $A(m,n)$ que es la función de Ackerman, que se define de la siguiente forma:

$$A(1,j) = 2^j \; para \; j \geq 1$$

$$A(i,1) = A(i-1,2)\, para\, i \geq 2$$
$$A(i,j) = A(i-1, A(i,j-1))\, para\, i,j \geq 2$$

Esta función crece mas rápido que practicamente cualquier función que podamos imaginar. Por lo tanto $\alpha(m,n)$ es una función de crecimiento increiblemente lento, para cualquier caso práctico podemos suponer que $\alpha(m,n) \leq 4$.

No podemos[4] probar que el costo de la estructura $Union-Find$ es $O(m\alpha(m,n))$, pero podemos probar un límite mas débil que es $O(m \log^* n)$. Donde la función $\log^* n$ es una función de crecimiento extremadamente lento y que definimos informalmente como la cantidad de veces que debemos aplicar la función log a n para obtener un numero ≤ 1. Notemos que para todo $n < 2^{65535}, \log^* n < 5$ Podemos asumir entonces que $\log^* n < 5$ para cualquier caso práctico[5]. .

Teorema 1 Tenemos que $rank(x) < rank(p[x])$ salvo que $x = p[x]$. Inicialmente $rank(x) = 0$ y $rank(x)$ crece en forma monotona hasta $x \neq p[x]$.

Demostración Los nodos son linkeados de forma tal que el padre tiene mayor rank o si ambos tienen el mismo rango entonces el rango del padre se incrementa. Por lo tanto $rank(x) < rank(p[x])$ salvo que x sea su padre (x es raiz). Cuando x ya no es una raiz su rango no puede cambiar. Lqqd.

Teorema 2 Sea $Size(x)$ el numero de nodos en el sub-árbol cuya raiz es x. Entonces $size(x) \geq 2^{rank(x)}$

Demostración Por inducción en $size(x)$. Claramente es verdad cuando $size(x) = 1$, ya que $rank(x) = 0$. Cuando y es linkeado a x entonces si $rank(y) < rank(x)$, $size(x)$ aumenta pero el rango no cambia por lo que la desigualdad se mantiene.

Si $rank(y) = rank(x) = n$ entonces antes de linkear la desigualdad se mantiene para x y para y, por lo tanto $size(x) \geq 2^n$ y $size(y) \geq 2^n$. Luego de que los nodos son linkeados, tenemos $size(x') \geq 2^{n+1}$ y $rank(x') = rank(x) + 1$ donde x' es el *nuevo* x. Por lo tanto la desigualdad se mantiene. Lqqd.

Teorema 3 En cualquier momento hay a lo sumo $n/2^r$ nodos de rango r.

Demostración Supongamos que cuando el nodo x recibe el rango r rotulamos todos los nodos en el sub-árbol de x con el rotulo x. Notemos que ningun nodo es rotulado mas de una vez, por el Teorema 1. Por el teorema 2, $size(x) \geq 2^r$, por lo tanto cada vez por lo menos 2^r nodos son rotulados. Si hubiera mas de $n/2^r$ nodos de rango r habria un total de $(n/2^r)(2^r)$ nodos rotulados lo cual sería una contradicción. Lqqd.

[4] No queremos, no debemos, no sabemos

[5] Es importante que quede claro que estamos hablando de funciones de crecimiento extraordinariamente lento

Corolario El rango mas grande que puede tener un nodo es $\log n$

Teorema 4 Supongamos que reemplazamos una serie S' de m' operaciones Make-Set, Union y Find por una serie S de m operaciones Make-Set, Link y Find (reemplazando cada Union por dos Finds y un Link). Entonces si S insume $O(m \log^* n)$ operaciones S' insume $O(m' \log^* n)$ operaciones.

Demostración Observemos que una Union consiste de dos Finds y un Link. Por lo tanto $m' \leq m \leq 3m'$.

Por el Teorema 4 podemos contar la cantidad de operaciones en función de la cantidad de Make-Sets. Links y Finds e ignorar el costo de las uniones.

Ahora estamos listos para probar que $O(m \log^* n)$ acota el tiempo de ejecución de la estructura Union-Find. Recordemos que m es la cantidad de operaciones Make-Set, Link, Find y n es la cantidad de nodos en total (la cantidad de Make-Set).

Cada Make-Set es $O(1)$ y hay n de ellos entonces el tiempo de los Make-Sets es $O(n) = O(m)$ ya que $m \geq n$.

Cada Link insume $O(1)$ y hay $O(m)$ Links, el tiempo total es por lo tanto $O(m)$.

Queda por contar el costo de los *Finds*. Cuando hacemos un *Find* de un nodo $x0$ recorremos el camino $x0, x1, \ldots, xi$ desde $x0$ hasta la raiz xi. El costo total del Find es la cantidad de nodos del camino. Vamos a calcular este costo rotulando cada nodo en cada camino recorrido por un Find con una letra B (costo de bloque) o una letra P (costo de camino). Una vez que esto fue realizado sumamos el total de Bs y Ps para obtener el costo total.

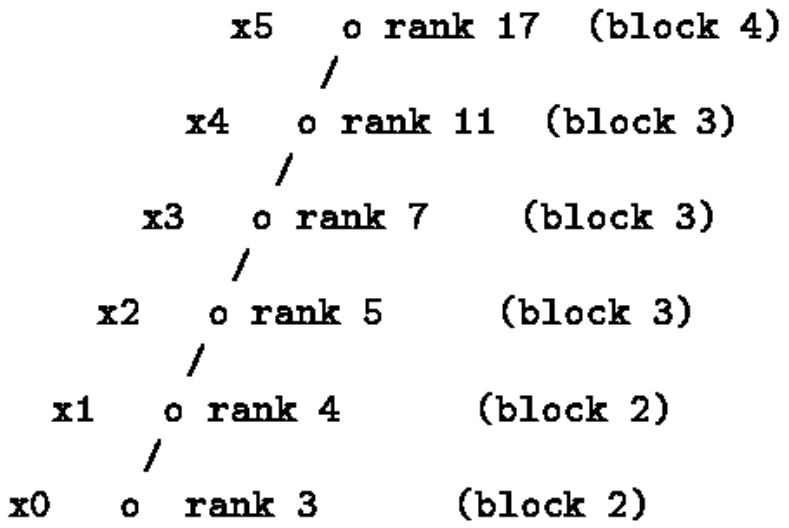
Definimos un *Bloque* de la siguiente manera: ponemos a cada nodo de rango r en el bloque $\log^* r$ por ejemplo:

rango	bloque
0,1	0
2	1
3,4	2
5,6,...,16	3
17,...,65536	4
etc	etc

Los costos se asignan de la siguiente forma: Si un nodo es el ultimo en el camino hacia la raiz con su rango en un bloque dado, rotulamos el nodo B. El hijo del nodo rotulado B tambien recibe una B, el resto de los nodos del camino son rotulados con P. Tal vez queda mas claro de esta forma:

xi es rotulado con B si $p[xi] = xi$ (es la raiz o su hijo=, o si $\log^* rank(xi) < log^* rank(p[xi])$. (el nodo y su padre estan en distintos bloques). Los restantes nodos del camino se rotulan con P.

Ejemplo Supongamos que hacemos un Find desde el nodo $x0$ y los rangos son los siguientes:



En el ejemplo $x1$, $x4yx5$ recibirían el rotulo B mientras que $x0$, $x2yx3$ recibirían el rotulo P.

Para computar el costo notemos que por el corolario del Teorema 3 hay a lo sumo $\log^* n$ bloques diferentes, por lo tanto hay a lo sumo $(\log^* n) - 1$ nodos que perteneces a un bloque distinto que el de su padra. Sumando 2 por la raiz y el hijo de la raiz tenemos a lo sumo $1 + \log^* n$ rotulaciones con B. El total de operaciones $Find$ es $O(m)$, por lo tanto el total de rotulaciones B es $O(m \log^* n)$

Para contar las veces que rotulamos con P la tecnica no la incluimos porque resulta demasiado extensa. De lo anterior y de la cuenta de los rótulos P se llega a $O(m \log^* n)$

Como puede verse esta implementación de la estructura Union-Find es extremadamente eficiente, el uso de estructuras de datos apropiadas implementadas en forma eficiente es un punto clave en el disenio de algoritmos, la estructura Union-Find es de utilidad en gran cantidad de algoritmos, sobre todo en aquellos de tipo **constructivo** como por ejemplo el algoritmo de Kruskal.

10.7 Hash-Tables

Una de las estructuras de datos mas importantes es la familia de los *diccionarios*. Un diccionario es una estructura de datos que permite almacenar elementos de la forma $(clave, descripcion)$ donde la clave identifica univocamente a cada elemento y la descripción puede ser absolutamente cualquier cosa.

Las operaciones básicas sobre esta estructura son tres: insertar un nuevo elemento en el diccionario. Buscar un elemento en el diccionario y opcionalmente recuperar su descripción y por último eliminar un elemento del diccionario.

Para implementar este tipo de estructura existen varias opciones.

En primer lugar se puede usar una lista enlazada con lo cual logramos una buena utilización de espacio en memoria (solo usamos lugar para los elementos insertados) pero perdemos mucha eficiencia al recuperar datos. La búsqueda en una lista enlazada es $O(n)$, por ejemplo.

La segunda opcion consiste en usar una tabla de acceso directo o Look-Up-Table, donde cada clave tiene una posición reservada en la tabla, de esta forma la mayoría de las operaciones son $O(1)$ pero como desventaja necesitamos una tabla con capacidad de almacenar todas las claves posibles. Si la clave es por ejemplo un numero de documento este tipo de esquema es inpracticable.

Las Hash-Tables combinan la buena utilización del espacio de las listas enlazadas con la velocidad de acceso de las tablas de acceso directo. Basicamente una Hash-Table es una tabla de acceso directo en donde el acceso a la tabla no se hace directamente por la clave sino aplicando a la clave una *función de hashing*. La función de hashing convierte la clave en una cierta posición de la tabla, la tabla se reserva del espacio que se crea conveniente. Por ejemplo para una clave que es un número de 5 dígitos podemos reservar solamente 100 posiciones y usar como función de hashing la función $f(x) = x \, mod \, 100$[6].

10.7.1 Operaciones

Las operaciones soportadas por una Hash-Table son las siguientes:

[6] x mod y devuelve el resto de la división entera entre x e y

1. Insert(e,key). Inserta un elemento e en la tabla con clave key. $O(1)$
2. Search(key). Busca en la tabla el elemento cuya clave es key. $O(1)$
3. Delete(key). Elimina de la tabla el elemento cuya clave es key. $O(1)$

La clave debe ser una parte del elemento que sirve para identificarlo en forma univoca con respecto al universo de elementos. Por ejemplo el padrón de un alumno sirve como clave para todo el conjunto de alumnos.

10.7.2 Resolución de colisiones

Definamos una **hash-table** como un vector $H[0, m]$ donde $m << |U|$ cantidad de claves posibles. Es decir que el espacio utilizado es muy cercano al espacio necesario.

Una **función de hashing** h mapea las claves a indices de H es decir que $h : U \rightarrow [0..m]$ o sea $x \in U, f(x) = z\, z \in [0..m]$.

El problema es que la función de hashing puede producir **colisiones** lo cual implica que la función de hashing genera el mismo valor para dos claves distintas. $h(k1) = h(k2)$ con $k1 \neq k2$. Por ejemplo si $h(x) = x\, mod 7$ los valores $x = 7$ y $x = 14$ producen una colisión. A las claves que colisionan se las llama sinónimos. Es necesario contar con algún **método de resolución de colisiones** que permita resolver este problema. Las técnicas básicas para resolver colisiones son:

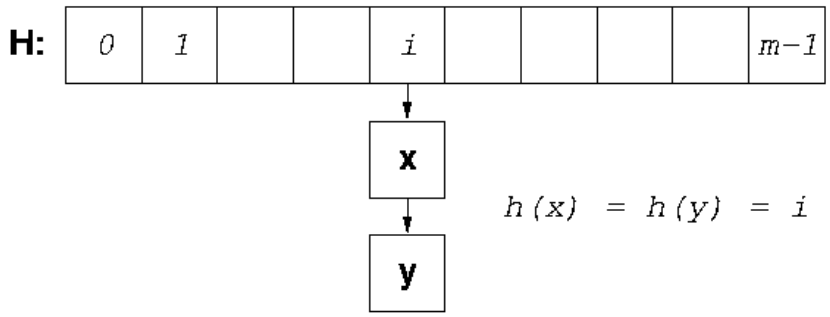
1. Encadenamiento También llamado direccionamiento cerrado, los elementos que colisionan se encadenan en una lista enlazada.
2. Direccionamiento Abierto Cuando ocurre una colisión se busca una posición vacía de la tabla para el elemento en cuestión. Hay varias formas de buscar este lugar:
   - Lineal Se prueba en forma sucesiva y lineal: x,x+1,x+2,...
   - Cuadrático Se prueba mediante valores que se incrementan en forma cuadratica.
   - Doble hashing Una segunda función de hashing determina donde probar.

Mas adelante consideraremos las funciones de hashing, por el momento supongamos que nuestra función de hashing satisface:

Principio de Uniformidad Simple *Cualquier clave tiene la misma probabilidad de ser hasheada a cualquier ubicación de la tabla H. La probabilidad de que a una clave x le corresponda la posición z en la tabla es la misma para todo x*

Resolución de colisiones por encadenamiento

Todos los sinónimos se encadenan en una lista enlazada. De esta forma, cada posición de la tabla es en realidad una lista enlazada que tiene tantos elementos como colisiones hayan ocurrido en dicha posición.



Operaciones

- Insert(x): Calcular $h(x)$ e insertar x en la cabeza de la lista en $H[h(x)]$. $\Theta(1)$ **siempre**.
- Search(x): Calcular $h(x)$ y buscar x en la lisra enlazada en $H[h(x)]$. $O(n)$ en el **peor caso**.
- Delete(x): Calcular $h(x)$ y buscar x en $H[h(x)]$ para eliminarlo. $O(n)$ **peor caso**.

Como en todo análisis de primitivas no estamos tan interesados en el peor caso sino en el caso medio si realizamos n operaciones.

Caso medio para Search(x)

Sea $H = [0..m-1]$

Sea n El numero de elementos en H **Factor de carga de H:** α

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

Caso 1: Caso medio de Search(x) si x no está en la tabla $\Theta(1)$ para calcular $h(x)$ + tiempo promedio para buscar x en $H[h(x)]$

$$\Theta(1) + \Theta(longitud - promedio - de - la - lista)$$

$$\Theta(1) + \Theta(\frac{n}{m})$$

$$\Theta(1+\alpha)$$

Caso 2: Caso medio de Search(x) si x está en la tabla

$\Theta(1)$ para calcular $h(x)$ + cantidad esperada de elementos a examinar en $H[h(x)]$ hasta encontrar x.

Nota: Si x es el $i$esimo elemento agregado a H entonces la longitud esperada de $H[h(x)]$ antes de agregar x es $\frac{i-1}{m}$

$$\Theta(1) + \sum_{i=1}^{n} P(x) \cdot (\frac{i-1}{m} + 1)$$

$$\Theta(1) + \sum_{i=1}^{n} \frac{1}{n} \cdot (\frac{i-1}{m} + 1)$$

$$\Theta(1) + \sum_{i=1}^{n} \frac{i-1}{nm} + \sum_{i=1}^{n} \frac{1}{n}$$

$$\Theta(1) + \frac{1}{nm} \sum_{i=1}^{n} (i-1) + \frac{1}{n} \sum_{i=1}^{n} 1$$

$$\Theta(1) + \frac{1}{nm} \cdot \frac{(n-1)n}{2} + \frac{1}{n} \cdot n$$

$$\Theta(1) + \frac{n}{2m} - \frac{1}{2m} + 1$$

$$\Theta(1) + \frac{1}{2} \cdot \alpha - \frac{1}{2m} + 1 = \Theta(1 + \alpha)$$

Por lo tanto Search(x) es en promedio $\Theta(1 + \alpha)$ lo cual es $\Theta(1)$ si $\alpha = \Theta(1)$ lo cual ocurre si $n < m$.

Resolución de colisiones por direccionamiento abierto

H es un vector simple, todos los elementos van a alguna posición de H. Sucesivamente se *prueban* distintas posiciones de H cuando ocurre una colision hasta encontrar un lugar para x.

La función de hashing recibe como parámetro adicional el numero de intento.

Secuencia de prueba: $h(k,0), h(k,1), \ldots mh(k,m-1)$ establece las posiciones a probar si ocurre una colision. Si $h(k,0)$ esta ocupado probamos con $h(k,1)$ etc. En el peor caso se examina toda la tabla.

Operaciones Para implementar las operaciones en forma funcional debemos tener asociada a cada posicion de la tabla un flag que indique su estado:

libre: La posición nunca fue ocupada.
ocupada: La posición está ocupada por un elemento.
borrada: La posición fue ocupada alguna vez pero el elemento ha sido borrado.

Operaciones

- Insert(x) Calculamos $h(x, i = 0)$. Si está libre o borrado insertamos x alli. Si la posición está ocupada probamos con $h(k, i + 1)$ hasta encontrar un lugar libre o borrado. Marcamos el lugar donde insertamos x como ocupado.
- Delete(x) Calculamos $h(x, i = 0)$. Si la posición está ocupada por x eliminamos el elemento marcandolo como borrado. Si la posición está libre se produce un error x no inexistente. Si la posición está ocupada por otro elemento o está borrada seguimos buscando con $h(x, i+1)$ hasta encontrar al elemento o a una posición libre en cuyo caso la búsqueda se detiene pues si el elemento existiera debería estar alli.
- Search(x) Calculamos $h(x, i = 0)$. Si la posición está ocupada por x fin. Si la posición está ocupada por otro elemento o está borrada seguimos buscando con $h(x, i + 1)$, si al buscar nos topamos con una posición libre termina la búsqueda con x inexistente.

Principio de uniformidad: *Toda clave tiene la misma probabilidad de tener a cualquiera de las $m!$ permutaciones posibles de índices de H como su secuencia de prueba.*

Notemos que es distinto del principio de uniformidad simple.

Direccionamiento abierto lineal

$$h(k, 0) = h(k)$$

$$h(k, i) = (h(k, i - 1) + 1) \, mod \, m$$

Notemos que la secuencia de pruebas es cíclica, de la forma:

$$x, x + 1, x + 1, \ldots, m, 1, 2, 3, \ldots, x - 1$$

Cada posición de la tabla establece una secuencia, por lo tanto hay tantas secuencias posibles como posiciones tiene la tabla. Entonces hay exactamente m secuencias posibles. Como $m < m!$ esta técnica no es uniforme.

Ventajas y desventajas La principal ventaja de este método es que resulta sumamente sencillo de implementar. La desventaja reside en que genera un fenómeno conocido como **clustering**, largas zonas del vector ocupadas y otras desocupadas, los datos tienden a distribuirse en forma despareja.

Direccionamiento abierto cuadratico

$$h(k,0) = h(k)$$
$$h(k,i) = (h(k) + c1 \cdot i + c2 \cdot i^2) \, mod \, m$$

Donde $c1$ y $c2$ son constantes. Al igual que en el método lineal hay tantas secuencias de prueba como posiciones tiene la tabla y por lo tanto el método no es uniforme.

Ventajas y desventajas Este método si bien es un tanto mas complejo de implementar también resulta sencillo, igual que el método lineal también genera clustering pero a diferencia del anterior lo hace en forma mas leve.

Direccionamiento abierto con Doble-Hashing

$$h(k,0) = h(k)$$
$$h(k,i) = (h1(k) + h2(k) \cdot i) \, mod \, m$$

Importante: Para estar seguros de que la secuencia de pruebas intenta con todos las posiciones de H debemos tener $h2(k)$ relativamente primo a m (sin divisores en común). Esto es fácil de conseguir asegurando que m sea un número primo. Si m es una potencia de 2 y $h1(k) = k \, mod \, m$ entonces $h2(k)$ siempre debe ser impar.

Para este caso hay mas de m secuencias de prueba pero de todas formas el método no es uniforme.

Análsis Para analizar la eficiencia de las operaciones usando direccionamiento abierto debemos analizar cual esla cantidad esperada de pruebas a realizar.

Suponemos hashing uniforme (todas las m! secuencias posibles son equiprobables). Suponemos que estamos analizando el $Insert(c)$ pero tengamos en cuaneta que el resto de las operaciones son análogas.
$\alpha = \frac{n}{m}$ por lo que necesitamos $\alpha \leq 1$

Teorema: La probabilidad de que por lo menos i pruebas accedan a posiciones ocupadas es menor o igual a α^i.

Demostración

$$P[ipruebas] = P[1] \cdot P[2] \ldots P[i-1]$$
$$P[i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-(i-1)}{m-(i-1)}$$
$$P[i] \leq (\frac{n}{m})^i$$
$$P[i] \leq \alpha^i$$

Teorema: Si $\alpha < 1$ entonces el numero esperado de pruebas en una búsqueda no-exitosa es menor o igual a $\frac{1}{1-\alpha}$

Demostración: En una búsqueda no-exitosa cuando $\alpha < 1$ alguna cantidad de pruebas acceden a posiciones ocupadas mientras que la última accede a una posición vacía o borrada.

Cantidad de pruebas =

1 + E(pruebas que acceden a posiciones ocupadas)

$$1 + \sum_{i=0}^{\infty} i \cdot Pr[exactamente - i - pruebas - acceden - a - posiciones - ocupadas]$$

Por teorema anterior.

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

$$\frac{1}{1-\alpha}$$

Por ejemplo si $\alpha = .5$ entonces el número promedio de pruebas es $\frac{1}{1-.5} = 2$. Si la tabla está llena en un 90% el número de pruebas promedio es 10.

Teorema: Si $\alpha < 1$ entonces el numero esperado de pruebas en una búsqueda exitosa es menor o igual a $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

Demostración Sea k la clave que estamos buscando. Supongamos que k fue la $(i+1$ clave insertada. La cantidad promedio de pruebas para insertar k fue (según teorema anterior):

$$\frac{1}{1 - \frac{i}{m}} = \frac{m}{m-1}$$

Cantidad de pruebas =

$$= \sum_{i=0}^{n-1} P[k(i+1) insertada] \cdot \frac{m}{m-i}$$

$$= \frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i}$$

$$= \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$= \frac{1}{\alpha} \cdot \sum_{j=m-n+1}^{m} \frac{1}{j}$$

$$\leq \frac{1}{\alpha} \cdot \int_{j=m-n}^{m} \frac{1}{x} dx$$

$$= \frac{1}{\alpha} \cdot \ln \frac{m}{m-n}$$

$$= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Si la tabla está llena en un 90% el número promedio de

10.7.3 Funciones de Hashing

No vamos a realizar un estudio de funciones de hashing debido a que no es tema de la materia, basicamente hay dos métodos básicos para construir una función de hashing.

Método de la división

$$h(k) = k \, mod \, m$$

Para que la función sea buena debe ser lo mas uniforme posible, esto ocurre cuando m es un número primo.

Método de la multiplicación

$$h(k) = \lfloor m(k \cdot A \, mod \, 1) \rfloor$$

Donde A es una constante tal que $0 < A < 1$. $kA \, mod \, 1$ es la parte fraccional de kA es decir $kA - \lfloor kA \rfloor$. El valor de m no es tan crítico en este método como en el de la división. La distribución depende mas del valor de A, un valor de A que suele tener buen resultado es $\phi = \frac{\sqrt{5}-1}{2}$. Otra ventaja de este método es que las multiplicaciones suelen costar menos que las divisiones en la mayoría de las arquitecturas conocidas.

10.8 Resumen y puntos clave

Este fue un capítulo realmente extenso cuyo principal objetivo es presentar las herramientas necesarias para analizar la eficiencia de diversas implmentaciones de algunas estructuras de datos y presentaralgunas estructuras nuevas que pueden permitir diseniar mejores algoritmos.

En principio se repasaron ciertas estructuras de datos conocidas como ser pilas, colas, listas, arboles, etc. Tener conocimiento de estas estructuras es fundamental para el disenio de algoritmos.

El analisis amortizado es la herramienta que debe utilizarse para analizar la eficiencia de las primitivas que implementan la funcionalidad de una cierta estructura de datos. Describimos el metodo de agregacion, el contable y el de los potenciales del analisis amorizado.

Los Heaps-Binomiales presentan una implementacion sumamente eficiencia para una estructura generica de Heaps Mergeables, para describir esta estructura hacemos uso de los arboles binomiales que tambien son nuevos.

La estructura Union-Find es necesaria para manejo de estructuras basadas en conjuntos disjuntos, describimos y analizamos diversas implementaciones de esta estructura.

Por ultimo las hash-tables son la opcion habitualmente mas eficiente para implementar estructuras de tipo diccionario.

10.9 Ejercicios

1. Insertar las siguientes claves:

   $$5, 28, 19, 15, 20, 33, 12, 17, 10$$

   En una hash-table con resolución de colisiones por encadenamiento con $m = 9$ y con $h(k) = k \, mod \, m$

2. Insertar las siguientes claves:

   $$10, 22, 31, 4, 15, 28, 17, 88, 59$$

   En una hash-table con $m = 11$ usando direccionamiento abierto usando $h(k) = k \, mod \, m$. Emplear:

   (a) Direccionamiento abierto lineal.

   (b) Direccionamiento abierto cuadratico con $c1 = 1$ y $c2 = 3$

   (c) Doble-hashing con $h2(k) = 1 + (k \, mod \, (m - 1))$

3. La operación Binomial-Heap-Minimum puede funcionar incorrectamente si las claves pueden tomar valor ∞, explicar porque y re-escribir la primitiva para poder manejar esta situación.

4. Mostrar paso a paso la estructura resultante luego de realizar las siguientes operaciones:

```
for i = 1 to 16 do
   Make-Set(x)
end for
for i = 1 to 15 by 2 do
   Union(Xi,Xi+1)
end for
for i = 1 to 13 by 4 do
   Union(Xi,Xi+2)
end for
Union(X1,X5)
Union(X11,X13)
Union(X1,X10)
Find-Set(X2)
Find-Set(X9)
```

Chapter 11

Complejidad

11.1 Introducción

A lo largo del curso hemos mencionado en algunas oportunidades problemas para los cuales no teníamos una solución eficiente, problemas que requerían para ser solucionados algoritmos de orden exponencial o superexponencial. En particular teníamos problemas con algoritmos de tipo $O(c^n), O(n^n), O(n!)$.

En la década del '70 cientos de problemas estaban en esta situación de incertidumbre. La teoría de los problemas NP-Completos desarrollada por Stephen Cook y Richard Karp aportó las herramientas necesarias para poder comprender a este tipo de problemas y para poder clasificar a un problema de acuerdo con su dificultad para resolverlo algoritmicamente.

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f(n) = n!$
10	0.01 μs	0.1 μs	1 μs	3.63 ms
20	0.02 μs	0.4 μs	1 ms	77.1 years
30	0.03 μs	0.9 μs	1 sec	8.4×10^{15} years
40	0.04 μs	1.6 μs	18.3 min	
50	0.05 μs	2.5 μs	13 days	
100	0.1 μs	10 μs	4×10^{13} years	
1,000	1.00 μs	1 ms		

Es importante comprender cual es el verdadero problema de los algoritmos de orden exponencial, uno podría suponer que el problema es que dichos algoritmos son ineficientes y que el escollo puede salvarse usando hardware mas poderoso, este concepto es equivocado. En el gráfico podemos ver el tiempo necesario para resolver un problema de acuerdo al orden del algoritmo utilizado en una computadora de primera línea. Como podemos ver los algoritmos que son exponenciales necesitan varios anios para resolver un problema en el cual $n > 100$, si el orden es $n!$ para $n > 20$ necesitamos mas de un siglo de cálculos!. El problema

de los algoritmos de este tipo es que para un cierto n no muy grande el problema sencillamente **no se puede resolver**.

11.1.1 Reducciones

Supongamos que tenemos un problema denominado *problema X* y que el problema X puede resolverse de la siguiente forma:

Problema-X(Q)

1. Convertir Q en R que es una instancia del problema Y. $R = f(Q)$.
2. Usar una rutina que resuelve el Problema-Y de la forma $Z = Problema - Y(R)$
3. Convertir Z en W aplicando la inversa de f. $W = f^{-1}(Z)$.
4. Devolver W como la solución de Problema-X(Q).

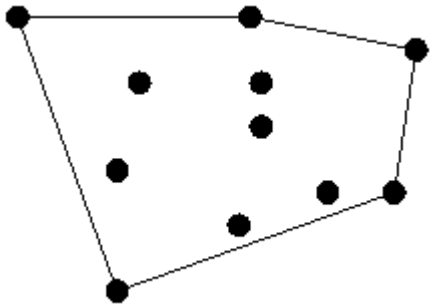
La transformacion de instancias de un tipo de problema en instancias de otro tipo de problema de forma tal que las soluciones se preservan se denomina **Reducción**. Las reducciones fueron introducidas por Richard Karp y son la herramienta fundamental para manejarnos dentro del mundo de los problemas NP-completos.

Si nuestra reducción traduce Q en R en $O(P(n))$ entonces.

1. Si Problema-Y(R) es $O(P'(n))$ entonces el Problema-X puede resolverse en $O(P(n) + P'(n))$
2. Si sabemos que $\Omega(P'(n))$ es una cota inferior para resolver el Problema-X, entonces $\Omega(P'(n) - P(n))$ tiene que ser una cota inferior del Problema-Y.

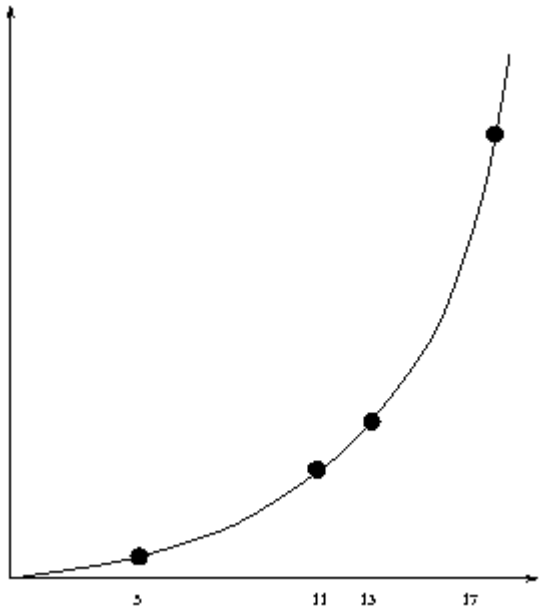
La segunda propiedad es la que vamos a utilizar para la tarea nada grata de demostrar que un problema es demasiado complejo como para poder resolverlo.

Ejemplo Un problema interesante de la geometría espacial es dado un conjunto de puntos en el plano encontrar el polígono convexo mas extenso que es frontera de dichos puntos. En la figura vemos una solución a un ejemplo.



Supongamos que disponemos de una rutina **Convex-Hull(V)** que dado un vector V de n puntos calcula el polígono convexo frontera y lo devuelve en un vector. Sorprendentemente aplicando una reducción podemos utilizar este problema para diseniar un curioso algoritmo de sort.

Dado un vector W de n numeros que queremos ordenar lo convertimos en un vector de puntos en el plano donde $W[i].x = W[i], W[i].y = W[i]^2$, al realizar esta transformación los puntos forman una parábola.



Como la parábola es convexa cada punto de la parábola forma parte del polígono convexo frontera, ademas como los puntos vecinos en el polígono tienen valores de x vecinos el polígono covexo externo devuelve los puntos ordenados por sus coordenadas x (es decir que ordena los números originales).

Algorithm 63 Sort-CX(V,n). Ordena el vector V.

for $i = 1$ to n **do**
  $W[i].x \leftarrow V[i]$
  $W[i].y \leftarrow V[i]^2$
end for
$Convex - Hull(W, n)$
for $i = 1$ to n **do**
  $V[i] \leftarrow W[i].x$
end for

Análisis Crear el vector W y luego copiarlo en V se hace en $O(n)$. Recordemos que la cota inferior de un algoritmo de sort era $\Omega(n \log n)$. Si pudieramos hacer el procedimiento $Convex - Hull$ en menos de $\Omega(n \log n)$ entonces podríamos ordenar en menos de $\Omega(n \log n)$ ya que un problema se puede reducir al otro!.

Por lo tanto el algoritmo Convex-Hull aunque no sepamos hacerlo debe ser $\Omega(n \log n)$.

Como podemos ver una reducción es una herramienta muy poderosa que nos permite demostrar propiedades de algunos problemas aun cuando no sabemos como resolverlos.

11.2 Fundamentos

Qué es un problema *Un problema es una pregunta genérica con parametros para los datos de entrada y condiciones que establecen como debe ser una solución satisfactoria a dicho problema.*

Instancia de un problema *Una instancia de un problema es un problema aplicado a un cierto conjunto de parámetros.*

Problemas de decisión *Un problema de decisión es un problema para el cual las respuestas posibles son Si o No.*

11.2.1 El problema del viajante

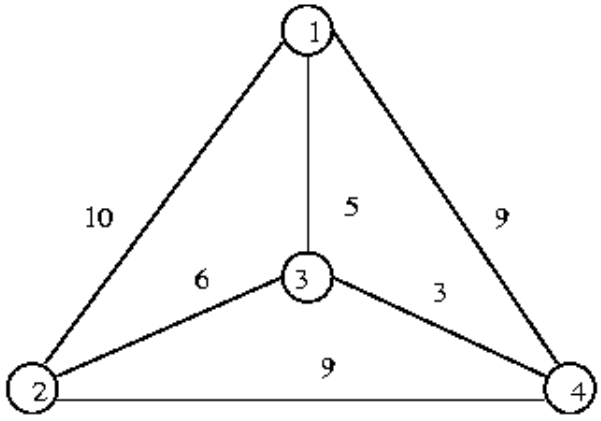
Problema genérico

Dado un grafo pesado $G = (E, V)$, encontrar un camino de distancia mínima que partiendo de un vértice dado visite todos los demas vértices una sola vez y que pase una sola vez por cada arista y luego regrese al origen sin formar sub-tours.

$$min\left\{\sum_{i=1}^{n-1} d[vi, vi+1] + d[vn, v1]\right\}$$

Una instancia del problema

d[1,2]=10; d[1,3]=5; d[1,4]=9; d[2,3]=6; d[2,4]=9; d[3,4]=3



Solución

$$\{v1, v2, v3, v4\}$$

Costo=27.

Problema de decisión

Dado un grafo $G = (E, V)$ y un entero k, indicar si existe una solución al problema del viajante tal que el costo sea $\leq k$.

Usando búsqueda binaria y el problema de decisión del viajante podemos encontrar la solución óptima al problema genérico del viajante.

Muy importante A partir de ahora vamos a concentrarnos en problemas de decisión, el problema genérico puede resolverse en general en tiempo polinómico si es posible resolver el problema de decisión en tiempo polinómico.

11.3 Complejidad

Necesitamos alguna forma de diferenciar aquellos problemas que se pueden resolver de aquellos que no debido a que no existe un algoritmo eficiente para solucionarlos. Esta distinción la vamos a hacer considerando aquellos problemas que se pueden resolver en tiempo polinómico.

Hemos medido el costo en tiempo de un algoritmo expresando el orden del mismo en función del tamanio de los datos de entrada n. Un algoritmo es de orden polinómico si su orden es $O(n^k)$[1] donde k es constante y no depende de n. Un problema es solucionable en tiempo polinómico si existe un algoritmo de orden polinómico que resuelve el problema.

Algunas funciones no parecen polinómicas pero lo son, por ejemplo $O(n \log n)$ no parece un polinomio pero esta acotada superiormente por $O(n^2)$ y por lo tanto puede considerarse polinómica.

11.3.1 La clase P de problemas

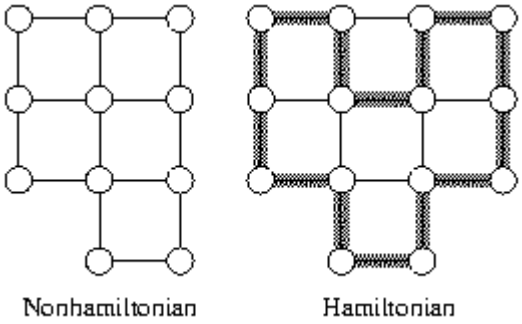
La clase P de problemas esta integrada por todos los problemas que son **solucionables** mediante un algoritmo de orden polinómico.

11.3.2 La éclase NP de problemas

Algunos problemas no se pueden resolver en tiempo polinómico pero dada una solución candidata es factible verificar si la solución es óptima en tiempo polinómico. Estos problemas forman la clase NP de problemas.

Ejemplo Un ejemplo de problema NP es el problema del ciclo Hamiltoniano no dirigido. Dado un grafo $G = (E, V)$ no dirigido encontrar un ciclo que visite cada vértice de G exactamente una vez.

[1] A veces una función de la forma n^m es ambigua pues no se sabe si es exponencial o polinómica dependiendo de que varía en función de que. Una notación que permite resolver este problema es $\lambda n.n^m$ (polinómica) o $\lambda m.n^m$ (exponencial) donde λ indica cual es el factor que varía.



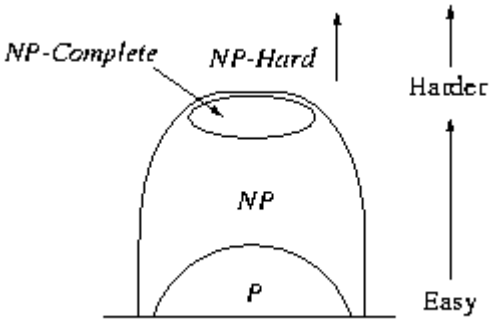
Un aspecto interesante de este problema es que si el grafo tiene un ciclo hamiltoniano es fácil verificarlo. Por ejemplo dado el ciclo $\langle v3, v7, v1, \ldots, v13\rangle$. Podemos inspeccionar el grafo y verificar que sea un ciclo y que ademas visite todos los vértices exactamente una vez. Por lo tanto aunque no sepamos como resolver el problema del ciclo Hamiltoniano si disponemos de un metodo eficiente para verificar una posible solución del problema. La solución se denomina **certificado**. Los problemas de tipo NP son verificables en tiempo polinómico, por lo tanto el problema del ciclo Hamiltoniano pertenece a la clase NP.

Observaciones En primer lugar es evidente que $P \subseteq NP$ ya que si un problema es solucionable en tiempo polinómico entonces tambien es verificable en tiempo polinómico. Lo que no se sabe es si $P = NP$, parece poco sensato pensar que las clases son iguales, es decir que poder verificar un problema en tiempo polinómico no necesariamente implica poder solucionarlo en tiempo polinómico, es aceptado que $P \neq NP$ pero nadie ha podido demostrarlo hasta ahora, este es uno de los puntos abiertos mas importantes en el mundo de la computación.

$$\boxed{P =? NP}$$

11.3.3 La clase de problemas NP-Completos

Los problemas $NP - Completos$ son los problemas mas difíciles que vamos a enfrentar en este curso, existen sin embargo problemas aun mas difíciles que los problemas NPC que se denominan problemas $NP - Hard$.



Es válido decir que $NPC \subseteq NP$ es decir que los problemas NP-Completos se pueden verificar en tiempo polinómico.

Los problemas NP-Completos son problemas NP que cumplen la propiedad de que si existiera una solución polinómica a uno de ellos entonces todos los demás serían solucionables en tiempo polinómico.

11.3.4 La clase de problemas No-NP

Los problemas $No - NP$ son problemas que ni siquiera pueden verificarse en tiempo polinómico. Por ejemplo: Cuantos tours de longitud menor o igual que K existen en un grafo G en el problema del viajante. Dado que hay un número exponencial de tours no podemos contarlos por lo que no podríamos verificar la solución en tiempo polinómico. Este es un problema que no petenece a la clase NP. En general este tipo de problemas no son de mucha utilidad.

Una historia En 1903 T.E.Bell y otros matemáaticos tenian como gran interrogante si el número $2^{67} - 1$ era o no primo. En la reunion de la sociedad matemática americana en Nueva York F.Cole presentó un paper con el modesto título "Sobre la factorización de números muy grandes".

Cuando le llegó el turno de exponer su trabajo F.Cole -que siempre fue una persona muy callada- caminó hacia el frente y con sumo cuidado procedió a calcular 2^{67}, luego le restó uno. Se movió hacia el otro extremo del pizarrón y escribió $193707721x761838257287$ y usando el método que todos aprendimos en la primaria hizo la cuenta, los dos números coincidieron sin decir una sola palabra volvió a su asiento.

Por primera y unica vez la sociedad matemática americana aplaudió extensamente al autor de un paper sin siquiera leerlo. A Cole se le hizo una sola pregunta: cuanto tiempo le había llevado la factorización del número a lo cual respondió: "150 domingos".

Como puede verse hay problemas que son extremadamente simples de verificar pero muy dificiles de resolver. [2]

11.4 Reducciones

Las reducciones son útiles para probar que un problema no puede resolverse en tiempo polinómico. Supongamos que queremos probar que el problema B no puede resolverse en tiempo polinómico y que disponemos de un problema A que *sabemos* que no se puede resover en tiempo polinómico. Si suponemos que hay una solución polinómica para B y mediante una reducción podemos llegar a que con esa solución podemos resolver A en tiempo polinómico llegaríamos a una contradiccion y entonces podriamos afirmar que B no puede resolverse en tiempo polinómico.

La clave para demostrar que un problema B no puede resolverse en orden polinómico esta en poder reducir B a un problema A del cual sabemos no existe una solución polinómica.

11.4.1 Ejemplo I

Es un hecho que el problema del ciclo Hamiltoniano en un grafo no-dirigido es Np-Completo. Es decir que no se conoce algoritmo que pueda resolverlo en tiempo polinómico y los expertos ampliamente están convencidos de que tal algoritmo no puede eixstir.

Tratemos de ver que ocurre con el problema de encontrar un ciclo Hamiltoniano en un grafo dirigido. Luego de pensar durante mucho tiempo podemos llegar a la sospecha de que no existe un algoritmo polinómico para este problema, entonces podemos tratar de demostralo aplicando una reducción.

Dado un grafo no-dirigido $G = (E, V)$ creamos un grafo dirigido G' reemplazando cada arista del grafo no-dirigido por un par de aristas en el grafo dirigido de forma tal que la arista no-dirigida (u, v) se convierte en las aristas dirigidas $(u, v) y (v, u)$. Ahora cada camino simple en G es tambien un camino simple en G' y viceversa. Por lo tanto G tiene un ciclo Hamiltoniano solo si G' tambien lo tiene. Si pudiéramos resolver el problema del ciclo hamiltoniano en G' podríamos resolver el problema del ciclo Hamiltoniano en G ya que un camino en G' es tambien un camino válido en G. Por lo tanto el problema del ciclo Hamiltoniano dirigido también es NP-Completo.

[2] Curiosamente no se ha podido demostrar que la factorización sea un problema NP-Completo, la clasificación de este problema es uno de los misterios que aun no se han resuelto y del cual curiosamente depende la seguridad de varios organismos ya que las funciones criptográficas suelen estar basadas en la factorización de números.

Notemos que no hemos resuelto ninguno de los dos problemas, solo mostramos como podemos convertir la solución de un problema $P1$ en una solución válida para otro problema $P2$. A esto se lo denomina reducir $P1$ a $P2$.

11.4.2 Ejemplo II.

Consideremos los siguientes problemas.

El problema de los 3 colores (3COL) Dado un grafo G determinar si cada uno de sus vértices puede ser pintado con uno de 3 (tres) colores distintos de forma tal que dos vértices adyacentes no esten pintados del mismo color.

Cubrimiento con un Clique (CC) Dado un grafo G y un entero k determinar si los vértices de G pueden ser particionados en subconjuntos $V1, V2, \ldots, Vk$ de forma tal que $\cup_i Vi = V$ y que cada V_i es un clique.

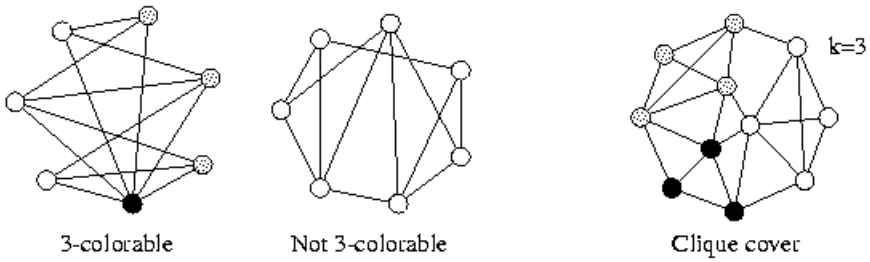
Recordemos que un clique es un subconjunto de vértices de forma tal que cada par de vértices del subconjunto son adyacentes entre si.

(3COL) es un problema NP-Completo conocido, veamos si usando este problema podemos probar que (CC) tambien es NP-Completo.

Demostración Tenemos que probar dos cosas, en primer lugar debemos probar que (CC) pertenece a NP, esto es sencillo ya que dados k conjuntos verificar que la union de los mismos forme el grafo y que cada uno de ellos es un *clique* puede hacerse en tiempo polinómico.

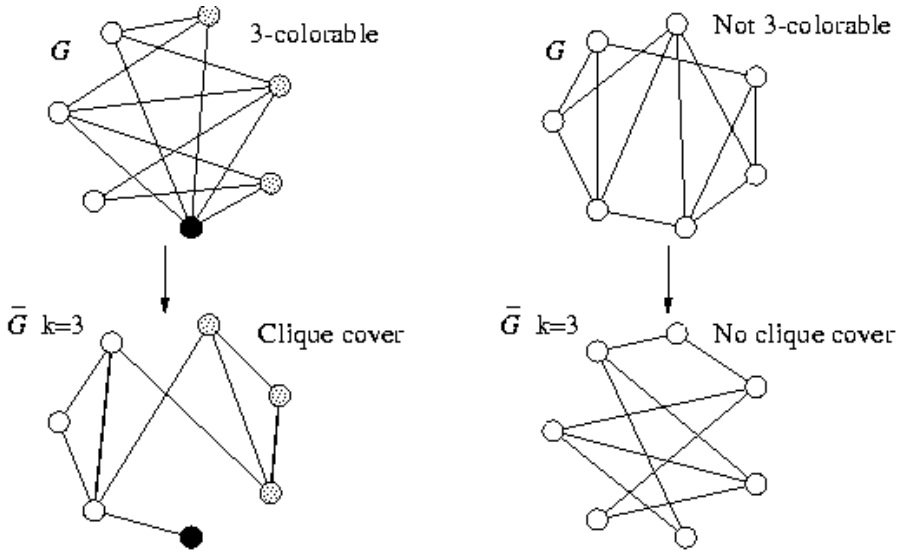
En segundo lugar debemos probar que un problema NP-Completo conocido es reducible en forma polinómica a (CC). En nuestro caso el problema elegido va a ser (3COL).

Asumimos que existe una subrutina CliqueCover(G,k) polinómica que determina si un grafo tiene un cubrimiento clique de tamanio k. Necesitamos poder usar esta rutina para resolver (3COL).



Veamos en que aspectos estos problemas son similares. Ambos problemas dividen los vértices en grupos, en (CC) para que dos vértices estén en un grupo deben ser adyacentes. En (3COL) para que dos vértices estén en el mismo grupo no deben ser adyacentes. En cierta forma los problemas son similares pero la noción de adyacencia está invertida.

Recordemos que G^c es el grafo complemento de G es decir que tiene las aristas que le faltan a G y no tiene las aristas que tenia G. Nuestra observación es que G es (3COL) si y solo si G^c tiene un cubrimiento clique con $k = 3$. La demostración no es difícil y queda como ejercicio.



Usando esta observación podemos reducir (3COL) a (CC) de la siguiente manera. Dado un grafo G obtenemos su complemento G^c y luego invocamos a Clique-Cover(G^c,3). Luego cada uno de los subconjuntos que devuelve $Clique - Cover$ corresponde a un color en el grafo original y el problema (3COL) queda resuelto.

Si podemos resolver (CC) en tiempo polinómico entonces podríamos resolver (3COL) en tiempo polinómico, pero como sabemos que (3COL) es NP-Completo entonces (CC) tambien es NP-Completo.

Es importante notar que la reducción funciona siempre mostrando como la solución del problema que se sospecha NP-Completo puede usarse para encontar una solución al problema que se sabe es NP-Completo. Esto es importante.

Importante: Siempre se debe probar que el problema NP-Completo se puede reducir al problema que sospechamos es NP-Completo.

Notación Cuando realizamos una reducción utilizaremos la siguiente notacion:

$$X \propto Y$$

Donde: X es un problema que se sabe es NP-Completo, Y es el problema que queremos probar que es NP-Completo, los pasos a seguir son:

1. Demostrar que Y pertenece a NP, es decir que dada una solución se la puede verificar en tiempo polinómico.
2. Suponer que el problema Y se puede resolver en tiempo polinómico.
3. A partir de una instancia del problema X aplicar una transformación que genere una entrada válida para el problema Y
4. Demostrar que una vez solucionado Y la solución permite encontrar una solución a la instancia del problema X planteada.

11.5 Algunos problemas NP-Completos

11.5.1 SAT

SAT (satisfiability[3]) es un problema bastante sencillo de plantear y sumamente importante para la teoría de los problemas NP-Completos, el problema es el siguiente: dada una expresión lógica de la forma

$$(v11 \wedge -v12 \wedge \ldots v1n) \vee (v21 \wedge v22 \wedge \ldots v2o) \vee \ldots \vee (vm1 \wedge \ldots vmz)$$

Que podemos anotar también como

$$(v11, v12, \ldots v1n)(v21, v22, \ldots v2o) \ldots (vm1, vm2, \ldots vmz)$$

Encontrar valores para todas las variables booleanas vij de forma tal que la expresión sea verdadera.

Ejemplo 1

$$C = (v1 \wedge -v2) \vee (-v1 \wedge v2)$$

La expresión es verdadera cuando $v1 = v2 = true$

Ejemplo 2

$$C = (v1 \wedge v2) \vee (v1 \wedge -v2) \vee (-v1)$$

Por mas que probemos y probemos no existen valores para v1 y v2 de forma tal que la expresión sea verdadera.

[3]Esta es una palabra con traducción imposible, "satisfactibilidad?"

Claramente SAT pertenece a NP, ya que dada una expresión y un conjunto de valores para sus variables podemos chequear en tiempo polinómico si la expresión evalua como verdadera, en la década del '70 Cook se encargo de demostrar que SAT era un problema NP-Completo, volveremos sobre esta demostración mas adelante, por ahora supongamos que SAT es NP-Completo y como veremos a continuación a partir de SAT y aplicando reducciones es sencillo demostrar que muchos otros problemas son NP-Completos.

11.5.2 3-SAT (SAT $\propto$ 3-SAT)

3-SAT es una variación de SAT en la cual cada "ororia" tiene exactamente tres términos, de la forma:

$$(v11 \wedge v12 \wedge v13) \vee (v21 \wedge v22 \wedge v23) \vee \ldots \vee (vn1 \wedge vn2 \wedge vn3)$$

Claramente 3-SAT puede verificarse en tiempo polinómico y por lo tanto pertenece a NP, pero al ser un problema mas restringido que SAT podríamos pensar que 3-SAT no es NP-Completo (tal vez sea la cantidad ilimitada de términos en cada ororia la fuente de la dificultad de SAT).

Teorema: 3-SAT es NP-Completo La demostración se hace mediante la reducción $SAT \propto 3 - SAT$

Supongamos que existe un solución polinómica para 3-SAT, y tomemos un problema genérico SAT, donde la $i$ésima cláusula de la expresión tiene exactamente k literales.

$$(vi1, vi2, \ldots vik)$$

El objetivo es transformar cada cláusula de SAT en un conjunto de cláusulas de 3-SAT de forma tal que si todas las variables de una cláusula de SAT son falsas no exista forma de lograr que las cláusulas de 3-SAT sean verdaderas. De esta forma lograríamos que ambos problemas fuesen equivalentes ya que una solución válida de 3-SAT implicaría una solución válida de SAT.

- Si $k = 1$ entonces $Ci = (z1)$, creamos dos nuevas variables $v1, v2$ y tres nuevas cláusulas:

  $$(v1, v2, z1)(v1, -v2, z1)(-v1, v2, z1)(v1, v2, z1)$$

  Observemos que para que las cuatro cláusulas sean verdaderas z debe ser verdadero. Si $z1$ es falso entonces alguna de las cláusulas seguro que es falsa.

- Si $k = 2$ $Ci = (z1, z2)$ creamos una nueva variable $v1$ y dos nuevas cláusulas:

  $$(v1, z1, z2)(-v1, z1, z2)$$

  Observemos que si $z1$ es falso no hay forma de asignar $v1$ y $v2$ para que la expresión sea verdadera.

- Si $k = 3$ $Ci = (z1, z2, z3)$ no realizamos ninguna modificación
- Si $k > 3$ $Ci = (z1, z2, \ldots, zn)$ creamos una cadena de la forma:

  $$(v1, z1, v1)(v1, z2, -v1)(v1, z3, -v1), \ldots (v1, zn-1, -v1)(-v1, zn, -v1)$$

  De esta forma si todas las variables son falsas no hay forma de que la expresión de verdadero, pero si una sola de ellas es verdadero entonces es posible asegurar que todas las demas sean verdadero. Esto puede resultar difícil de ver en forma inmediata pero vale la pena dedicar un tiempito a observarlo.

Una vez aplicada esta transformación si podemos resolver el problema 3-SAT entonces también resolvimos el problema SAT ya que ambos problemas son equivalentes, como SAT es un problema NP-Completo conocido resulta que 3-SAT debe ser entonces NP-Completo.

11.5.3 Programación Lineal Entera (SAT $\propto$ PLE)

Un problema de programación lineal entera esta dado por un conjunto de variables que deben ser enteras y no negativas (condicion de no-negatividad), un conjunto de inecuaciones lineales que involucran a dichas variables y una función también lineal que se debe maximizar o minimizar.

Ejemplo

$$x1 + x2 \leq 5$$

$$x2 - x3 \geq x2$$

$$f(x) = x1 + 2x2 - 3x3(Max)$$

La conversión a un problema de decisión se hace fijando una constante para la función a optimizar. Por ejemplo:

$$x1 + 2x2 - 3x3 \geq 100$$

Teorema: Un problema de programación lineal entera es NP-Completo
En primer lugar observemos que PLE pertenece a NP ya que dado un conjunto de valores para las variables verificar que estas satisfacen las inecuaciones resulta simple. Para demostrar que PLE es NP-Completo vamos a aplicar una reducción usando como base el problema SAT que como sabemos es NP-Completo.

Suponemos que existe una solución polinomica a PLE, entonces dada una expresión de SAT de la forma:

$$C = (v11, v12, \ldots v1a)(v21, v22, \ldots, v2b) \ldots (vn1, vn2, \ldots, vnz)$$

Podemos escribir a partir de la expresión un problema de programación lineal de la forma:

$$vij \leq 1$$

$$v11 + v12 + \ldots + v1a \geq 1$$

$$v21 + v22 + \ldots + v2b \geq 1$$

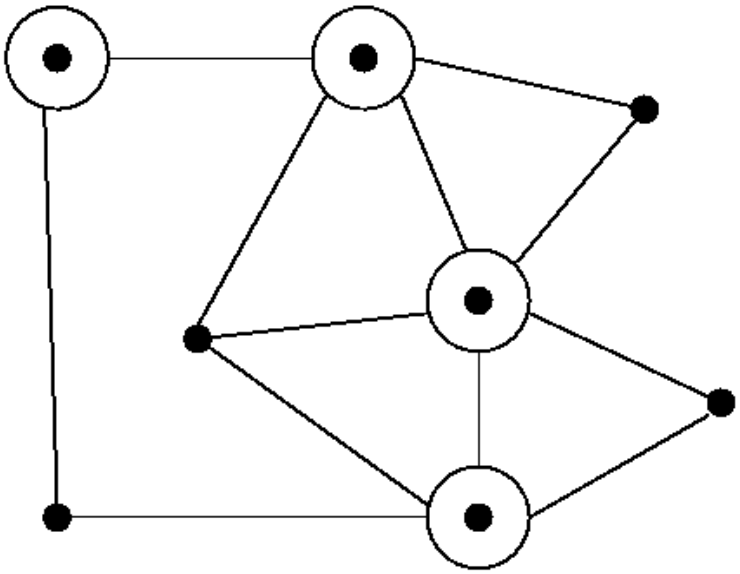
etc

La función a maximizar o minimizar no tiene importancia, vale por ejemplo maximizar o minimizar cualquiera de las variables. Observemos que si resolvemos el problema de PLE los valores que quedan asignados a las variables satisfacen la expresion SAT, ya que cada inecuación se cumple si y solo si al menos una de las variables de PLE es mayor que cero lo cual implica que al menos una de las variables de la cláusula de SAT es verdadera. Como deben cumplirse todas las inecuaciones tenemos que deben cumplirse todas las cláusulas de SAT. Por lo tanto todo problema de PLE es NP-Completo.

Precaución Aquellos que esten familiarizados con la programación lineal entera podrían deducir en este momento que aquellos problemas que se pueden expresar como un problema de PLE son NP-Completos, esto no es cierto y es un mal-uso del mecanismo de reducciones, el uso de PLE para probar que un problema es NP-Completo implicaría demostrar que solucionando un determinado problema "X" en tiempo polinómico podemos resolver "Cualquier" problema de PLE, por lo tanto "X" debe ser NP-Completo. Hay problemas (muchísimos) que se pueden resolver en tiempo polinómico y también pueden escribirse como un problema de PLE (obviamente usar PLE para estos problemas es ineficiente pero no imposible).

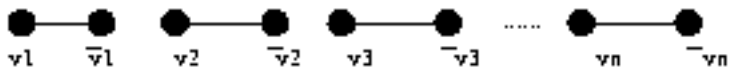
11.5.4 Vertex Cover (3-SAT $\propto$ VC)

Dado un grafo no dirigido $G = (V, E)$ indicar si existe un conjunto de a lo sumo k vértices de forma tal que cada arista del grafo parte o va hacia uno de los vértices del conjunto. En la figura podemos ver un ejemplo.



Teorema: VC es NP-Completo En primer lugar VC es NP ya que dada una posible solución podemos verificar que toda arista del grafo involucre a algun vértice del conjunto en forma sencilla en tiempo polinómico. Para demostrar que es NP-Completo vamos a aplicar una reducción a partir de 3-SAT que ya demostramos es NP-Completo. Suponemos pues que existe una solución polinomica para VC.

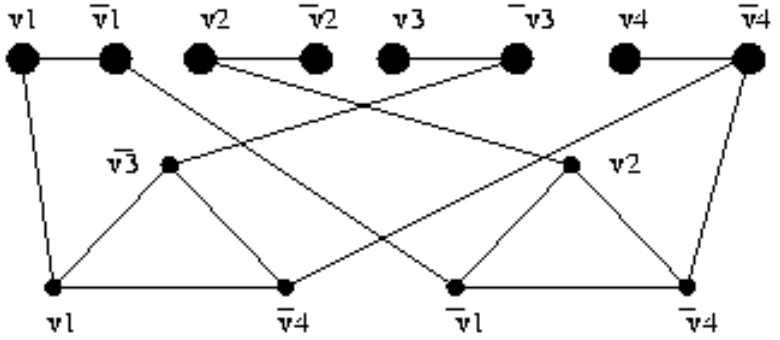
Para convertir una instancia de 3-SAT en un problema de VC partimos de un problema 3-SAT con N variables y C cláusulas y construimos un grafo con $2N+3C$ vértices.Para cada variable creamos dos vértices unidos por una arista.



Para cubrir las aristas por lo menos n vértices deben pertenecer al conjunto, uno por cada par. Para cada cláusula creamos tres nuevos vértices, uno por cada literal en cada cláusula y los conectamos en un triángulo.

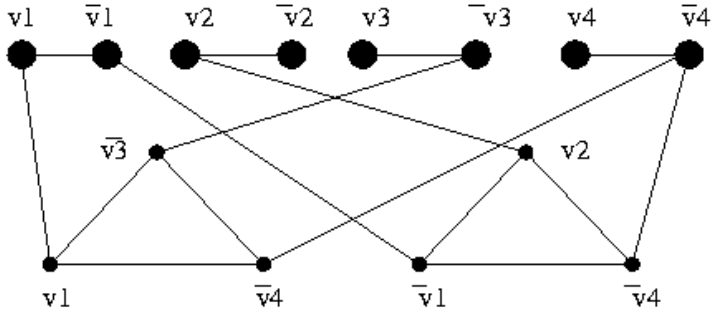
Por lo menos dos vértices por triángulo deben pertenecer al conjunto para cubrir las aristas que son lados del triángulo lo cual implica $2C$ vértices.

Finalmente conectamos cada literal en la estructura plana a los vértices de los triángulos que tienen el mismo literal.



Aseguramos que el grafo resultante tiene una solución a VC de tamanio N+2C si y solo si la expresion 3-SAT puede evaluarse como verdadera. De nuestro análisis anterior al construir el grafo toda solución a VC debe tener al menos N+2C vértices.

Requeriría un análisis mas riguroso pero si nos tomamos algunos minutos analizando la transformación que hicimos de 3-SAT se nos permite asegurar que si encontramos una solución para VC en el grafo construido a partir de dicha solución encontramos una solución a 3-SAT. Por lo tanto VC es NP-Completo.

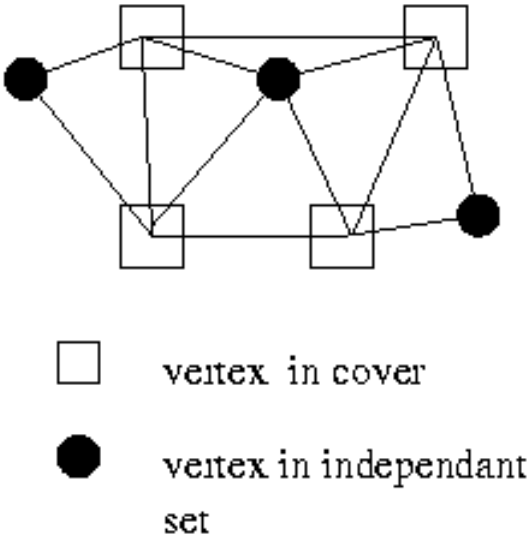


Esta reducción fue bastante ingeniosa, como podemos ver hay que tener un criterio muy amplio y gran imaginación para encontrar la reducción apropiada para demostrar que un problema es NP-Completo.

11.5.5 Conjunto Independiante (VC $\propto$ IS)

El problema del conjunto independiente consiste en encontrar un conjunto de vértices (de tamanio k) de un grafo de forma tal que ninguna arista del grafo conecte dos vértices de dicho conjunto.

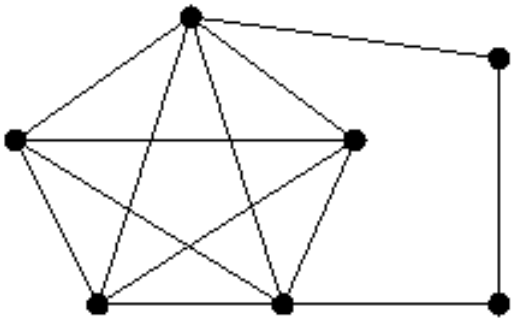
La reducción es muy sencilla ya que si calculamos el VC de un grafo los vértices restantes (los que no pertenecen a VC) forman un conjunto independiente ya que si existiera una arista entre dos de estos vértices los restantes no podrían cubrir todas las aristas y no serían un VC.



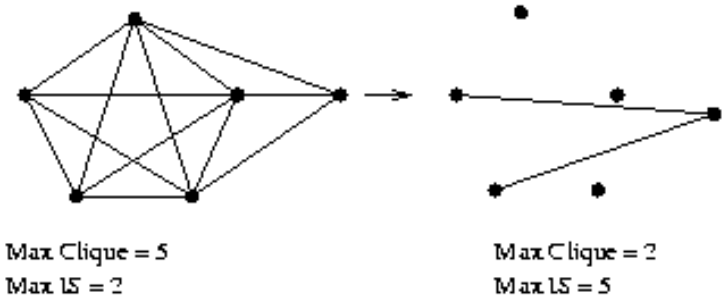
Por lo tanto VC e IS son problemas equivalentes y como VC es NP-Completo IS tambien debe serlo.

11.5.6 Clique (IS $\propto$ Clique)

El problema del Clique consiste en encontra un subconjunto de K vértices de un grafo de forma tal que dicho subconjunto de vértices posee todas las aristas posibles (es fuertemente conexo).



La ilustración muestra un grafo con un clique de tamanio 5. Probar que el problema del clique es Np-Completo es sencillo ya que si calculamos el conjunto independiente de tamanio x en un grafo de v vértices resulta que el grafo complemento del anterior tiene un clique de tamanio $v - x$. Por lo tanto el problema del clique es equivalente al problema del conjunto independientey debe ser NP-Completo.



11.5.7 Partición Entera (VC $\propto$ IP)

Dado un conjunto de enteros S y un entero T determinar si existe un subconjunto de S cuya suma es exactamente igual a T.

Ejemplo $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ y $T = 3754$.

Respuesta: $1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = T$.

Teorema: IP es Np-Completo Esta reducción a partir de VC es bastante extensa y por eso no la incluimos en el apunte, si alguien la necesita esta disponible!.

11.6 Técnicas paéra demostrar que un problema es NP-Completo

11.6.1 Restricción

Mostrar que un caso en particular del problema en cuestión es NP-Completo. Por ejemplo encontrar un camino de longitud k en un grafo es Np-Completo ya que el problema del camino Hamiltoniano es un caso particular (k=cantidad de vértices), por lo tanto el problema es Np-Completo.

11.6.2 Reemplazo local

Realizar cambios locales a la estructura del problema. Un ejemplo de esta tecnica es la que utilizamos en (SAT $\propto$ 3-SAT).

11.6.3 Recomendaciones

- Al realizar una reducción $X \propto Y$ buscar que X sea tan simple (restringido) como sea posible.

  Nunca usar el problema del viajante genérico como problema fuente, en su lugar usar el problema del ciclo hamiltoniano donde todos los pesos son 1 o ∞ o mejor aun usar el problema del camino hamiltoniano, o aun mejor todavía usar el problema del camino hamiltoniano en grafos planares dirigidos donde cada vértice tiene grado 3, etc. Todos estos problemas son NP-Completos y pueden usarse para hacer una reducción, entre problemas equivalentes elegir siempre al mas restringido, de esta forma la reducción será menos trabajosa.

- Hacer que el problema destino Y en $X \propto Y$ sea lo mas genérico posible (cuanto mas difícil mas fácil probar que es NP-Completo).

- Seleccionar el problema fuente correcto. Elegir el problema NP-Completo conocido para hacer la reducción es sumamente importante y es la causa fundamental de problemas en demostrar que un problema es NP-Completo, en general lo mas recomendable es contar con una base de problemas NP-Completos quese conocen en gran profundidad (se conocen los problemas y sus variantes y el grado de dificultad de las mismas). Una buena base podria formarse con los siguientes problemas:

- 3-SAT. Cuando no parece encontarse ningún problema apropiado para aplicar una reducción es recomendable volver al problema origen.
- Partición Entera. La unica opción para problemas cuya dificultad parece residir en la necesidad de usar números grandes.
- VC. Para problemas sobre grafos cuya dificultad reside en la **seleccion** de un conjunto de vértices o aristas.
- Camino Hamiltoniano.Para problemas de grafos cuya dificultad reside en establecer un ordenamiento de vértices o aristas. Si se trata de problemas de ruteo o scheduling este suele ser el mejor problema fuente.

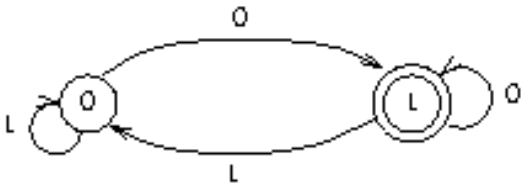
11.7 Teoría formal del los problemas NP-Completos

La teoría de los problemas Np-Completos esta basada en los lenguajes formales y las máquinas de Turing, por lo que debemos trabajar en un nivel mas abstracto del habitual.

Para un alfabeto de simbolos $0, 1, \&$ podemos formar un conjunto infinito de strings o palabras: $'\&10', '1011\&10', '1', etc.$. Un subconjunto de todos los strings posibles de un alfabeto es un lenguaje formal.

La teoría de los lenguajes formales se encarga de estudiar cuan poderosa debe ser una maquina para reconocer si un string pertenece a un lenguaje en particular.

Por ejemplo podemos testear si un string es la representación binaria de un número par, esto se puede hacer con un autómata simple.



Observemos que los problemas de decisión pueden ser pensado como un problema de reconocimiento de un lenguaje formal. Las instancias de los problemas son codificadas como strings y los strings son incluidos en el lenguaje solamente si la respuesta al problema es "Si".

Para reconocer estos lenguajes es necesaria una máquina de Turing. Cualquier algoritmo que resuelva el problema es quivalente a la maquina de turing que reconoce si un determinado string pertenece a un lenguaje. Esto nos va a servir para establecer un marco formal que nos permita precisar que queremos decir cuando afirmamos que un problema no puede o si puede ser resuelto en tiempo polinómico.

11.7.1 Máquinas de Turing No-Determinísticas

Supongamos que extendemos la maquina de Turing agregándole un "módulo adivinador" que en tiempo polinómico construye una posible solución a un determinado problema. Para convencernos de que realmente es una solución podemos correr otro programa que lo verifique. A esta máquina extendida la llamamos máquina de Turing No-Determinística.

Por ejemplo para el problema del viajante el módulo adivinador genera una permutación de los vértices y luego podemos verificar que formen un camino y que la suma de los pesos sea menor que k.

La clase de lenguajes que podemos reconocer en tiempo polinómico en función del tamanio del string en una máquina de turing determinística (sin modulo adivinador) se denomina "P".

La clase de lenguajes que podemos reconocer en tiempo polinómico en función de la longitud del string en una maquina de Turing no-determinística se denomina "NP".

Es claro que $P \in NP$ ya que cualquier programa que corra en una máquina de Turing determinística puede correr en una máquina no-determinística simplemente ignorando al modulo adivinador.

Un problema que pertenece a NP para el cual un algoritmo polinómico implicaría que todos los lenguajes de NP pertenecen a P se denomina $NP - Completo$.

11.7.2 El teorema de Cook

Teorema: SAT es NP-Completo. Esta demostración que afortunadamente no vamos a incluir fue realizada por Cook en la década del '70 basándose en reducir a todas las máquinas de Turing a SAT, sabiendo la definición de un lenguaje NPC, Cook demostró que si existiera una solución polinómica para SAT entonces sería posible verificar en una maáquina de Turing determinística un lenguaje NPC, como sabemos que tal cosa no es cierta se demuestra que SAT es NP-Completo. La demostración es de suma importancia ya que a partir de

SAT y aplicando reducciones se ha demostrado que cientos de problemas son NP-Completos.

11.8 Resumen y puntos clave

Hemos introducido la noción de complejidad de un problema definiendo las clases P, NP y NPC de problemas. Observamos que para algunos problemas que parecen demasiado difíciles a veces lo único que podemos hacer es demostrar que son NP-Completos. Una reducción es una herramienta que permite demostrar que un problema es NP-Completo. Por último se vio en forma muy elemental la base de la teoría de complejidad que se basa en la teoría de los lenguajes formales y la máquina de Turing.

11.9 Ejercicios

1. Dado un vector de 3 elementos escribir un problema de PLE que ordene el vector (que devuelva en variables enteras $I1, I2, I3$ la posición del elemento de cada vector. Por ejemplo si $V = 13, 1, 6$ debería obtenerse $I1 = 3, I2 = 1, I3 = 2$

2. Realizar las siguientes reducciones.

   (a) $HC \propto TSP$ (HC=Ciclo hamiltoniano, TSP=Problema del viajante).

   (b) $TSP \propto PLE$

Aquellos que estén interesados en este tema **deben** leer este libro:

```
Computers and Intractibility:
A guide to the theory of NP-completeness
Michael R. Garey and David S. Johnson
W. H. Freeman, 1979.
```

Chapter 12

12. Sorting Networks

12.1 Sort utilizando varios procesadores

Los métodos de sort han sido estudiados con mucho detenimiento por aquellos que se dedican al análisis y disenio de algoritmos, los métodos de sort mas estudiados entre los cuales incluimos al burbujeo, selección, inserción, quicksort, heapsort y mergesort entre otros se basan en la suposición de que se trabaja con un solo procesador y realizando comparaciones para ordenar. Como fue demostrado oportunamente este tipo de algoritmos tiene una cota inferior dada por $\Omega(n \log n)$ que se deriva de la necesidad de realizar una cierta cantidad de comparaciones para poder determinar el ordenamiento de cualquier secuencia.

Otros métodos de sort que se estudiaron no funcionan en base a comparaciones y permiten realizar el proceso de sort en tiempo lineal, estos métodos sin embargo no pueden aplicarse al ordenamiento de cualquier tipo de datos y estan restringidos a ordenar enteros chicos, fechas, etc. Counting-sort, Radix-Sort y Bucket-Sort son los algoritmos mas conocidos de este tipo.

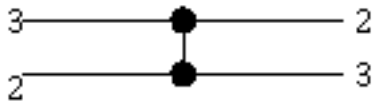
El avance de la tecnología en los ultimos anios ha causado que las máquinas que solo disponen de un procesador estan lentamente desapareciendo dejando su lugar a arquitecturas mas modernas basadas en al menos dos procesadores y en muchos casos una cantidad mucho mayor (computadoras con mas de mil procesadores han sido ya construidas en estos días).

Al disponer de mas de un procesador es posible para un método de sort realizar mas de una comparación a la vez, de hecho se pueden realizar tantas comparaciones paralelas como procesadores estén disponibles. Para aprovechar esta característica se deben diseniar nuevos algoritmos de sort que permitan realizando una cierta cantidad de comparaciones paralelas ordenar una secuencia en forma eficiente. El algoritmo que permite esto se basa en una Sorting-Network

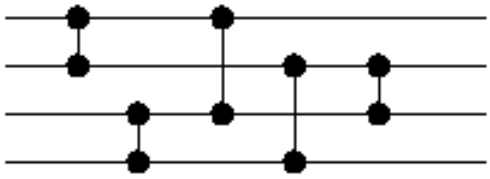
o Red de Ordenamiento y permite ordenar cualquier conjunto de datos en tiempo sub-lineal.

12.2 Sorting Networks

Una Red de Ordenamiento es una red que se construye a partir de un dispositivo básico denominado *comparador* (comparator). Un comparador es un dispositivo que tiene dos entradas $e1, e2$ y dos salidas $s1, s2$ y que al ingresarse dos números cualesquiera en las entradas genera una salida en la cual $s1$ es el menor de los dos números ingresados y $s2$ es el mayor de los números ingresados.



La figura muestra un comparador, una red de ordenamiento para un secuencia de longitud n esta formada por n conectores o cables sobre los cuales se disponen los comparadores.



La figura muestra una red de ordenamiento que permite ordenar una secuencia de cuatro elementos utilizando 5 comparadores.Por ejemplo si la secuencia a ordenar fuera $\langle 3, 1, 4, 2 \rangle$ la red realizaría lo siguiente.

```
1. 3-1-4-2 (inicial)
2. 1-3-4-2 (primer comparador)
3. 1-3-2-4 (segundo comparador)
4. 1-3-2-4 (tercer comparador)
5. 1-3-2-4 (cuarto comparador)
6. 1-2-3-4 (quinto comparador)
7. 1-2-3-4 (final)
```

Paralelismo Si estuviéramos trabajando en una máquina de un solo procesador el procedimiento realizaba 5 comparaciones (una por comparador), sin embargo usando varios procesadores podemos realizar mas de una comparación al mismo tiempo, a los efectos teóricos vamos a suponer que podemos realizar un número ilimitado de comparaciones al mismo tiempo.

Si observamos la red de ordenamiento vamos a ver que los comparadores 1 y 2 se pueden activar al mismo tiempo, luego los comparadores 3 y 4 se activan a la vez y por ultimo el comparador 5, por lo que la secuencia con paralelismo es:

```
1. 3-1-4-2 (inicial)
2. 1-3-2-4 (primer y segundo comparadores)
3. 1-3-2-4 (tercer y cuarto comparadores)
4. 1-2-3-4 (quinto comparador)
5. 1-2-3-4 (resultado final)
```

Debemos definir cuando dos comparadores se pueden activar en paralelo:

N comparadores pueden activarse en paralelo si y solo si no tienen conectores en común y ademas los $2N$ conectores involucrados no tienen comparadores previos sin activarse.

De acuerdo a esto los comparadores 1 y 3 de nuestro ejemplo no pueden activarse a la vez pues tienen al conector 1 en comun.

12.2.1 Principio de paralelismo

En cada instante se activa la cantidad máxima de comparadores que pueden activarse

12.2.2 Análisis de redes de ordenamiento

La eficiencia de una red de ordenamiento se mide de acuerdo a su **profundidad**. La profundidad de un conector esta dada por la cantidad de comparadores que inciden sobre el. La profundidad de la red es el máximo entre las profundidades de sus conectores.

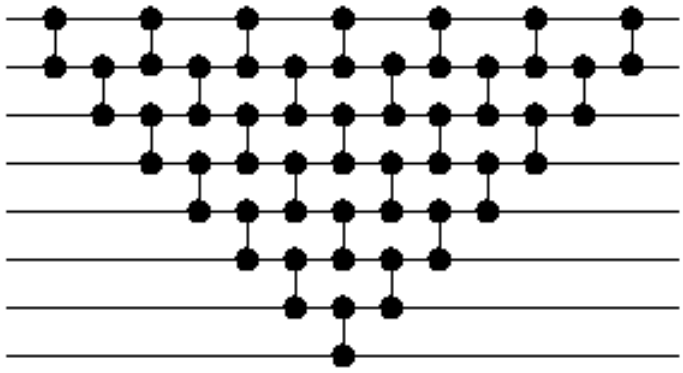
En nuestro ejemplo tenemos:

```
Conector 1 = Profundidad 2
Conector 2 = Profundidad 3
Conector 3 = Profundidad 3
Conector 4 = Profundidad 2

Profundidad de la red : 3
```

La eficiencia de la red es inversamente proporcional a su profundidad.

12.2.3 Ejemplo I

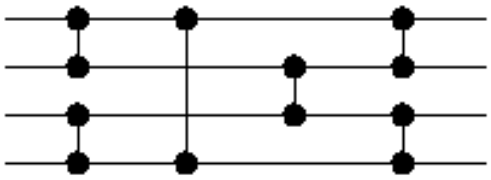
La siguiente red que ordena una secuencia de ocho elementos esta basada en el método insert-sort.



Como puede verse la profundidad de la red es 7.

12.2.4 Ejemplo II

Esta es otra red para ordenar cuatro elementos en la cual todos los conectores tienen profundidad 3.



12.3 El principio del 0-1

Teorema: Si una red de ordenamiento ordena una secuencia $\langle s1, s2, \ldots sn \rangle$ entonces dada una función monótona y creciente f la red tambien ordena

$$\langle f(s1), f(s2), \ldots, f(sn) \rangle$$

.

Esto puede demostrarse en forma bastante simple aplicando inducción aqui no lo hacemos para economizar tiempo y espacio.

Teorema del 0-1: Si una red de ordenamiento ordena todas las secuencias posibles de n elementos donde cada elemento de la secuencia es 0 o 1, entonces la red ordena cualquier conjunto de elementos.

Demostración: Supongamos que la red ordena a cualquier secuencia binaria de n elementos pero que sin embargo no es capaz de ordenar la secuencia $\langle a1, a2, \ldots, an \rangle$. Entonces podemos decir que para un cierto i y un cierto j con $i < j$ la red ubica a aj antes que a ai en el resultado final. Definamos la siguiente función:

$$f(x) = \{ \begin{array}{lll} 0 & si & x \leq ai \\ 1 & si & x > ai \end{array}$$

Como la red ubica a aj antes que a ai al recibir $\langle a1, a2, \ldots, an \rangle$ entonces por el teorema anterior como $f(x)$ es monótona y creciente ubica a $f(aj)$ antes que a $f(ai)$. Como $f(aj) = 1$ y $f(ai) = 0$ resulta que la red ubica un 1 antes que un 0 lo cual es una contradicción porque supusimos que la red ordenaba cualquier secuencia binaria. Lo cual demuestra el teorema.

Este teorema es útil pues permite construir una red de ordenamiento basándose en secuencias binarias conociendo que si se ordenan todas las posibles secuencias binarias se puede ordenar cualquier secuencia genérica.

12.4 Construcción de una red de ordenamiento

En esta sección vamos a mostrar como construir una red de ordenamiento de orden sub-lineal que sirve para ordenar cualquier conjunto de elementos. Las redes de ordenamiento fueron introducidas por primera vez en 1954 por Armstrong, Nelson y O'Connor. El método que aqui describimos que es el mas eficiente fue introducido por Batcher en 1960.

12.4.1 Secuencias Bitónicas

Una secuencia es bitónica si y solo si cumple alguno de estos requisitos:

1. Puede generarse concatenando dos secuencias $s1$ y $s2$ siendo $s1$ monotona y creciente y $s2$ monotona y decreciente.

2. Puede generarse concatenando dos secuencias $s1$ y $s2$ siendo $s1$ monotona y decreciente y $s2$ monotona y creciente.

3. La secuencia esta formada por n elementos iguales, en cuyo caso se la denomina secuencia bitónica pura.

Ejemplos

$$\langle 1, 5, 15, 23, 7, 4, 2 \rangle$$

$$\langle 10, 11, 12, 13, 14, 2 \rangle$$

$$\langle 1, 1, 1, 1, 1, 1, 1, 1 \rangle$$

Secuencias bitónicas binarias Las secuencias bitónicas binarias pueden responder a las siguientes formas:

$$\langle 1, 1, 1, 1, 1, 1, 1, 1 \rangle$$

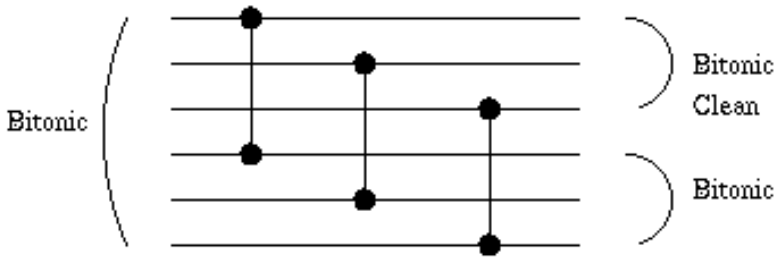
$$\langle 1, 1, 0, 0, 0, 1, 1, 1 \rangle$$

$$\langle 0, 0, 0, 1, 1, 1, 0, 0 \rangle$$

12.4.2 Bitonic-Sorter

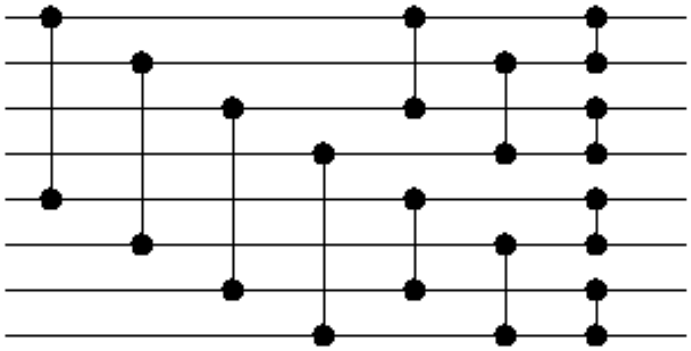
El primer paso para construir la red de ordenamiento es construir una red que permita ordenar una secuencia bitónica, para ello utilizamos un dispositivo que se denomina Bitonic-Sorter. Un Bitonic-Sorter de 2 elementos es un comparador simple, para mas de dos elementos se recurre a la ayuda de un dispositivo denominado Half-Cleaner.

Half-Cleaner



Un Half-Cleaner recibe una secuencia bitónica de longitud N y genera dos secuencias bitónicas de longitud $n/2$, en donde una es una secuencia bitónica y la otra es una secuencia bitónica pura (de aqui el nombre de Half-Cleaner). Los Half-Cleaner tienen profundidad 1.

Un Bitonic-Sorter se construye en forma recursiva utilizando un Half-Cleaner y luego aplicando Bitonic-Sort a las dos secuencias generadas por el Half-Cleaner.



La figura muestra un Bitonic-Sorter para ocho elementos, esta compuesto por un Half-Cleaner de 8 elementos y luego 2 Bitonic-Sorters de 4 elementos (uno arriba y otro abajo), cada uno de estos tiene a su vez un Half-Cleaner de 4 elementos y Bitonic-Sorters de 2 elementos los cuales son comparadores simples.

Análisis

La profundidad de un Bitonic-Sorter esta dado por la clasica recurrencia $T(n) = 2T(N/2) + 1$ (porque un Half-Cleaner tiene profundidad 1) y por lo tanto es $O(\log n)$.

12.4.3 Merger

Otro dispositivo necesario para construir la red de ordenamiento es la construccion de un Merger, observemos que dadas dos secuencias ordenadas si a la primera le concatenamos la reversa de la segunda obtenemos una secuencia bitónica, por ejemplo:

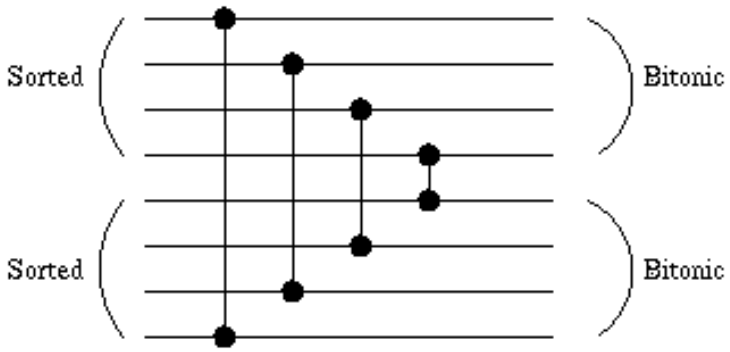
$$s1 = \langle 1, 5, 14, 34, 76 \rangle$$

$$s2 = \langle 2, 3, 8, 11, 20 \rangle$$

$$s2^{-1} = \langle 20, 11, 8, 3, 2 \rangle$$

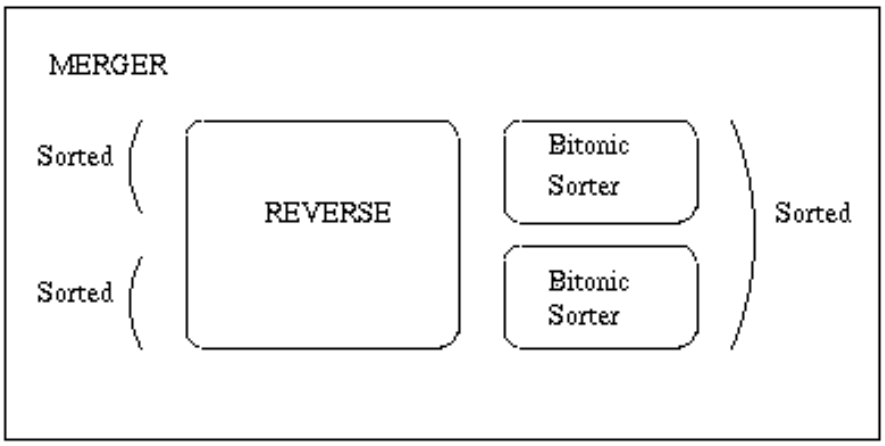
$$s1.s2^{-1} = \langle 1, 5, 14, 34, 76, 20, 11, 8, 3, 2 \rangle$$

Para proceder al merge utilizamos un dispositivo que dadas dos secuencias ya ordenadas construye dos secuencias bitónicas. El dispositivo tiene como propiedad que la primera secuencia tiene siempre elementos menores a la segunda y es:



Como puede verse este dispositivo tiene profundidad 1.

Por lo anto si tomamos dos secuencias ordenadas y les aplicamos este dispositivo y a los resultados les aplicamos Bitonic-Sorter lo que obtenemos es una sola secuencia ordenada, es decir el merge de ambas.

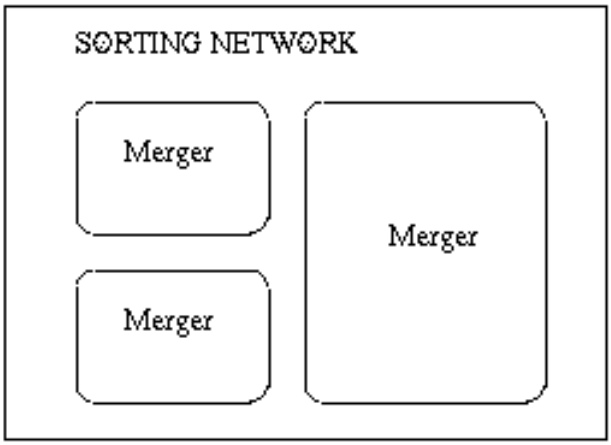


Análisis

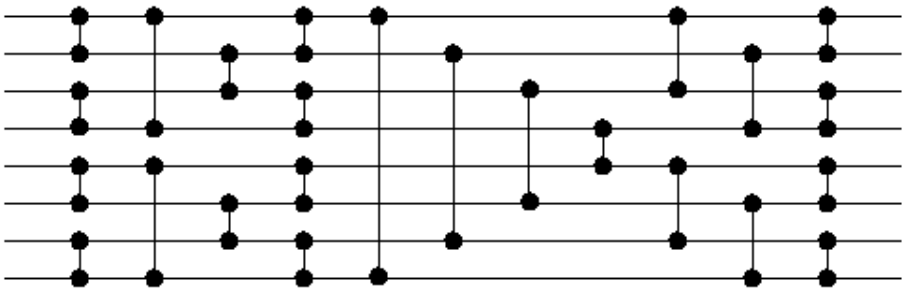
El analisis del Merger es idéntico al del Bitonic-Sorter por lo que su profundidad es $O(\log n)$

12.4.4 Sorting Network

Una vez construido un merger es sencillo construir un sort utilizando una suerte de merge sort invertido, empezamos por hacer un $N/2$ merges de 2 elementos cada uno, luego hacemos $N/4$ merges de 4 elementos cada uno y asi sucesivamente hasta obtener la secuencia ordenada, el dispositivo es de la forma:



Por ejemplo para 8 elementos la red de ordenamiento que se construye es:



Análisis

La profundidad de la red de ordenamiento esta dada por la recurrencia.

$$T(n) = T(n/2) + \log n$$

Y su solucion es $O(\log^2 n)$, siendo esta una función de orden sub-lineal.

12.5 Resumen y puntos clave

Como vemos la construcción de una red de ordenamiento es sencilla e intuitiva a partir de un conjunto de dispositivos muy simples, las redes de ordenamiento son sumamente eficientes cuando se dispone de varios procesadores ya que permiten aprovechar el paralelismo de los mismos para construir un algoritmo de sort genérico de tiempo sub-lineal.

12.6 Ejercicios

1. Indicar cuantos comparadores son necesarios para una Sorting-Network de n elementos
2. Probar que la profundiad de una red de ordenamiento es exactamente $(\log n)(\log n + 1)/2$

Chapter 13

Aproximación y Heurísticas

13.1 Introducción

Como se vio en el capítulo sobre complejidad existen numerosos problemas de optimización que no pueden resolverse en tiempo polinómico. Como varios de estos son problemas realmente importantes no es posible ignorarlos debido a que ciertas aplicaciones necesitan que se resuelvan estos problemas. Las opciones para enfrentar este tipo de problemas son:

- Usar un algoritmo de orden exponencial: Incluso en las computadoras paralelas mas avanzadas esta opción solo es aplicable para instancias sumamente reducidas de estos problemas, hay pocas esperanzas de que el avance de la tecnología permita resolver problemas para mas de 100 elementos mediante un algoritmo exponencial.

- Heuristicas: Frecuentemente no es necesario encontrar una respuesta óptima a un problema sino que alcanza con descartar las soluciones que son realmente muy malas. En estos casos existen algoritmos heurísticos basados en estrategias muy fáciles de computar pero que no necesariamente llevan a una solucion óptima.

  Una **regla heurística** es una regla que establece como tomar una decisión a partir de un conocimiento intuitivo del problema a resolver.

- Inteligencia Artificial: Hay varias técnicas poderosas para resolver problemas de optimización combinatorios desarrollados en el campo de la inteligencia artificial. Algunas de las técnicas mas populares involucran algoritmos genéeticos, redes neuronales, búsquedas de tipo A^* y variaciones dela programación lineal. El rendimiento de estos métodos depende sensiblemente del problema al cual se apliquen. A menudo estos métodos obtienen soluciones muy buenas.

- Algoritmos de aproximación Son algoritmos de orden estrictamente polinómico que calculan una solución con un cierto margen de tolerancia con respecto de la solución óptima, son muy necesarios cuando es necesario garantizar la calidad de la solución. Pueden ser vistos como heurísticas que tienen una cierta performance calculable.

13.2 Algoritmos de aproximación

Un algoritmo de aproximación es aquel que permite obtener una solución para un problema que puede no ser óptima pero que no se aleja del óptimo mas alla de un cierto factor conocido.

13.2.1 Tasa de aproximación

Sea $C(I)$ el costo de la solución producida por un algoritmo de aproximación, y sea $C^*(I)$ el costo de la solución óptima. Asumimos que los costos son valores estrictamente positivos. Para un problema de minimización queremos que $C(I)/C*(I)$ sea lo mas chico posible mientras que para un problema de maximización queremos que $C*(I)/C(I)$ sea lo mas chico posible. Para un problema de longitud n decimos que el algoritmo alcanza una *tasa de aproximación*[1] $\rho(n)$ si para todo I se verifica que:

$$max\left(\frac{C(I)}{C^*(I)}, \frac{C^*(I)}{C(I)}\right) \leq \rho(n)$$

Observemos que $\rho(n)$ siempre es mayor o igual que 1 y es igual a uno si y solo si la solución aproximada es igual a la solución óptima. De esta forma podemos clasificar los algoritmos de aproximación de acuerdo al valor de $\rho(n)$, muchas veces $\rho(n)$ es un valor fijo que no depende de n, por ejemplo $\rho(n) = 2$ implica que el algoritmo siempre produce soluciones que son a lo sumo el doble de la solución óptima, cuanto mas chico sea $\rho(n)$ mejor será la aproximación. En otros casos sin embargo $\rho(n)$ varía según el tamanio de n, el objetivo es entonces que $\rho(n)$ crezca poco en función de n, $\rho(n) = \log n$ podría no ser malo, pero $\rho(n) = n$ es demasiado impreciso.

Además de la tasa de aproximación $\rho(n)$ puede ser interesante calcular la *tasa de error* del algoritmo que se calcula como:

13.2.2 Tasa de error

$$\epsilon = \frac{|C(I) - C^*(I)|}{C^*(I)}$$

[1] aproximation ratio

13.2.3 Aproximando problemas de optimización

Los problemas de optimización presentan comportamientos diversos al intentarse una aproximación, algunos problemas pueden ser aproximados al óptimo tanto como se desee en tiempo polinómico (obviamente cuando la solución tiende al óptimo el tiempo tiende a infinito.) Estos algoritmos se denominan *esquema de aproximación polinómica* y presentan un tiempo de ejecución del estilo $O(2^{(1/e)}n^2)$ de forma tal que cuando e, el error que se comete es mas chico el tiempo se vuelve mayor. Si el tiempo solo depende de una función polinómica de $1/e$ el algoritmo se clasifica como *esquema de aproximación polinómica completo* $O((1/e)^2n^3)$ sería un ejemplo de esta clase.

Otros problemas no pueden aproximarse mas alla de un cierto límite, por ejemplo hay problemas para los cuales no hay algoritmo que supere $\rho(n) = 3/2$ independientemente del tiempo que se disponga. En otros casos ni siquiera podemos garantizar un factor constante y en los casos mas extremos se llega a demostrar que encontrar un algoritmo de aproximación con tasa de aproximación constante es en si mismo un problema NP-Completo.

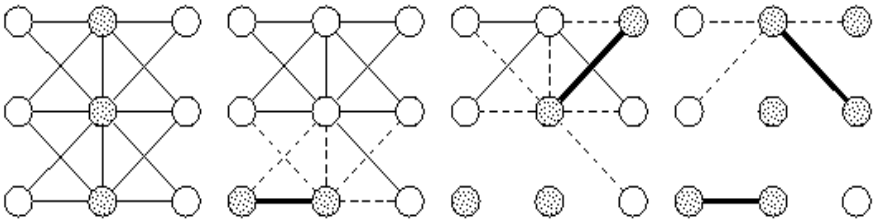
13.3 Vertex Cover (VC)

El problema (VC) es un problema NP-Completo conocido que, recordemos, consite en encontrar un conjunto mínimo de vértices de un grafo de forma tal que cualquier arista del grafo parte o llega a alguno de los vértices en el conjunto.

13.3.1 La heurística del dos por uno

Existe una heurística muy sencilla para Vertex-Cover que garantiza $\rho(n) = 2$. Consideremos una arista cualquiera del grafo (u, v), uno de sus dos vértices debe pertenecer al conjunto pero no sabemos cual de los dos. La idea de la heurística es simplemente poner ambos vértices en el conjunto[2]. Luego quitamos todas las aristas que sean incidentes a u y v (ya que ya están cubiertas). Para cada vértice que debe estar en el conjunto ponemos dos por lo que es sencillo ver que esta estrategia genera a los sumo un conjunto del doble de tamanio del conjunto mínimo (óptimo).

[2]No se puede ser mas tonto



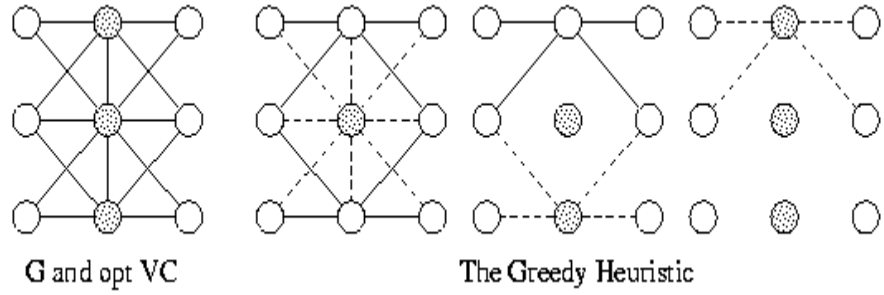
Algorithm 64 Aprox-Vertex-Cover1(G=(V,E)) Calcula el VC usando 2x1

$C \leftarrow vacio$
while E no es vacio **do**
  $(u, v) \leftarrow$ elegir una arista cualquiera
  add u, v to C
  remove from E incident edges to v o u
end while
return C

13.3.2 La heurística golosa

Aparentemente existe una forma simple de mejorar la heurística del 2x1. El algoritmo 2x1 selecciona cualquier arista y agrega ambos vértices al conjunto, en lugar de esto podemos elegir vértices de grado alto ya que un vértice de grado alto cubre mayor cantidad de aristas. Esta es la heurística golosa.

Aproximación con la heurística golosa Seleccionamos el vértice de mayor grado y se lo agrega al conjunto. Luego eliminamos todas las aristas que son incidentes sobre este vértice. Se repite el algortimo en el grafo resultante hasta que no quedan mas aristas.



Algorithm 65 Aprox-Vertex-Cover2(G=(V,E)) Calcula el VC usando la heuristica golosa

$C \leftarrow vacio$
while E no es vacio **do**
  $u \leftarrow$ vértice de grado maximo
  add u to C
  remove from E incident edges to u
end while
return C

Para implementar este algoritmo en forma eficiente puede que sea necesario utilizar un Heap en donde se almacenen los vértices ordenados de acuerdo a su grado.

Es interesante notar que en nuestro ejemplo (ver ilustración) la heurística golosa calcula la solución óptima real, esto no ocurre en todos los casos. La pregunta es si es posible demostrar que la heurística golosa siempre es superior a la heurística del 2x1, la respuesta es un *no definitivo*, de hecho la heuristica golosa no tiene una tasa de aproximación constante. Puede resultar arbitrariamente ineficiente o puede llegar a la solución óptima. Aunque no puede demostrarse es aceptable decir que la heurística golosa es superior a la del 2x1 para grafos aceptados como normales.

13.4 Aproximaciones y reducciones

Una vez que se consigue un algoritmo de tipo $\rho(n) = 2$ para un problema NP-Completo podría suponerse que cualquier problema NP-Completo puede aproximarse usando $\rho(n) = 2$ ya que aplicando reducciones se puede probar que los problemas son equivalentes. Esto no es cierto ya que la tasa de aproximación no necesariamente se preserva al aplicar una reducción.

Por ejemplo recordemos que si V' es el conjunto (VC) de G entonces $V - V'$ es un conjunto independiente de G. Supongamos que G tiene n vértices y que el (VC) de G tiene k vértices. Entonces nuestra heurística garantiza un (VC) V'' que tiene a lo sumo $2k$ vértices. Si consideramos el complemento $V - V'$ sabemos que G tiene un conjunto independiente de tamanio $n - k$. Complementando la aproximación. $V - V''$ tenemos un conjunto independiente aproximado de

tamanio $n-2k$, calculemos a partir de esto la tasa de aproximación.

$$\rho(n) = \frac{n-k}{n-2k}$$

El problema de esta tasa de aproximación es que podría resultar arbitrariamente grande. Por ejemplo si $n = 1001$ y $k = 500$ $\rho(n) = 500$ lo cual es terrible.

Como vemos la teoría de las reducciones no es aplicable a las aproximaciones.

13.5 El problema del centro equidistante

Supongamos que una importante cadena de video-clubes desea abrir una serie de locales (k locales en total) en una ciudad y necesita encontrar la ubicación ideal para los negocios de forma tal de minimizar la distancia máxima hasta alguno de los negocios desde cualquier punto de la ciudad. Si modelamos la red de caminos de la ciudad como un grafo no-dirigido donde los pesos de las aristas representan las distancias entre las intersecciones entonces tenemos una instancia del problema del centro equidistante.

El problema del centro equidistante *Dado un grafo no-dirigido $G = (E, V)$ con pesos no-negativos en las aristas y dado un entero k calcular un subconjunto de k vértices $C \subseteq V$ denominados centros de forma tal que la distancia máxima entre cualquier vértice de V y su centro mas cercano es mínima. Este problema es NP-Completo.*

Sea $G = (V, E)$ el grafo y sea $w(u, v)$ el peso de la arista (u, v), $w(u, v) = w(v, u)$ ya que G es no-dirigido. Suponemos que todos los pesos son no-negativos. Para cada par de vértices $u, v \in V$ sea $d(u, v) = d(v, u)$ la distancia de u hasta v es decir el costo del camino mínimo desde u hasta v.

Consideremos el subconjunto $C \subseteq V$ de vértices: *centros*. Para cada vértice $v \in V$ podemos asociarlo con su vértice mas cercano denotado $C(v)$. Podemos tambien, asociar cada centro con el conjunto de vértices denominado su *vecindario* para el cual el centro es su centro mas cercano. Para simplificar las definiciones supongamos que la distancia al centro mas cercano es unica. Para cada $v \in V$ y $ci \in Ci$ definimos.

$$C(v) = ci\, tal\, que\, d(v, ci) \leq d(v, cj)\, para\, i \neq j$$

$$V(ci) = \{v\, tal\, que\, C(v) = ci\}$$

Si rompemos los empates para los vértices que son equidistantes desde dos o mas centros podemos pensar en $V(c1), V(c2), \ldots$ formando una partición de los vértices de G. La distancia asociada con cada centro es la distancia al vértice mas lejano de V(ci):

$$D(ci) = max_{v \in V(ci)} d(v, ci)$$

Esta es la distancia maxima oara cualquier vértice asociado con ci con respecto al centro. Finalmente definimos.

$$D(C) = max_{ci \in C} D(ci)$$

Que es la distancia máxima desde cualquier vértice hasta su centro mas cercano, esta es la distancia crítica para la solución y por eso se la denomina *cuello de botella*, todo vértice que esté a esta distancia de su centro mas cercano en la solución se denominará vértice cuello de botella.

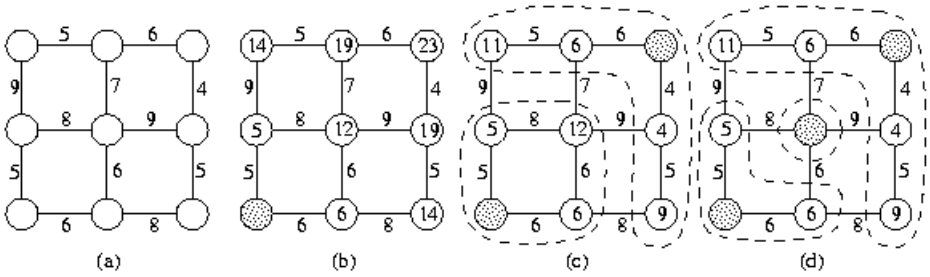
Dada la nomenclatura podemos definir el problema del centro equidistante de ls siguiente manera: dado un grafo no dirigido $G = (V, E)$ y un entero $k \leq |V|$ encontrar un subconjunto $C \in V$ de tamanio k de forma tal que $D(C)$ sea mínimo.

Una solucion por fuerza bruta implicaría enumerar todos los subconjuntos de k elementos de V y calcular $D(C)$ para cada uno, esto es $O(n^k)$. Dado que el problema es NP-Completo es altamente improbable que un algoritmo significativamente mas eficiente exista.

13.5.1 Aproximación golosa

Nuestro algoritmo de aproximación se basa en un algoritmo goloso simple que produce una distancia cuello de botella D(C) que no es mas del doble de la distancia óptima.

Comenzamos permitiendo que el primer vértice $c1$ sea cualquier vértice del grafo. Calculamos las distancias entre este vértice y los demas vétices del grafo, tomamos el vértice mas lejano de este vrtice al cual consideramos el vértice cuello de botella de $c1$. A este vétice lo elegimos como próximo centro $c2$. A continuacion computamos las distancias de todos los vétices del grafo al centro mas cercano ($c1$ o $c2$). Nuevamente tomamos el vértice cuello de botella como el próximo centro y continuamos. El proceso se repite hasta seleccionar k centros.



Algorithm 66 KCenter-Aprox(G,k) usando la heurística golosa

for each u in v **do**
  $d[u] \leftarrow \infty$
end for
for $i = 1$ to k **do**
  $c[i] \leftarrow u$ tal que d[u] es maximo (cuello de botella)
  for each v in V **do**
    $d[v]$ =distancia minima desde v hasta cualquier centro $c1, c2, \ldots, ect$
  end for
end for
return c[1..k]

Mediante el algoritmo de Dijkstra podemos calcular los caminos minimos desde un vértice hasta todos los demás. Sin embargo cada paso de este algoritmo pide solucionar múltiples problemas de este tipo, esto puede hacerse con una modificación del algoritmo original de Dijkstra. El tiempo del algoritmo es k veces el tiempo del algoritmo de Dijkstra : $O(k(n+e)\log n)$

13.5.2 Tasa de aproximación

Queremos mostrar que este algoritmo siempre produce una distancia final $D(C)$ que es a lo sumo el doble de la distancia de la solución óptima es decir que $\rho(n) = 2$.

Sea $C^* = \{c^*1, c^*2, \ldots, c^*k\}$los centros de la solucion óptima y sea $D^* = D(C^*)$ la distancia cuello de botella óptima.

Sea $C_G = \{c1, c2, \ldots, ck\}$ los centros del algoritmo goloso. Ademas sea c_{k+1} el próximo centro que sería agregado, es decir el vértice cuello de botella para C_G. Sea $D_G = D(C_G)$ la distancia cuello de botella de C_G notemos que la distancia desde c_{k+1} a su centro mas cercano es D_G

Teorema: $D_G \leq 2D^*$

Demostración: Sea $C = \{c1, c2, \ldots, ck, ck+1\}$ el (k+1) elemento consistente de centros golosos y el próximo centro goloso. Observemos que $d(ci, cj) \geq D_G$. Este es el resultado de nuestra estrategia golosa de selección. A medida que los centros son seleccionados cada uno de ellos es la distancia cuello de botella para los centros anteriores. A medida que agregamos mas centros la distancia cuello de botella solamente puede decrementarse. Dado que la distancia cuello de botella final es D_G todos los centros estan por lo menos alejados esta distancia entre si.

Cada $ci \in C'$ esta asociado con su centro mas cercano en la solución óptima, es decir que corresponde a $V(c^*k)$ para un cierto k. Debido a que hay k centros en C^* y $k+1$ elementos en C' se deduce que por lo menos dos centros de C estan en el mismo conjunto $V(c^*k)$ para algun k. Sean estos centros ci y cj.

Dado que D^* es la distancia cuello de botella para C^* sabeos que hay un camino de distancia D^* desde ci hasta c^*k y un camino de distancia D^* desde c^*k hasta cj implicando que existe un camino de longitud $2d^*$ desde ci hasta cj. Por lo tanto $d(ci, cj) \leq 2D^*$. Pero de los comentarios anteriores tenemos que $d(ci, cj) \geq D_G$. Por lo tanto:

$$D_G \leq d(ci, cj) \leq 2D^*$$

Que es lo que se quería demostrar.

13.6 El problema del viajante

Durante el estudio de los problemas NP-Completos enunciamos este problema que es muy importante, dado un grafo dirigido y pesado donde los pesos representan distancias entre los vértices encontrar un ciclo de longitud mínima que permita recorrer todos los vértices sin repetir vértices ni aristas.

13.6.1 Aproximación para el problema Euclideano del viajante

Un grafo cumple la desigualdad triangular cuando para toda arista (u, v) no existe un vértice w tal que $d(u, w) + d(w, v) \leq d(u, v)$. Es decir que si existe una arista entre dos vértices entonces la distancia mínima entre estos vértices está dada por la arista. Cuando un grafo cumple con la desigualdad triangular decimos que es *Euclideano*.

$$d(u, v) < d(u, w) + d(w, v)\, desigualdadtriangular$$

Cuando el problema del viajante se plantea sobre un grafo euclideano se dice que se trata del problema del viajante euclideano.

Algoritmo de aproximación Cuando el problema es Euclideano existe un algoritmo que permite solucionar el problema del viajante con $\rho(n) = 2$. El algoritmo es el siguiente:

1. Convertir el problema del viajante a un grafo completo y pesado $G = (V, E)$ no dirigido, asignar distancia ∞ a los caminos que no existen.
2. Calcular el árbol generador minimo (AGM) para G, usando por ejemplo el algoritmo de Kruskal o el de Prim.
3. Recorrer el AGM por ambos lados para obtener un ciclo (ver figura)
4. Ajustar el camino de forma tal de no visitar cada ciudad mas de una vez.

Análisis En primer lugar el paso 2 garantiza que el costo del árbol por se el árbol generador mínimo es menor o igual que el costo de la solución óptima del viajante. (En caso contrario quitando una arista cualquiera a la solución del problema del viajante tendríamos un AGM menor).

Luego al recorrerlo dos veces en el paso 3 garantizamos que la longitud de nuestro tour es menor o igual al doble de la longitud de la solución óptima ya que recorrermos el AGM dos veces.

Por último el paso 4 no puede empeorar la solución ya que de acuerdo a la desigualdad triangular si reemplazamos $(u, w), (w, v)$ por (u, v) la distancia no puede aumentar.

Por lo tanto $\rho(n) = 2$ que es la tasa de aproximación de este algoritmo.

Si el grafo no es euclideano no existe algoritmo de aproximación que permita resolver el problema del viajante con una cierta tasa de aproximación menor a ∞. Esto puede demostarse ya que si dicha aproximación existiese el problema del ciclo Hamiltoniano no sería NP-Completo!.

Para el problema Euclideano existen otros algoritmos de aproximacion, la Heurística de *Christofides* obtiene $\rho(n) = 1.5$ pero insume mucho mas tiempo. Si los pesos de las aristas son las distancias Euclideanas entre puntos en el plano existe una aproximación $(1 + \epsilon)$ (fue descubierta recientemente).

13.7 Otros problemas

13.7.1 K-Coloring

Dado un grafo G indicar cual es la cantidad mínima de colores con las que pueden pintarse los vértices del grafo de forma tal que no existan dos vétices adyacentes pintados del mismo color. Este problema no se puede aproximar en menos de $O(n^{\epsilon})$ a menos que $P = NP$.

13.7.2 Arboles de decision

Calcular cual es el árbol de decisión mas chico para un conjunto de datos dado. Este problema es análogo al de los colores, no se puede aproximar en forma acotada.

13.7.3 Clique

Encontrar el clique de tamanio máximo dado un grafo G. Tampoco puede aproximarse.

13.7.4 Set-Cover

Dado un conjunto $X = \{x1, x2, \ldots, xm\}$ y $F = \{S1, S2, \ldots, Sn\}$ una familia de subconjuntos de X de forma tal que cada elemento de X pertenece al menos a un

conjunto de F. Encontrar un conjunto $C \subseteq F$ de forma tal que cada elemento de X este en algun elemento de C es decir:

$$X = \bigcup_{Si \in C} Si$$

El problema de optimización consiste en encontrar la familia C de tamanio mínimo.

Ejemplo

$$X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

$$S1 = \{1, 2, 3, 4, 5, 6\}$$

$$S2 = \{5, 6, 8, 9\}$$

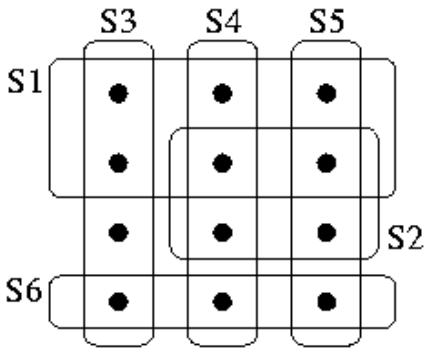
$$S3 = \{1, 4, 7, 10\}$$

$$S4 = \{2, 5, 8, 11\}$$

$$S5 = \{3, 6, 9, 12\}$$

$$C = \{S3, S4, S5\}$$

Este problema es sumamente amplio y cubre por ejemplo al problema de Vertex-Cover (VC), entre otros. Las mejores aproximaciones a este problema son de tipo $\rho(n) = \ln n$. Una aproximación golosa aplicable es elegir cada vez el conjunto Si que cubra mayor cantidad de elementos de X, esto permite $\rho(n) = n$ lo cual no es muy óptimo.



13.8 Resumen y puntos clave

Los algoritmos de aproximación y los algoritmos heurísticos permiten obtener soluciones satisfactorias a problemas que no se pueden resolver en tiempo polinómico. Una regla heurística permite decidir en base a un conocimiento intuitivo del problema, la mayoría de las heurísticas están basadas en tcnicas golosas, es decir que son localmente favorables pero no garantizan una solución globalmente óptima.

Un algoritmo de aproximación puede evaluarse según una tasa de aproximación que indica que tanto puede alejarse la solución encontrada por el algoritmo de la óptima, esto se expresa mediante $\rho(n)$. Que un problema NP-Completo sea aproximable con $\rho(n) = c$ no implica que los demás lo sean pues la tasa de aproximación no se preserva al realizar una reducción.

Algunos problemas pueden aproximarse mediante $\rho(n) = c$, otros pueden aproximarse tanto como se desee en detrimento del tiempo en lo que se denomina esquema de aproximación. Otros problemas no son fáciles de aproximar en tiempo polinómico y algunos no son aproximables en absoluto.

Chapter 14

Indices

14.1 Indice de algoritmos

14.2 Indice Tematico