CC40A
Diseño y Análisis de Algoritmos

Técnicas básicas de diseño

Prof. Benjamin Bustos

Inducción

- Cálculo de números de Fibonacci

$$\begin{cases} f_n = f_{n-1} + f_{n-2} \\ f_0 = 0 \\ f_1 = 1 \end{cases}$$

Inducción

- Tabla de valores:

n	f_n
2	1
3	2
4	3
5	5
6	8
7	13
8	21
…	…
n	$\approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}$

Inducción

- Solución recursiva:

```
int F(int n)
{
  if (n<=1)
    return n;
  else
    return F(n-1)+F(n-2);
}
```

Inducción

- Esto resulta **muy** ineficiente.
- Si $T(n)$ representa el número de sumas ejecutadas para calcular f_n, se tiene:

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

Inducción

- Tabla de valores para $T(n)$:

n	$T(n)$
0	0
1	0
2	1
3	2
4	4
5	7
6	12
7	20

- ¿$T(n)=f_n-1$? Si. (Ejercicio: demostrarlo).

Inducción

- Luego, el tiempo para el algoritmo recursivo crece de manera exponencial: $O(f_n)$
  - ¡Esto es horroroso!
- El problema es que se está recalculando una y otra vez los mismos valores
  - Una forma de evitar esto es anotar los valores calculados en un arreglo
- Solución inductiva:
  - Asumir inductivamente que se conoce f_{n-1} y f_{n-2}
  - Estos valores están almacenados en un arreglo

Inducción

- El arreglo debe llenarse de manera ordenada:

```
int F(int n)
{
  int fib[n+1];
  fib[0]=0;
  fib[1]=1;
  for (i=2; i<=n; ++i)
    fib[i]=fib[i-1]+fib[i-2];
}
```

Inducción

- Tiempo del nuevo algoritmo: $O(n)$
- Espacio del nuevo algoritmo: $O(n)$
- Observación: no es necesario mantener en una tabla todos los valores de f_n
  - Basta con mantener solamente los dos últimos valores
  - Complejidad espacial se reduce a $O(1)$
- ¿Es posible calcular f_n más rápido que $O(n)$? Respuesta: Si.

Inducción

- Se tiene que:

$$\begin{cases} f_n = f_{n-1} + f_{n-2} \\ f_0 = 0 \\ f_1 = 1 \end{cases}$$

- Se define una función auxiliar *g* tal que:

$$\begin{cases} f_n = f_{n-1} + g_{n-1} \\ g_n = f_{n-1} \\ f_1 = 1 \\ g_1 = 0 \end{cases}$$

Inducción

- Con esto se plantea el siguiente sistema de ecuaciones:

$$\underbrace{\begin{bmatrix} f_n \\ g_n \end{bmatrix}}_{\vec{f}_n} = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} f_{n-1} \\ g_{n-1} \end{bmatrix}}_{\vec{f}_{n-1}}, \underbrace{\begin{bmatrix} f_1 \\ g_1 \end{bmatrix}}_{\vec{f}_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\vec{f}_n = A^{n-1} \cdot \vec{f}_1$$

Inducción

- A^{n-1} se puede calcular eficientemente

```
y = 1;
z = x;
for (j=n; j>0; --j) // Invariante: y*z^j == x^n
{
  while (j es par)
  {
    z = z*z;
    j = j/2;
  }
  y = y*z;
}
```

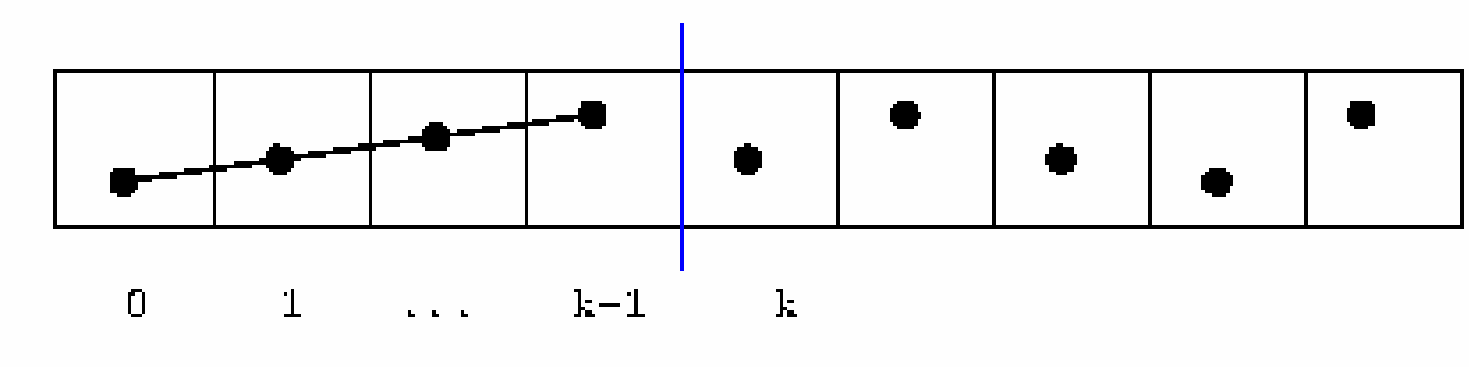
- Tiempo: $O(\log n)$

Inducción

- Algoritmos cuadráticos de ordenamiento
- Ordenación por inserción
  - Se construye un trozo ordenado de izquierda a derecha
  - En cada iteración se agrega un nuevo elemento a dicho trozo
  - El proceso termina cuando todos los elementos del arreglo se encuentran en el trozo ordenado

Inducción

- Solución inductiva
  - Los *k*-1 primeros elementos del arreglo están ordenados
  - Se agrega el *k*-ésimo a la parte arreglada
    - Asegurarse de recuperar el invariante



Inducción

- Algoritmo

```
k = 0; // inicialmente no hay
// elementos ordenados (k=1 también
// funcionaría)
while( k<n )
{
// Insertar a[k] entre a[0],..,a[k-1]
  t = a[k];
  for( j=k; j>0 && a[j-1]>t; --j )
      a[j] = a[j-1];
  a[j] = t;
  ++k;
}
```

Inducción

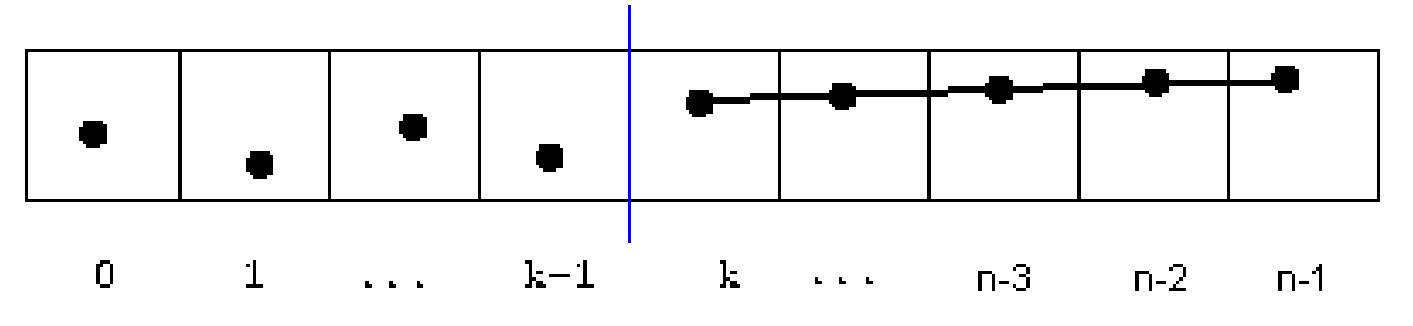
- Costo del algoritmo: $O(n^2)$ comparaciones entre elementos (peor caso y caso promedio)
- Cuesta $O(n)$ si el arreglo está ordenado

Inducción

- Ordenación por selección
  - Se construye un trozo ordenado de derecha a izquierda
  - Se agrega en cada iteración al inicio del trozo ordenado el máximo del conjunto restante
  - Se termina cuando se han agregado todos los elementos del arreglo al trozo ordenado

Inducción

- Solución inductiva
  - Los elementos *a*[*k*], …, *a*[*n*-1] están ordenados
  - Todos los elementos entre a[1] y a[k-1] son menores que *a*[*k*] (por construcción)



Inducción

- Algoritmo

```
k = n-1;
while( k>0 )
{
  // Buscar a[j]=max{a[0],..,a[k]}
  j=0;
  for (int i=1; i<k; i++)
    if (a[i]> a[j])
      j=i;
  // Intercambiar a[j] con a[k]
  aux=a[j];
  a[j]=a[k];
  a[k]=aux;
  --k;
}
```

Inducción

- Costo del algoritmo: $O(n^2)$ comparaciones entre elementos (peor caso y caso promedio)
- Si el arreglo esta ordenado el algoritmo también cuesta $O(n^2)$
- Problema de ambos algoritmos de ordenamiento: sólo intercambian elementos adyacentes del arreglo

Inducción

- Subsecuencia de suma máxima
  - Dados enteros A_1, …, A_n (posiblemente negativos), encontrar el maximo valor de

$$\sum_{k=i}^{j} A_k$$

  - Si todos los números son negativos, la subsecuencia de suma máxima es 0
  - Ejemplo: -2,11,-4,13,-5,-2 la respuesta es 20

Inducción

- Solución 1: Fuerza bruta

```
int maxSum = 0;
for( i=0; i<a.length; i++)
{
  for( j=i; j<a.length; j++)
  {
    int thisSum = 0;
    for (k=i; k<=j; k++)
      thisSum += a[k];
    if (thisSum > maxSum)
      maxSum = thisSum;
  }
}
```

Inducción

- Tiempo: $O(n^3)$

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1=\frac{n^3+3n^2+2n}{6}$$

Inducción

- Solución 2: Mejora a fuerza bruta ($O(n^2)$)

```
int maxSum = 0;
for( i=0; i<a.length; i++)
{
  int thisSum = 0;
  for (j=i; j<=a.length; j++)
  {
    thisSum += a[j];
    if (thisSum > maxSum)
      maxSum = thisSum;
  }
}
```

Inducción

- Solución 3: Algoritmo eficiente
  - Observaciones:
    - No es necesario conocer donde esta la mejor subsecuencia
    - La mejor subsecuencia no puede comenzar en un número negativo
      - Cualquier subsecuencia negativa no puede ser prefijo de la subsecuencia óptima

Inducción

- Solución 3: Algoritmo eficiente
  - Inducción (reforzada)
    - Se conoce la mejor subsecuencia entre 1 y *j*
    - Se conoce la mejor subsecuencia que termina en *j*
  - Algoritmo
    - Se almacenan ambos valores (inicialmente 0)
    - Se incrementa *j* en 1
    - Se actualiza mejor subsecuencia si es necesario
    - Si subsecuencia que termina en j es < 0 se puede descartar, volver su valor a 0

Inducción

- Seudocódigo

```
int maxSum = 0, thisSum = 0;
for( j=0; j<a.length; j++)
{
  thisSum += a[j];
  if (thisSum > maxSum)
    maxSum = thisSum;
  else if (thisSum < 0)
    thisSum = 0;
}
```

- Tiempo: $O(n)$

Dividir para reinar

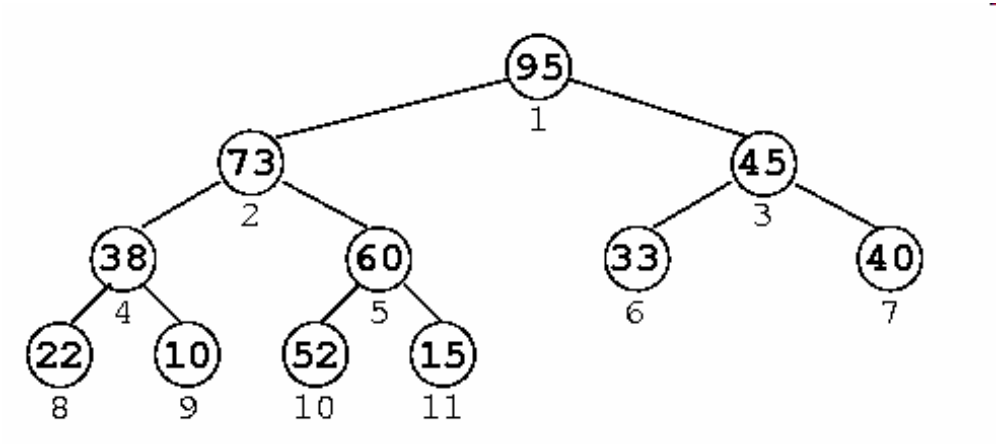
- Algoritmos de ordenamiento eficiente
- Quicksort (seudocódigo):

```
Quicksort(A,n)
{
  Si n==1, return
  Sea p en A
  A1 = {x en A, x < p}
  A2 = {x en A, x > p}
  return quicksort(A1,|A1|)
         :p:
         quicksort(A2,|A2|)
}
```

- $O(n \log n)$ promedio, $O(n^2)$ peor caso

Dividir para reinar

- Heapsort
  - Heap
    - Árbol binario completo salvo posiblemente el último nivel
    - Todo nodo es mayor o igual que sus hijos



Dividir para reinar

- Heapsort
  - Comenzar con una cola de prioridad vacía
  - Fase de construcción de la cola de prioridad:
    - Traspasar todos los elementos del conjunto que se va a ordenar a la cola de prioridad, mediante n inserciones ($O(n \log n)$)
  - Fase de ordenación:
    - Sucesivamente extraer el máximo n veces. Los elementos van apareciendo en orden decreciente y se van almacenando en el conjunto de salida ($O(n \log n)$)

Dividir para reinar

- Optimización de la fase de construcción
- Idea: invertir el orden de creación del heap (input a la izquierda, heap a la derecha)
  - Heap a la derecha: no es realmente un heap, porque no es un árbol completo (le falta la parte superior)
  - Sólo nos interesa que en ese sector del arreglo se cumplan las relaciones de orden entre a[k] y {a[2*k],a[2*k+1]}
  - En cada iteración, se toma el último elemento del input y se le "hunde" dentro del heap de acuerdo a su nivel de prioridad

Dividir para reinar

- Seudocódigo

```
Heapify(A,i,n)
{
  while ( (2i <= n) && ((A[i] < A[2i]) ||
((2i+1 <= n) && (A[i] < A[2i+1]))) )
  {
    l = 2i;
    if ((2i+1 <= n) && (A[2i] < A[2i+1]))
      l = 2i+1;
    swap(A[i],A[l]);
    i=l;
  }
}
```

Dividir para reinar

- Seudocódigo

```
MakeHeap(A,n)
{
  for (i=n; i>=1; i--)
    Heapify(A,i,n);
}

Heapsort(A,n)
{
  MakeHeap(A,n);
  while (n > 0)
  {
    swap(A[1],A[n]);
    n--;
    Heapify(A,1,n)
  }
}
```

Dividir para reinar

- Tiempo para ejecutar MakeHeap: $O(n)$
- Demostración:
  - $h = \log_2(n)$ altura del árbol
  - Costo por cada nivel: $1*n/2$, $2*n/4$, $3*n/8$, …

$$\sum_{i=1}^{h} \frac{n}{2^i} \cdot i = n \sum_{i=1}^{h} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i} = n \cdot \frac{1/2}{(1-1/2)^2} = 2n$$

Dividir para reinar

- Otra demostración:
  - $x=1/2, t=h$

$$\sum_{i=1}^{t} i x^i = x \sum_{i=1}^{t} i x^{i-1} = x \frac{d\left(\sum_{i=1}^{t} x^i\right)}{dx} = x \frac{d\left(\frac{x^{t+1}-x}{x-1}\right)}{dx} = x \cdot \frac{\left((t+1)x^t - 1\right)(x-1) - \left(x^{t+1}-x\right)}{(x-1)^2}$$

$$\Rightarrow$$

$$n \cdot \frac{1}{2} \cdot \frac{\left((h+1)\frac{1}{2^h} - 1\right)\left(\frac{1}{2} - 1\right) - \left(\frac{1}{2^{h+1}} - \frac{1}{2}\right)}{\left(\frac{1}{2} - 1\right)^2} = \frac{n}{2} \cdot \frac{\left((h+1)\frac{1}{2^h} - 1\right)\left(-\frac{1}{2}\right) - \left(\frac{1}{2^{h+1}} - \frac{1}{2}\right)}{\frac{1}{4}} = 2n + o(n)$$

Dividir para reinar

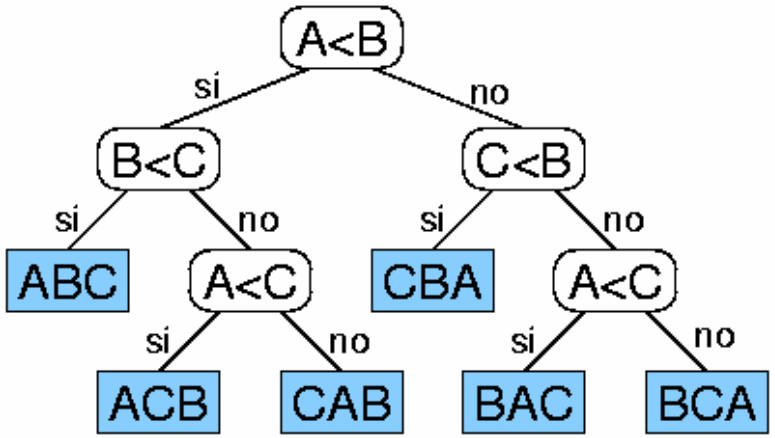
- Costo de fase de ordenación
  - heapify(n): $\log_2(n)$
  - heapify(n-1): $\log_2(n-1)$
  - …

$$\sum_{i=1}^{n} \log i \leq \int_{1}^{n+1} \log x dx = x \log x - x \Big|_{1}^{n+1} \Rightarrow O(n \log n)$$

Dividir para reinar

- Cota inferior de algoritmos de ordenamiento
  - Se desea ordenar tres datos A, B y C
  - Árbol de decisión posible para resolver este problema
    - Los nodos internos del árbol representan comparaciones
    - Los nodos externos representan salidas emitidas por el programa

Árbol de decisión



Dividir para reinar

- Todo árbol de decisión con H hojas tiene al menos altura $\log_2 H$
- La altura del árbol de decisión es igual al número de comparaciones que se efectúan en el peor caso
- En un árbol de decisión para ordenar n datos se tiene que $H=n!$, por lo tanto un algoritmo debe hacer al menos $\log_2 n!$ comparaciones en el peor caso
- Usando la aproximación de Stirling, se puede demostrar que $\log_2 n! = n \log_2 n + O(n)$, por lo cual la cota inferior es de $O(n \log n)$.

Dividir para reinar

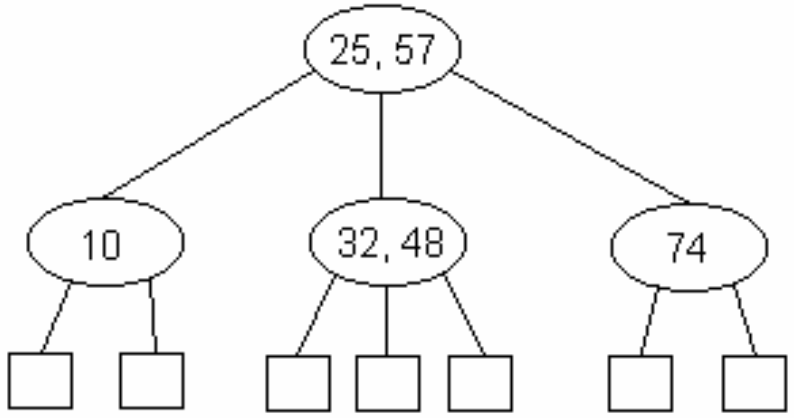
- Los mejores algoritmos de ordenamiento se basan en balancear la partición
- Mejora a Quicksort: mediana de 3
  - Elegir tres elementos aleatorios
  - Elegir el pivote *p* como la mediana de esos tres
  - Particionar y ordenar recursivamente
- Pero caso sigue siendo cuadrático, pero constante de caso promedio es menor

Dividir para reinar

- Repaso de árboles binarios: Árboles de búsqueda binaria (ABB), AVL, árboles 2-3
- Altura árbol binario: $h = 2 \ln n = 1{,}39 \log_2 n$ para árbol balanceado
- AVL: diferencia entre altura de subárbol izquierdo y derecho es a lo más 1
- Altura de árbol AVL:
  - $\log_2 (n+1)$ (promedio)
  - $1{,}44 \log_2 n$ (peor caso)

Dividir para reinar

- Árboles 2-3
  - Nodos internos pueden contener hasta 2 elementos, por lo tanto puede tener 2 o 3 hijos, dependiendo de cuántos elementos posea



$$\log_3(n) \leq altura \leq \log_2(n)$$

Dividir para reinar

- Selección (*k*-ésimo)
  - Problema: dado un arreglo desordenado encontrar el *k*-ésimo del conjunto
- Determinar mínimo o máximo: $O(n)$ (cota mínima)
- Supongamos una especie de torneo entre elementos, *x* es el primero (máximo)
- El segundo puede ser cualquiera de los que perdieron directamente con *x*

Dividir para reinar

- Luego, para calcular segundo, tercero, …, toman tiempo:
  - Segundo: $n+\log_2 n$
  - Tercero: $n+2\log_2 n$
  - …
  - k: $n+(k-1)\log_2 n$
- Esto está bien para k constante, pero para un k genérico (como la mediana)
  - $k=n/2$: $O(n \log n)$

Dividir para reinar

- Quickselect
  - Se basa en el tipo de operaciones de quicksort
  - Se escoge pivote al azar
  - Se particiona el arreglo de acuerdo al pivote escogido
  - Si el pivote cae más allá de la posición *k*, sólo se ordena la parte izquierda
  - Si el pivote estaba en la posición *k*, lo encontramos de inmediato

Dividir para reinar

- Seudocódigo

```
Quickselect(S,k)
{
  Sea p en S
  S1 = {x en S, x < p}
  S2 = {x en S, x > p}
  Si k <= |S1| return Quickselect(S1,k)
  Si k = |S1|+1 return p
  return Quickselect(S2, k-|S1|-1)
}
```

Dividir para reinar

- Peor caso: $O(n^2)$ (mala elección del pivote)
- Caso promedio: $O(n)$
- En la práctica este algoritmo es muy rápido, pero su peor caso es pésimo
- Uno quisiera asegurar una garantía de orden lineal para encontrar el k-ésimo
- Idea: buscar un pivote tal que deje fuera por lo menos una fracción fija del total de elementos

Dividir para reinar

- Método de selección lineal
  - Dividir S en $|S/5|$ conjuntos (cada S_i contiene 5 elementos)
  - Obtener las medianas m_1, m_2, …
  - Obtener p=Select($\{m_i\}$, ($|S|/5)/2$) (mediana de las medianas)

Dividir para reinar

- Características de *p*
  - Mayor que la mitad de las medianas
  - Menor que la otra mitad de las medianas
  - De los grupos con medianas menores (que fueron obtenidas de entre 5 elementos)
    - 3 elementos son menores que *p*
  - De los grupos con medianas mayores
    - 3 elementos son mayores que *p*
  - Esto implica que 3/10 elementos son menores que *p* y que 3/10 son mayores que *p*

Dividir para reinar

- El pivote *p* sólo puede ser mayor que el 3/10 menor y menor que el 3/10 mayor de *S*
  - En el peor caso habrá que buscar recursivamente en un grupo con 7/10 de los elementos

$$T(n) = n + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$$

  - Cálculo de m_i y particiones + cálculo de mediana de medianas + recursión sobre (7/10)*n* restantes

Dividir para reinar

- Suponiendo solución $O(n)$

$$T(n) \leq dn \Rightarrow T(n) \leq n + \frac{dn}{5} + \frac{7}{10} dn \leq dn$$

$$d \geq 10 \Rightarrow T(n) = O(n)$$

Dividir para reinar

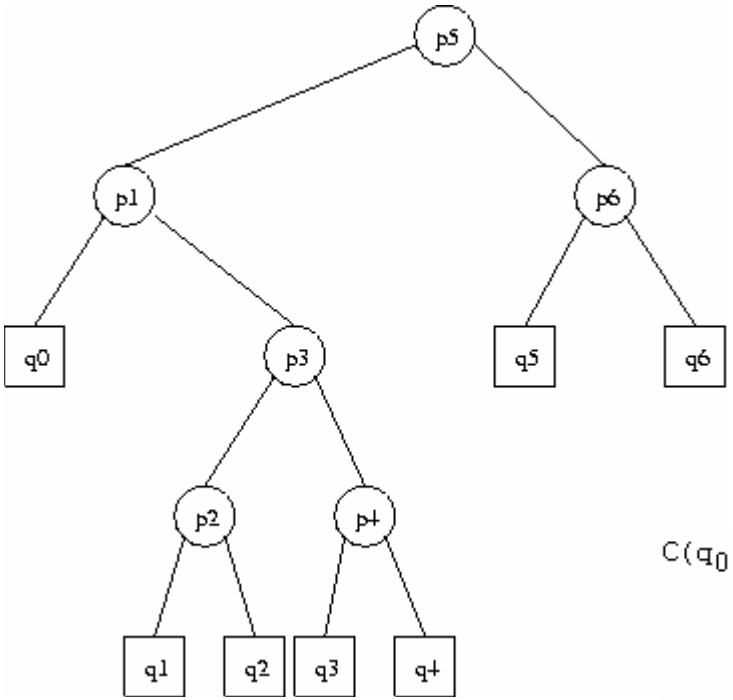
- La elección de 5 elementos para los grupos S_i se debe a que:
  - Este número debe ser impar para obtener mediana exacta
  - Debe ser mayor o igual a 5 para asegurar linealidad del algoritmo
- Se escoge 5 porque:
  - Mediana de medianas queda muy a la mitad
  - Para números muy grandes de elementos calcular las medianas toma tiempo mayor

Programación dinámica

- Árboles de búsqueda binaria (ABB) óptimos
- Problema: probabilidades de acceso no uniforme
  - n elementos $X_1 < X_2 < ... < X_n$
  - Probabilidades de acceso conocidas $p_1, p_2, ..., p_n$
  - Probabilidades de búsqueda infructuosa conocidas $q_0, q_1, ..., q_n$,
  - Encontrar el ABB que minimice el costo esperado de búsqueda

Programación dinámica

- Ejemplo:

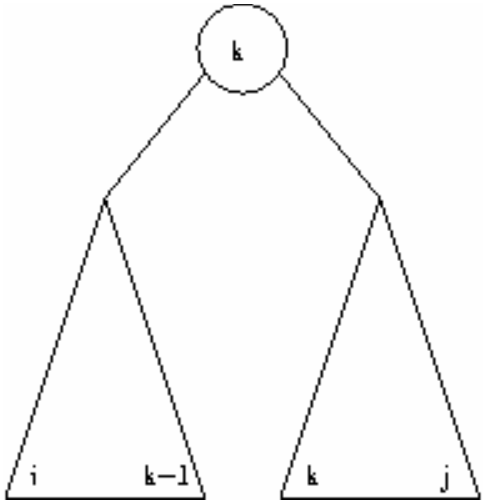


Costo esperado de búsqueda:

$$
\begin{aligned}
C(q_0,q_6) &= (2q_0+2p_1+4q_1+4p_2+4q_2+3p_3+4q_3+4p_4+4q_4)+p_5+(2q_5+2p_6+2q_6) \\
&= (q_0+p_1+q_1+p_2+q_2+p_3+q_3+p_4+q_4+p_5+q_5+p_6+q_6)+ \\
&\quad (1q_0+1p_1+3q_1+3p_2+3q_2+2p_3+3q_3+3p_4+3q_4)+ \\
&\quad (1q_5+1p_6+1q_6) \\
&= W(q_0,q_6)+C(q_0,q_4)+C(q_5,q_6)
\end{aligned}
$$

Programación dinámica

- El costo del árbol completo es igual al "peso" del árbol más los costos de los subárboles
- Si la raíz es *k*



$$C\_opt_{i,j} = \min_{i+1<=k<=j} \{W_{i,j} + C\_opt_{i,k-1} + C\_opt_{k,j}\}$$

$$C\_opt_{i,i} = 0 \text{ para todo } i=0..n$$

Note que el "peso" $W_{i,j}$ no depende de la variable *k*.

Programación dinámica

- Implementación de ABB óptimos
  - Recursiva (equivalente a fuerza bruta): Calcular todos los árboles posibles ($O(4^n)$)
    - Notar que si los subárboles son óptimos no necesariamente el árbol completo será óptimo
    - La raíz pudo haber sido mal escogida
  - Usando programación dinámica (tabulación)
    - Ir calculando árboles óptimos para pocos elementos
    - Probar con todas las raíces hasta encontrar la que minimice el costo total

Programación dinámica

- Seudocódigo

```
for (i = 0; i <= n; i++)
{
  C[i,i]=0; //árboles de tamaño 0
  W[i,i]=q_i;
}
for (l = 1; l <= n; l++)
{
  for (i = 0; i <= n-l; i++)
  {
    j = i+l;
    W[i,j] = W[i,j-1] + p_j + q_j;
    C[i,j] = min_{i+1<=k<=j}{W[i,j]+C[i,k-1]+C[k,j]};
  }
}
```

Programación dinámica

- Tiempo: $O(n^3)$
- Una mejora:
  - Se define $r_{i,j}$ como el k que minimiza

    $\min_{i+1<=k<=j} \{W[i,j]+C[i,k-1]+C[k,j]\}$
  - Intuitivamente $r_{i,j-1} <= r_{i,j} <= r_{i+1,j}$
  - Como $r_{i,j-1}$ y $r_{i+1,j}$ ya son conocidos al momento de calcular $r_{i,j}$, basta con calcular

    $\min_k \{W[i,j]+C[i,k-1]+C[k,j]\}$ con $r_{i,j-1} <= k <= r_{i+1,j}$
- Con esta mejora, se puede demostrar que el método demora $O(n^2)$ (Ejercicio: demostrarlo).