Técnicas de Diseño I

M. Andrea Rodríguez-Tastets
Ayudante: Erick Elejalde

Universidad de Concepción,Chile
www.inf.udec.cl\ ∼andrea
andrea@udec.cl

I Semestre - 2014

Introducción

Al terminar esta sección del curso los alumnos deberán ser capaces de

- Comprender las siguientes técnicas de diseño: achica y vencerás, y transforma y vencerás.
- Analizar la complejidad de estos algoritmos.
- Aplicar algunas de estas técnicas en la resolución de problemas.

Achicar y vencer

- Reduce la instancia del problema a una instancia más pequeñas del mismo problema.
- Resuelve la instancia pequeña
- Extiende la solución de la instancia pequeña para obtener la solución a la instancia original
- Puede ser implementado top o bottom up
- También conocido como enfoque inductivo o incremental

Dividir y vencer: Achicar por un factor

1. Búsqueda binarias y ternarias
2. Mediana
3. Ordenación: mergesort y quicksort
4. Multiplicación
5. Multiplicación de matrices
6. Exponenciación por el cuadrado

Generalización de búsqueda binaria (binary search - BS)

- Dado un intervalo $[a, b]$, y una propiedad P sobre el intervalo tal que $\forall x, y \in [a, b]$, si $x < y$ entonces $P(y) \rightarrow P(x)$. BS nos encuentra el elemento más grande que cumple P o el elemento más chico que no cumple P (el siguiente del más grande que cumple P).
- Intuitivamente, supongamos que tenemos un intervalo $[a, b]$ tal que cumple las condiciones dadas arriba, con algún elemento en el intervalo satisfaciendo P, y que $[a, b]$ tiene tamaño n. Sea $mid = (a + b)/2$, si $P(mid)$ se cumple, entonces como P es monótona, se puede asegurar que el elemento más grande que vale P esá en $[mid, b]$, de lo contrario estoy seguro de que esta en $[a, mid]$.

Algoritmo

```
int BSearch(array A, int r)
  left ← 0
  right ← n
  while (right − left > 1)
     mid ← (left + right)/2
     if (A[mid] <= r)
         left ← mid
     else
         right ← mid
  return left

bool IsFound (array A, int r)
  return A[BSearch(A, r)] == r
```

Ejemplo

- *LowerBound*(r): Dado arreglo ordenado de n elementos, devolver la posición del primer elemento no menor a r (mayor igual). Usar $P(x) : x < r$ y retornar elemento en posición $BSearch(A, r) + 1$ si es que $BSearch(A, r) + 1 < n$, en otro caso, elemento *LowerBound*(r) no existe.
- *UpperBound*(r): Dado arreglo ordenado de n elementos, devuelve la posición del primer elemento mayor a r. Usar $P(x) : x <= r$ y retornar elemento en $BSearch(A, r) + 1$ si $BSearch(A, r) + 1 < n$, en otro caso *UpperBound*(r) no existe.

Complejidad

La complejidad del algoritmo *BSearch* está determinado por cuántas veces itera en el ciclo While. Este costo se puede expresar como un sistema recurrente $T(n) = T(n/2) + O(1)$, con la condición de borde $T(1) = 1$. Ojo que la función *BSearch* no es recursiva pero su función de costo se puede expresar como un sistema recurrente.
Aplicando el Teorema Maestro, $a = 1$, $b = 2$ y $f(n) \in O(n)$. Luego:

$$\begin{aligned} T(n) \quad &\in \quad \Theta(n^{log_b a} log n) \\ &\quad\;\; \Theta(log n) \end{aligned}$$

Esto es el mejor, peor y caso promedio.

Ejemplo: Búsqueda binaria generalizada

- Dado n cajas tal que en cada caja entran $c[i]$ elementos. Dados r elementos, distribuir los r elementos tal que la cantidad de elementos en la caja que más tiene sea mínimo. Devolver este valor.
- La idea de este problema es encontrar el número x que represente el número máximo de elementos en una caja. Notar que si un número x alcanza a distribuir los r elementos en las n, todo elemento mayor que ese x también lo logra, siempre que la capacidad de la caja lo permite.
- Definamos $P(x, r, \text{array } c)$ verdadero si se pueden poner los r elementos de tal modo que cada caja no tiene más de x elementos. La idea es buscar entonces el valor menor de x que cumpla esta condición, o dicho de otro en forma inversa, el valor mayor que no lo cumple.

Ejemplo (Cont.)

```
int Search(array C, int r)
  left ← 0
  right ← r
  while (left < right − 1)
    mid ← (left + right)/2
    if (P(mid, r, C)
        left ← mid
    else
        right ← mid
  return left

bool P(int mid, int r, array C)
  cnt ← 0
  for i = 0 to i < n      cnt ← cnt + min(x, C[i])
    i ← i + 1
  return cnt >= r
```

Búsqueda ternaria: Concepto

- Es una técnica para encontrar el máximo o mínimo en un intervalo de una función que es estrictamente creciente en un intervalo y luego estrictamente decreciente o viceversa.
- Se puede aplicar en funciones continuas o discretas.
- En funciones discretas es importante verificar que la función cumpla que sea estrictamente creciente y luego estrictamente decreciente. (No hay dos valores consecutivos iguales).

Generalización

Asumamos que estamos buscando el máximo de una función f , y sabemos que el máximo está en el intervalo $[A, B]$. Para poder aplicar TS, debe existir algún valor x tal que

- Para todo a, b tal que $A <= a < b <= x$, tenemos $f(a) < f(b)$
- Para todo a, b tal que $x <= a < b <= B$, tenemos $f(a) > f(b)$

Algoritmo

Double $MaximumTS(f, double\ A, double\ B, AbsolutePrecision)$
  $left \leftarrow A$
  $right \leftarrow B$
  while $(right - left >= AbsolutePrecision)$
    $lt \leftarrow (2. * left + right)/3$
    $rt \leftarrow (left + 2. * right)/3$
    if $(f(lt) < f(rt))$
      $left \leftarrow lt$
    else
      $right \leftarrow rt$
  return $(left + right)/2$

Ojo: El valor debe existir dadas las condiciones impuestas.

Complejidad

En cada paso del ciclo dividimos el intervalo en 3 y achicamos el nuevo intervalo a $2/3$ del intervalo inicial. Esto nos da un sistema recurrente $T(n) = T(2/3n) + O(1)$, con la condición de borde $T(1) = 1$. En este caso no se puede aplicar el Teorema Maestro ya que b debe ser mayor que 1.
Analizando el ciclo, en cada paso del ciclo dividimos el intervalo en 3 y achicamos el nuevo intervalo a $2/3$ del intervalo inicial. Por lo tanto generamos una secuencia de de tamaño n,
$(2/3)n, (4/9) \times n, ..(2^j/3^j) \times n$, con j el número de ciclos.
Haciendo $(2/3)^j \times n$ sea tan chico como EPS, la complejidad de la búsqueda ternaria es $O(log2/3(n)) = O(log(n))$.

Ejemplo

Supongamos que granjero John viven en una casa en la posición (xh, yh) con $yh > 0$, y quiere regar su árbol que está en la posición (xa, ya), con $ya > 0$. Para poder regarlo, John camina hasta el río que se encuentra en el eje x, toma agua y se dirige hasta el árbol. ¿ Cuál es la mínima distancia que tiene que recorrer granjero John para regar su árbol?

Mediana

La mediana es el número en un arreglo que tiene la mitad de números mayores y la mitad menores. La forma de determinar la mediana es simple. Basta con ordenarlos. El problema es que ordenar toma en el mejor caso $O(nlogn)$, cuando nos gustaría que fuera lineal. El ordenar hace bastante más trabajo que simplemente obtener la mediana, no nos interesa el orden relativa de los demás números en el arreglo.

Una alternativa es la siguiente. Dado un arreglo S y un valor cualquier v, siempre se puede dividir el arreglo en tres sublistas S_l, S_r y S_v, representado el conjunto de valores que son menores, mayores o iguales a v, respectivamente. Entonces la búsqueda se reduce a una de las sublistas. Esto se puede generalizar de esta manera, donde k representa el $k-$ésimo valor a seleccionar:

$$seleccion(S,k) = \begin{cases} selection(S_l,k) & \text{if } k \leq |S_l| \\ v & \text{if } |S_l| < k \leq |S_l| + |S_v| \\ selection(S_R, k - |S_l| - |S_v|) & \text{if } k > |S_l| + |S_v| \end{cases}$$

Mediana (cont.)

Las tres sublistas S_l, S_r y S_v, pueden obtenerse en forma lineal. Entonces se aplica recursividad en las sublistas. El efecto de este split es achicar el número de elementos desde S a lo más a $max(|S_l|, |S_r|)$. El algoritmo es entonces especificado excepto por el problema de escoger v, donde lo ideal es que se escoja de manera que permita achicar el arreglo sustancialmente, idealmente $|S_l|, |S_r| \sim 1/2|S|$. En ese caso:

$$T(n) = T(n/2) + O(n), T(1) = 0$$

Esto significa que justo escogemos la mediana, que es el caso óptimo. A diferencia de eso, asumimos un esquema razonablemente bueno, una elección aleatoria.

Mediana: análisis

El tiempo de ejecución depende de la elección de v. En el *peor caso*, si seleccionamos el número mayor o el número menos, tenemos que el problema solo se achica en un elemento. Esto es:

$$n + (n-1) + (n-2) + \cdots = \Theta(n^2)$$

Por el otro lado, en el *mejor caso* seleccionamos un v que divide el problema en la mitad, con un costo de $O(n)$. El promedio entonces cae entre $O(n)$ y $\Theta(n^2)$.
Distingamos un caso bueno de uno malo. Diremos que un caso bueno es uno donde el valor v cae entre un 25 a un 75 percentil del arreglo. Esto asegura que tanto S_l como S_r no pueden tener más de $3/4$ elementos de S. Esto es, un 50 % de los elementos caen entre este rango.
Dada una elección aleatoria, la pregunta entonces es ¿cuántas iteraciones en promedio necesito para tener un buen valor v? Esto se puede determinar con el siguiente razonamiento: *Sea E el número esperado de intentos antes de obtener un buen v. Ciertamente necesitamos al menos 1. Si a la primera no resulta, entonces se necesita repetir. Entonces $E = 1 + 1/2E$, lo que nos lleva a $E = 2$.*

Por lo tanto, nos queda que el tiempo esperado de $T(n)$ es $T(n) \leq T(3/4n) + O(n)$, lo cual es de orden $O(n)$.

Ordenación: MergeSort

```
     Procedure Merge_Sort (A, p, r)
1            if p < r then [
2                q := ⌊(p + r)/2⌋
3                Merge_Sort(A, p, q)
4                Merge_Sort(A, q + 1, r)
5                Merge(A, p, q, r)]
     end Merge_Sort
```

MergeSort: Mezcla

```
      Procedure Merge(A, p, qr)
1             n_1 := q − p + 1
2             n_2 := r − q
3             for i = 1 to n_1 do L[i] := A[p + i − 1]
4             for i = 1 to n_2 do R[i] := A[q + i]
5             L[n_1 + 1] = R[n_2 + 1] = ∞
6             i := j = 1
7             for k = p to r do
8                 if L[i] ≤ R[j] then[
9                     A[k] := L[i]
10                    i := i + 1]
11                else
12                    A[k] := R[j]
13                    j := j + 1]
      end Merge
```

MergeSort: Análisis

En sistema recurrente en todos los posibles casos del mergeSort es:

$$T(n) = 2T(n/2) + \Theta(n), T(1) = 0$$

Entonces en este caso el orden de complejidad queda dado por $O(nlogn)$.

Ordenación: QuickSort

La idea principal de este ordenamiento es:

- Seleccione un pivote - el primer elemento en la implementación considerada
- Reordene la lista de manera que todos los elementos en las primeras s posiciones son menores o iguales al pivote y todos los elementos en las $n - s$ posiciones son mayores o iguales que el pivote
- Intercambie el pivote con el último elemento en el primer subarreglo. El pivote queda en su posición final
- Ordene los dos subarreglos recursivamente

Ordenamiento Quicksort

```
     Procedure QuickSort (A, p, r)
1            if p < r then [
2                q := Partition(A, p, r)
3                QuickSort(A, p, q)
4                QuickSort(A, q + 1, r) ]
     end QuickSort
```

Partición del Quicksort

```
      Procedure Partition(A, p, r)
1             x := A[p]
2             i := p − 1
3             j := r + 1
4             while true do [
5                 repeat j := j − 1
6                     until A[j] ≤ x
7                 repeat i := i + 1
8                     until A[i] ≥ x
9                 if i < j then[
10                    temp := A[j]
11                    A[j] := A[i]
12                    A[i] := temp]
13                else return j ]
      end Partition
```

Quicksort: Análisis

Mejor caso (y caso promedio, divide y vencerás):

$$T(n) = 2T(n/2) + \Theta(n), T(1) = 0$$

Peor caso (problema de achica y vencerás):

$$T(n) = T(n-1) + \Theta(n), T(1) = 0$$

Multiplicación

Consideremos dos enteros x e y de enteros de n bits, y asumamos por conveniencia que n es una potencia de 2. El primer paso para hacer la multiplicación es dividir cada número en dos partes, izquierda (x_l e y_l) y derecha (x_r e y_r) con $n/2$ bits de largo. Entonces

$$x = 2^{n/2}x_l + x_r$$
$$y = 2^{n/2}y_l + y_r$$

El producto entonces puede ser expresado como:

$$xy = (2^{n/2}x_l + x_r)(2^{n/2}y_l + y_r) = 2^n x_l y_+ 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r$$

En esta expresión la suma toma un tiempo lineal, tal como la multiplicación por la potencia de dos que es solo un shift (corrimiento). Las operaciones básicas (significativas) aquí son entonces las multiplicaciones: $x_l y_l, x_l y_r, x_r y_l, x_r y_r$, lo cual puede ser realizado como 4 llamadas recursivas. Entonces, la multiplicación consiste en dividir el problema en 4 de tamaño $n/2$ y entonces evaluar la expressión de sumas de tiempo $O(n)$. En un sistema recurrente:

$$T(n) = 4T(n/2) + O(n)$$

Multiplicación (cont.)

Sin embargo, es posible reducir a tres multiplicaciones en vez de 4. Las tres multiplicaciones son $x_l y_l$ y $x_r y_r$ y $(x_l + x_r)(y_l + y_r)$, ya que $x_l y_r + x_r y_l = (x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r$. El algoritmos entonces queda dado por:

Algoritmo

multiply(x, y)
begin
  $n \leftarrow$ max(size of x, size of y);
  if $(n = 1)$ **then return** xy;
  $x_l, x_r \leftarrow$ los $\lceil n/2 \rceil$ bits más significativo y los $\lfloor n/2 \rfloor$ bit menos significativos de x;
  $y_l, y_r \leftarrow$ los $\lceil n/2 \rceil$ bits más significativo y los $\lfloor n/2 \rfloor$ bit menos significativos de y;
  $P_1 \leftarrow$ *multiply*(x_l, y_l);
  $P_2 \leftarrow$ *multiply*(x_r, y_r);
  $P_3 \leftarrow$ *multiply*$(x_l + x_r, y_l + y_r)$;
  return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$
end

Esto resulta en una recurrencia de $T(n) = 3T(n/2) + O(n)$ con un orden de complejidad de $O(n^{1,59})$.

Multiplicación de matrices

El producto de dos matrices X e Yde $n \times n$ es una tercera matriz de $n \times n$ $Z = XY$, con la entrada (i, j) tal que $Z_{i,j} = \sum_{k=1}^{n} X_{i,j} Y_{i,j}$. Esto implica un orden $O(n^3)$. Sin embargo, la multiplicación de matrices se puede dividir en subproblemas. Para ello, redefina X e Y de la siguiente forma:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Multiplicación de matrices (cont)

Entonces podemos usar dividir para conquistar donde:

$$Z = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Esto tiene un costo de $T(n) = 8T(n/2) + O(n^2)$, lo que es $O(n^3)$, igual que el algoritmo normal. Sin embargo, se puede mejorar usando la idea de la multiplicación de enteros.

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Donde:

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

Lo que tiene un costo de $T(n) = 7T(n/2) + O(n^2)$, que por el teorema maestro es $O(n^{log_2 7} \sim O(n^{2,81})$.

Exponenciación por el cuadrado

Resolver el exponencial de un número se puede resolver al aplicar en forma recursiva las siguientes fórmulas:

- Para valores pares de n:

$$a^n = (a^{n/2})^2,$$

  donde $n > 0, a^0 = 1$.

- Para valores impares de n:

$$a^n = (a^{(n-1)/2})^2 a,$$

  donde $n > 0, a^0 = 1$.

La recurrencia queda dada entonces por $T(n) = T(\lfloor n/2 \rfloor) + f(n)$, donde $f(n) \in \Theta(1)$ y $T(0) = 0$. Por teorema maestro entonces $T(n) \in \Theta(logn)$.

Achica por una constante y vencerás: ejemplos

1. Ordenamientos
2. Recorrido de un grado

Ordenamiento por Inserción

```
     Procedure Insertion_Sort (A)
1            for j = 2 to length[A] do [
2                i := j − 1
3                key := A[j]
4                while i > 0 and A[i] > key do [
5                    A[i + 1] := A[i]
6                    i := i − 1]
7                A[i + 1] := key]
     end Insertion_Sort
```

Ordenamiento por Inserción: análisis

- Eficiencia:

$$\begin{aligned} T_{peor}(n) &= n(n-1) \in \Theta(n^2) \\ T_{promedio}(n) &\sim n^2/4 \in \Theta(n^2) \\ T_{mejor}(n) &= n-1 \in \Theta(n) \end{aligned}$$

  El caso promedio se obtiene asumiendo que para cada posible clave es igualmente probable que la clave esté en cada uno de las posiciones del orden. Por ejemplo, para la cuarta clave a analizar, es igualmente probable que deba estar en la 1,2,3 o 4ta posición. Entonces el número esperado de intentos (comparaciones) para poder ubicar la clave j es: $E(j) = \frac{\sum_{i=1}^{j} i}{j} = (j+1)/2$. Conclusión, para el caso promedio sobre todas las claves se tiene $\sum_{j=2}^{n} \frac{j+1}{2} = \frac{n^2+3n-4}{4}$

- Estable
- El mejor ordenamiento elemental.

Recorrido de un grafo

Muchas algoritmos necesitan recorrer un grafo.

Procedure *DEPTH_FIRST_SEARCH*(v)
$A(v) := 1$
for each $w \in V$ adjacent to v do
    if $A(w) = 0$ then call *DEPTH_FIRST_SEARCH*(w)
end *DEPTH_FIRST_SEARCH*(v)

Procedure *BREADTH_FIRST_SEARCH*(v)
$A(v) := 1$
inicializa Q con vértice v
while Q not empty do[
    call *DELETE*(v, Q)
    for all vértices w adjacent to v do [
        if $A(w) = 0$ then [
            call *ADD*(w, Q)
            call $A(w) := 1$]]]
end *BREADTH_FIRST_SEARCH*(v)

Depth-First Search (DFS)

DFS puede ser implementado con grafos representados por:

- matrices de adyacencia: $\Theta(|V|^2)$
- lista de adyacencia: $\Theta(|V| + |E|)$

Genera dos ordenamientos distintos de vértices:

- orden en el cual los vértices son encontrados (push en la pila)
- orden en el cual los vértices son eliminados de la pila (pop de la pila cuando ya no tiene nodos adyacentes que visitar)

Breadth-First Search (BFS)

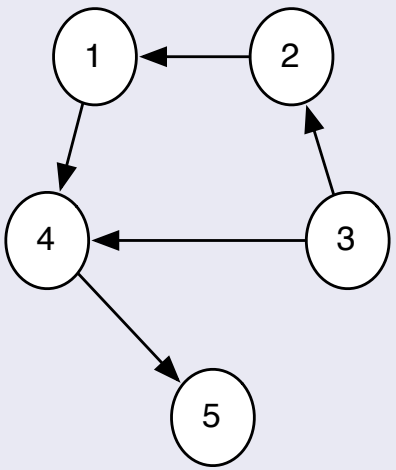
BFS tiene la misma eficiencia que DFS

- matrices de adyacencia: $\Theta(|V|^2)$
- lista de adyacencia: $\Theta(|V| + |E|)$

Genera un único orden de vértices.

Ordenamiento Topológico

Sea G un grafo dirigo y sin ciclos, G es denominado *dag*. Los vértices del dag pueden ser linealmente ordenados de manera que por cada arco, su vértice de comienzo sea listado antes que su vértice terminal. La condición de ser acíclico es condición necesaria para el ordenamiento topológico.



Ordenamiento topológico

3 2 1 4 5

Ordenamiento Topológico (cont)

- Una alternativa de algoritmo es usar el DFS algoritm. El orden reverso detecta cuando no es un dag al encontrarse con un arco entrante.
- Otra alternativa es un algoritmo de remoción de origen (source removal algorithm): En forma iterativa identificar y remover un origen (un vértice con no arco entrante) y todos los arcos incidentes a él hasta no encontrar arcos o que no exista un vértice de origen restante. Para encontrar un vértice origen invierta la lista de adyacencia por cada vértice para contar el número de arcos entrantes. Para remover un origen, decremente el contador en cada uno de sus vecinos por 1. Si al final tenemos que aún no se han recorrido todos los vértices pero no hay ninguno que sea de origen, entonces no tenemos un dag.

Algoritmo de Euclides: achica por una variable

El algoritmo de Euclides para el máximo comun divisor se basa en una aplicación recursiva de:

$$gcd(m, n) = gcd(n, m \ mod \ n)$$

Ejemplo: : $gcd(80, 44) = gcd(44, 36) = gcd(36, 8) = gcd(8, 4) = gcd(4, 0) = 4$

Un puede probar que el tamaño, medido por el primer número decrese al menos la mitad después de dos iteraciones consecutivas, por lo que $T(n) \in O(logn)$.

Demo: Asuma $m > n$,y considere m y $m \ mod \ n$:
(i) Caso 1: Si $n \leq m/2$, entonces $(m \ mod \ n) < n \leq m/2$.
(ii) Caso 2: Si $n > m/2$, entonces $(m \ mod \ n) = m - n < m/2$.

Transformar y Vencer

Un grupo de técnica que resuelven un problema por transformación son:

- Tranforma a una instancia más simple o más conveniente el mismo problema (simplificación) : pre ordenamiento
- Transforma a una representación diferente de la misma instancia (cambio de representación) : árboles de búsqueda balanceados, heap, heapsort
- Transforma un problema diferente para el cual un algoritmo existe (reducción del problema) : reducción a problemas de grafos

Pre ordenamiento

Al ordenar muchas instancias de un problemas pueden ser transformadas a una forma más simple. Algunos problemas que consideran lista son más fáciles de resolver (pero no necesariamente menos costos) cuando están ordenados:

- búsqueda
- Calcular la mediana (problema de selección que resulta ser un caso donde el pre ordenamiento no lo hace más eficiente)
- Encontrar elementos repetidos
- Convex Hull.

Eficiencia: Agrega el overhead de un preprocesamiento de costo $\Theta(nlogn)$, pero usualmente reduce el problema por al menos una base de clase de eficiencia (e.j., $\Theta(n^2) \rightarrow \Theta(n)$).

Elementos Repetidos

- Con pre ordenamiento, usa el mergesort para ordenar $\Theta(nlogn)$ y luego busca los elementos adyacente repetidos con costo $\Theta(n)$. Conclusión, el problema se resuelve con orden $\Theta(nlogn)$.
- Por fuerza bruta, se comparan cada elemento, con un orden de $\Theta(n^2)$.
- En este caso, pre ordenamiento lleva a una mayor eficiencia.

Búsqueda

- La fuerza bruta para m búsquedas en n elementos es $\Theta(n \times m)$
- Preordenadas m búsqueda en n elementos es $O(nlogn + mlogn)$.

Arboles

- La búsqueda, inserción y eliminación en un árbol de búsqueda balanceado es $\Theta(logn)$, y no balanceado $\Theta(n)$.
- Instancias simplificadas: AVL y árboles red and black restringen el desbalance haciendo rotaciones.
- Los AVL y Red and Black son casos en que se simplifica el balanceo.
- Cambio de representación: B-trees tienden a un balanceo perfecto permitiendo más de un elemento en un nodo. Un heapsort es un árbol binario completo donde el padre es mayor a cualquiera de sus hijos en el árbol. Entonces, un subárbol de un heap es también un heap.
- Tanto AVL, Red and Black, Btree y Heapsort fueron vistos en estructuras de datos, material al cual se hace referencia para su análisis.

Cambio de representación: Regla de Horner

La evaluación de un polinomio de la forma:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

se puede formular como

$$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots)x + a_0$$

Un ejemplo es el siguiente:

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 = x(2x^3 - x^2 * 3x + 1) - 5 \\ &= x(x(2x^3 - x^2 * 3) + 1) - 5 = x(x(x(2x - 1) + 3) + 1) - 5 \end{aligned}$$

Regla de Horner: Algoritm

Algoritmo

$p \leftarrow P[n]$;
for $i \leftarrow n-1$ **to** 0 **do** ;
$p \leftarrow x * p + P[i]$;
endfor
return p **end**

Eficiencia, el número de multiplicaciones = número de sumas = $\Theta(n)$

Cómputo de a^n

Para este problema usar la exponenciación binaria de izquierda a derecha que sigue el siguiente procedimiento:

- Inicialize el producto acumulador en 1.
- Recorra los dígitos binarios de la exponenciación de izquierda a derecha. Si el dígito binario actual es cero, aplique el cuadrado al acumulador (S), si es 1, aplique el cuadrado del acumulador y multiplique por a (SM).

Ejemplo: Sea a^{13}, donde $n = 13 = 1101$. Entonces, las iteraciones son:

1	1	0	1
$1^2 a = a$	$(a^2)a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 a = a^{13}$

Eficiencia: $(b-1) \leq T(n) \leq 2(b-1)$, donde $b = \lfloor log_2 n \rfloor + 1$

Reducción de problema

- Si necesitas resolver un problema, redúcelo a uno que conozcas como resolver.
- Se usa en teoría de complejidad para clasificar problemas.
- Para que sea de valor práctico, el tiempo combinado de la transformación y la resolcuión del otro problema debe menor que el tiempo de resolver el problema por otro método.
- Ejemplo 1: el mínimo común múltiplo (LCM) de dos positivos números m y n es el entero más pequeño divisible (MCM) por ambos m y n. El problema de reducción entonces es: $MCM(m, n) = m * n/LCM(m, n)$.
- Ejemplo 2: El problema de maximización se puede ver como el converso de la minimización : $minf(x) = -max[-f(x)]$

Reducción a problemas de grafos

- Algoritmos tales como el DFS o BFS pueden ser usados al hacer la reducción a una representación de grafos.
- Grafos de estado son usados con mucha frecuencia en inteligencia artificial. En estos grafos los vértices representan estados y los arcos representan transiciones válidas entre estados. Usualmente se usa un vértice original y uno de destino.