Diseño de Algoritmos - Algoritmos II

(notas basadas en el material adicional de "The Design & Analysis of Algoritmos" (Levitin 2003))

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 4: La Estrategia Decrease & Conquer

Decrease & Conquer

La estrategia de diseño de algoritmos Decrease & Conquer apunta a resolver problemas algorítmicamente de la siguiente forma:

Reducir instancias del problema a instancias más pequeñas del mismo problema

Resolver las instancias más pequeñas

Extender las soluciones de instancias más pequeñas para conseguir una solución a la instancia original

Tipos de Decrease & Conquer

De acuerdo a la forma de "decrementar" un problema, existen diferentes tipos de Decrease & Conquer

- **Decremento por una constante** (usualmente 1):
  - insertion sort / selection sort
  - algoritmos de recorrido de grafos (DFS and BFS)
  - ordenamiento topológico
  - algoritmos para generar permutaciones, subconjuntos, ...
- **Decremento por un factor constante** (usualmente la mitad)
  - búsqueda binaria
  - exponenciación por cuadrados
- **Decremento de tamaño variable**
  - Algoritmo de Euclides

Un Ejemplo: Exponenciación

Problema: Computar a^n, donde n es un entero no negativo.

Solución: Para computar a^n, con n > 0, podemos computar a^{n-1}, y multiplicar el resultado por a.

```
exp a 0 = 1

exp a n = (exp a (n-1)) * a
```

Un Ejemplo: Insertion Sort

Para ordenar un arreglo A[0..n-1], ordenar primero A[0..n-2] y luego insertar A[n-1] en el lugar que le corresponde dentro del segmento ya ordenado A[0..n-2]

Ejemplo: ordenar 6, 4, 1, 8, 5

6 | 4 1 8 5

4 6 | 1 8 5

1 4 6 | 8 5

1 4 6 8 | 5

1 4 5 6 8

Pseudo-código de Insertion Sort

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Recorridos en Grafos

Muchos problemas requieren representar datos como grafos (relaciones), y utilizan recorridos de grafos para procesar información de forma sistemática

Algoritmos básicos de recorrido de grafos:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Depth-First Search (DFS)

Visita los vértices del grafo moviéndose desde el último vértice visitado hacia uno no visitado, y dando marcha atrás (backtracking) si no hay vértices no visitados disponibles.

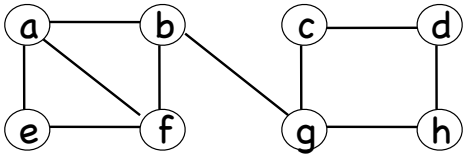
Para su implementación, se puede utilizar una pila

Depth First Search genera un "árbol" de visita a partir de un grafo

Pseudo-código de DFS

```
DFS(G,v)   ( v is the vertex where the search starts )
   Stack S := {};   ( start with an empty stack )
   for each vertex u, set visited[u] := false;
   push S, v;
   while (S is not empty) do
      u := pop S;
      if (not visited[u]) then
         visited[u] := true;
         for each unvisited neighbour w of u
            push S, w;
      end if
   end while
END DFS()
```

Ejemplo: DFS sobre grafo no dirigido

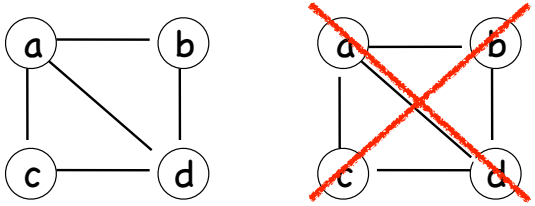


Pila DFS:

Árbol DFS:

DAGs y Ordenamiento Topológico

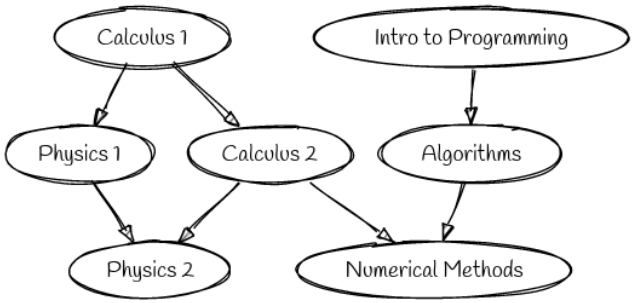
Dag: grafo dirigido acíclico (directed acyclic graph), i.e., un grafo dirigido sin ciclos (dirigidos)



Surgen en el modelado de muchos problemas que involucran la noción de pre-requisito (e.g., control de versiones)

Los vértices de un dag pueden ordenarse de manera tal que para cada arco su vértice de origen se liste antes que su vértice destino (***ordenamiento topológico***). El grafo debe ser un dag para que este proceso sea posible.

Ejemplo de Ordenamiento Topológico



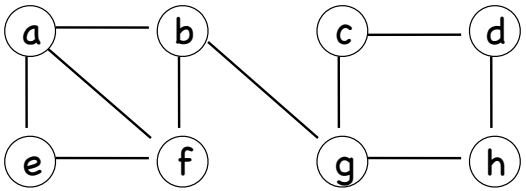
¿En qué orden puedo cursar cumpliendo con las correlatividades?

Algoritmo de Eliminación de Orígenes

Algoritmo de Eliminación de Orígenes

Iterativamente, identificar y eliminar los "orígenes" (vértices sin arcos entrantes), y todos los arcos que de ellos salen, hasta que ya no queden vértices (problema resuelto) o hasta que haya vértices pero no orígenes (no es un dag)

Ejemplo:



Algoritmos de Decremento por Factor Constante

En esta variante de Decrease & Conquer, el tamaño de instancia de problema se reduce en un mismo factor en cada caso (usualmente, 2)

Ejemplos:

Búsqueda binaria

Exponenciación mediante cuadrados

Problema de la moneda falsa

Exponenciación por Cuadrados

El Problema: Computar a^n donde n es un entero no negativo

El problema se puede resolver aplicando recursivamente las siguientes fórmulas:

Para valores pares de n $\quad a^n = (a^{n/2})^2$ si $n > 0$ y $a^0 = 1$

Para valores impares de n $\quad a^n = (a^{(n-1)/2})^2 a$

Recurrencia: $M(n) = M(\lfloor n/2 \rfloor) + f(n)$, donde $f(n) = 1$ o 2,

$M(0) = 0$

Teorema Maestro: $M(n) \in \Theta(\log n)$

Multiplicación por Duplicación

El problema: Computar el producto de dos enteros positivos

Se puede resolver mediante un decremento a la mitad, usando las siguientes fórmulas:

Para valores pares de n: $\quad n * m = \frac{n}{2} * 2m$

Para valores impares de n:

$$n * m = \frac{n-1}{2} * 2m + m \text{ si } n > 1 \text{ y } m \text{ si } n = 1$$

Algoritmos de Decremento Variable

En esta variante de Decrease & Conquer se reduce cada instancia del problema en tamaños variables de un paso a otro

Ejemplos:

- Algoritmo de Euclides
- Búsqueda por Interpolación
- Algoritmos sobre árboles binarios de búsqueda (no balanceados)

Algoritmo de Euclides

El Algoritmo de Euclides se basa en la aplicación iterativa de la siguiente igualdad

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

Ej.: gcd(80,44) = gcd(44,36) = gcd(36, 12) = gcd(12,0) = 12

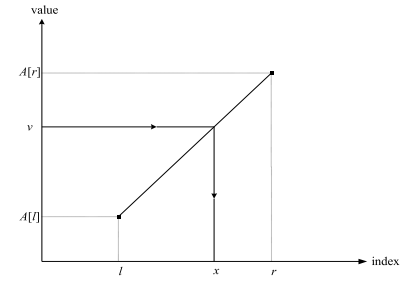
Se puede demostrar que el tamaño, medido como el segundo valor, decrece al menos a la mitad en dos iteraciones consecutivas.

Luego, $T(n) \in O(\log n)$

Búsqueda por Interpolación

Es una búsqueda en una secuencia ordenada A[l..r], similar a la búsqueda binaria, pero estimando la ubicación de la clave buscada en el segmento a explorar, usando su valor.

Más precisamente, se supone que los valores del arreglo crecen linealmente, y la ubicación de la clave v a buscar se estima como el valor de la coordenada x del punto correspondiente a v, en el segmento que une (l, A[l]) con (r, A[r]), y cuya coordenada x es v:



$$x = l + \lfloor (v - \mathrm{A}[l])(r - l)/(\mathrm{A}[r] - \mathrm{A}[l]) \rfloor$$

Análisis de la Búsqueda por Interpolación

- Eficiencia

  caso promedio: $C(n) < \log_2 \log_2 n + 1$

  peor caso: $C(n) = n$

- Es preferible a la búsqueda binaria sólo para búsquedas sobre espacios muy grandes, o donde las comparaciones son costosas