Algoritmos y Estructuras de Datos III

1er cuatrimestre 2015

Min Chih Lin

Irene Loiseau

Observaciones:

- *Estas transparencias pueden cambiar un poco antes o después de la clase correspondiente a cada tema.*

- *Las presentaciones* **NO INCLUYEN** *la mayoría de las demostraciones que* **TAMBIEN** *forman parte del curso y que son dadas en clase. En la mayoría de los casos se incluyen sólo los enunciados de los teoremas.* (o sea esto NO es un apunte de la materia!!).

- *También puede haber algún ejemplo o resultado que se de en clase y no esté las transparencias.*

.

PROGRAMA

1. ALGORITMOS:

Definición de algoritmo. Modelos de computación: modelo RAM, Máquina de Turing. Complejidad, definición, complejidad en el peor caso, en el caso promedio. Algoritmos de tiempo polinomial y no polinomial. Límite inferior. Ejemplo: análisis de algoritmos de ordenamiento. Algoritmos recursivos. Análisis de la complejidad de algoritmos recursivos.

Técnicas de diseño de algoritmos: dividir y conquistar, backtracking, algoritmos golosos, programación dinámica.

Algoritmos probabilísticos. Algoritmos aproximados Algoritmos heurísticos: ejemplos. Metaheurísticas. Nociones de evaluación de heurísticas.

2. GRAFOS:

Definiciones básicas. Adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc. Grafos eulerianos y hamiltonianos. Grafos bipartitos. Arboles: caracterización, árboles orientados, árbol generador. Enumeración. Planaridad. Coloreo. Número cromático. Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

3. ALGORITMOS EN GRAFOS Y APLICACIONES:

Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas.

Algoritmos de búsqueda en grafos: BFS, DFS, A^*. Mínimo árbol generador, algoritmos de Prim y Kruskal. Arboles ordenados: códigos unívocamente descifrables. Algoritmos para detección de circuitos. Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Floyd, Dantzig. Planificación de procesos: PERT/CPM.

. Heurísticas para el problema del viajante de comercio. Algoritmos para determinar si un grafo es planar. Algoritmos para coloreo de grafos.

Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson. Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

4. PROBLEMAS NP-COMPLETOS:

Problemas tratables e intratables. Problemas de decisión. P y NP. Maquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP. Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.

BIBLIOGRAFIA

BIBLIOGRAFÍA BÁSICA :

- Brassard,G., Bratley,P.”Fundamental of Algorithmics”, Prentice Hall,1996.
- Cormen, T.,Leiserson, C.,Rivest,R.,Stein, C.,”Introduction to Algorithms”, The MIT Press, McGraw-Hill,2001.
- Garey M.R. and Johnson D.S.: “Computers and intractability: a guide to the theory of NP- Completeness”, W. Freeman and Co., 1979.
- Gross,J., and Yellen, J. ,”Graph theory and its applications”, CRC, 1999
- Harary F., “Graph theory”, Addison-Wesley, 1969.

BIBLIOGRAFÍA DE CONSULTA : *ver en la página WEB de la materia y en el el archivo de las prácticas.*

ALGORITMOS

- *Qué es un algoritmo ?*
- *Qué es un buen algoritmo?*
- *Dados dos algoritmos para resolver un mismo problema, cuál es mejor?*
- *Cuándo un problema está bien resuelto?*
- *Cómo se hace para inventar un nuevo algoritmo?*

Qué es un algoritmo?

Noción "informal":

Un algoritmo es una sucesión finita de instrucciones "bien definidas" tal que:

i) No hay ambigüedad en las instrucciones.

ii) Después de ejecutar una instrucción no hay ambigüedad respecto de cual es la instrucción que debe ejecutarse a continuación.

iii) Después de un número finito de instrucciones ejecutadas se llega siempre a la instrucción STOP ("Un algoritmo siempre para").

PROBLEMA

- ***instancia de un problema***
- ***datos de entrada de una instancia (E)***
- ***solución (S)***

ALGORITMO:

- ***técnica para la resolución de un problema***
- ***función f tal que f (E) = S***

Problema de Collatz

Paso 1: z número entero positivo

Paso 2: mientras z ≠ 1 hacer

Paso 3: Si z es par poner z = : z / 2

En caso contrario poner z =: 3 z +1

Paso 4: parar

==========

es esto un algoritmo?

- *Cuánto tarda un algoritmo en resolver un problema?*

- *Cuándo un algoritmo que resuelve un problema es mejor que otro que resuelve el mismo problema?*

Complejidad Computacional

Complejidad

La complejidad de un algoritmo es una función que calcula el tiempo de ejecución en función del tamaño de la entrada de un problema.

- **Historia**

- Complejidad en el peor caso (*es siempre significativo el peor caso?)*
- Complejidad en el caso promedio (*porqué no usamos este enfoque siempre?)*

Cómo medir el tiempo de ejecución en función del tamaño de la entrada de los datos del problema?

MODELOS DE COMPUTACION

- RAM
- MAQUINA DE TURING

Maquina RAM (RANDOM ACCESS MACHINE)

Es el modelo que usamos implícitamente en la práctica para evaluar la complejidad de un algoritmo.

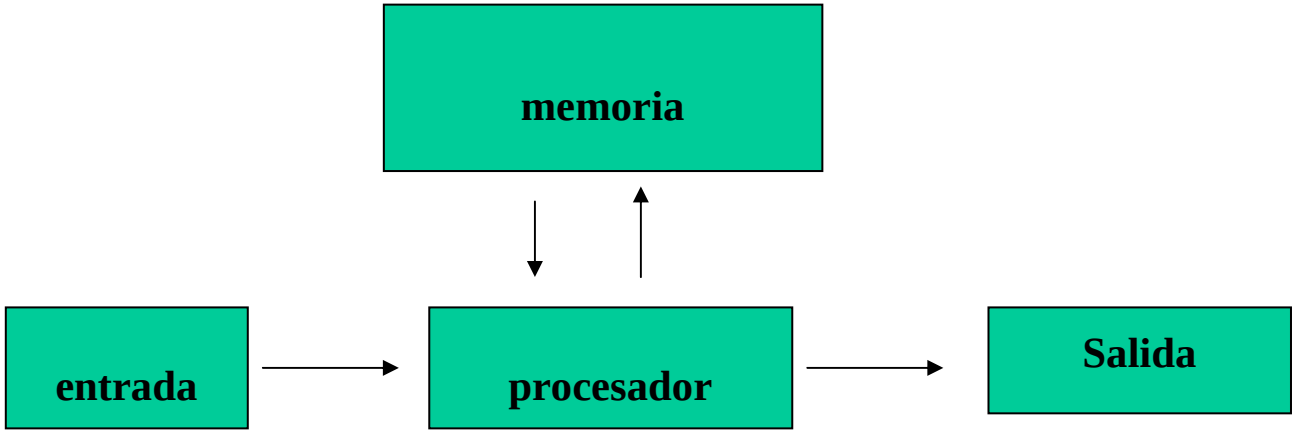
==========

- unidad de memoria
- unidad de entrada
- procesador (*programa*)
- unidad de salida
- conjunto de instrucciones

==========

Un programa RAM es una secuencia finita de estas instrucciones

Esquema de una máquina RAM



En este modelo suponemos que:

- La unidad de entrada es una cinta de registros en cada uno de los cuales se puede almacenar un entero.
- La memoria es una sucesión de registros, cada uno de los cuales puede almacenar un entero de tamaño arbitrario. Los cálculos se hacen en el primero de ellos, $r_{0.}$
- El programa no se almacena en en la memoria y no se modifica a si mismo.
- El conjunto de instrucciones es similar al de las *computadoras reales.* Cada instrucción tiene un nombre y una dirección.

- Este modelo sirve para modelar situaciones donde tenemos memoria suficiente para almacenar el problema y donde los enteros que se usan en los cálculos entran en una palabra.

Ejemplo

(sacado del libro de Aho, Hopcroft y Ullmann)

Ejemplo de instrucciones de una máquina RAM

- LOAD operando - Carga un valor en el acumulador
- STORE operando - Carga el acumulador en un registro
- ADD operando - Suma el operando al acumulador
- SUB operando - Resta el operando al acumulador
- MULT operando - Multiplica el operando por el acumulador
- DIV operando - Divide el acumulador por el operando
- READ operando - Lee un nuevo dato de entrada ! operando
- WRITE operando - Escribe el operando a la salida
- JUMP label - Salto incondicional
- JGTZ label - Salta si el acumulador es positivo
- JZERO label - Salta si el acumulador es cero
- HALT - Termina el programa

Los operandos pueden ser:

- LOAD = a: Carga en el acumulador el entero a.
- LOAD i : Carga en el acumulador el contenido del registro i .
- LOAD *i : Carga en el acumulador el contenido del registro indexado por el valor del registro i .

Ejemplo de un programa para calcula n^n en esta máquina RAM

```
====================
        READ     1                  leer r1
          LOAD     1                si r1 ≤ 0  entonces write 0
          JGTZ     pos                  "
          WRITE    = 0                  "
          JUMP     endif
pos       LOAD     1                sino r2 ←--  r1
          STORE    2                    "
          LOAD     1                    r3 ←--  r1 -1 .
          SUB      = 1                  "
          STORE    3                    "
while     LOAD     3                mientras r3 > 0 hacer .
          JGTZ     continue             "
          JUMP     endwhile             "
continue  LOAD     2                r2 ←-- r2 * r1 .
          MULT     1                    "
          STORE    2                    "
          LOAD     3                r2 ←-- r2 * r1
          SUB      = 1                  "
          STORE    3                    "
          JUMP     while
endwhile  WRITE    2                write r2
endif     HALT
====================================
```

Cómo calculamos la complejidad de un algoritmo con este modelo?

t_j : tiempo de ejecución de la instrucción j.

T: tiempo de ejecución de un programa que ejecuta n_j instrucciones de tipo j.

$$T = \sum_j n_j \, t_j$$

Si suponemos que todos los t_j son iguales tenemos

$$T = \sum_j n_j$$

Esta es el modelo que usamos implícitamente cuando "calculamos" la complejidad de un algoritmo en la práctica

Tamaño de la entrada de un problema

- Número de símbolos de un alfabeto finito necesarios para codificar todos los datos de una instancia de un problema.
- El tamaño de la entrada de un problema depende de la base o alfabeto elegidos:

- para almacenar un entero positivo N en base 2 se necesitan $L = \lceil \log_2(N+1) \rceil$ dígitos binarios.
- en una base b cualquiera se necesitan $L = \lceil \log_b(N+1) \rceil$
- si a y b $\neq 1$ entonces $\log_b N = \log_a N / \log_a b$

qué implica esto desde el punto de vista de la complejidad de un algoritmo?

Podemos considerar distintos modelos para determinar la complejidad en función del tamaño de la entrada del problema.

- **Modelo uniforme:** suponemos que los valores a almacenar están acotados.

- **Modelo logarítmico:** consideramos la representación binaria de los números a almacenar, o sea medimos el tamaño de la entrada en bits.

Qué consecuencias puede tener usar uno u otro modelo?

Máquina de Turing *(determinística)*

Componentes de una máquina de Turing

==========

- Cinta infinita dividida en celdas iguales que pueden contener un único símbolo de un alfabeto finito T.
- Cabeza de lectura escritura que apunta a una de las celdas.
- Lista finita de estados posibles $\{q_i\}$. Hay al menos un estado final.
- Operaciones:

  i) lectura del símbolo t_i inscripto en la celda señalada por la cabeza.

  ii) guardar en esta celda el símbolo t_f determinado por q_i.

  iii) transición al estado q_f determinado por t_i y q_i .

  iv) movimiento de la cabeza de lectura/escritura a izquierda o derecha.

  V) inicio de un nuevo ciclo.

Para cuando llega a un estado final.

==========

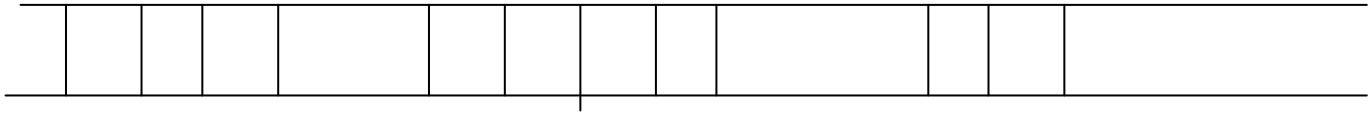
Un programa correspondiente a esta máquina de Turing es un conjunto finito de quintuplas (q_i,t_i,q_s, t_s, m).

La instrucción (q_i,t_i,q_s, t_s, m) se interpreta como:

" si la máquina está en el estado q_i y lee en la posición actual de la cabeza de lectora el símbolo del alfabeto t_i , entonces escribe en dicha posición el símbolo t_s , pasa al estado q_s y se mueve a la izquierda o a la derecha según el valor de m sea -1 ó +1 ".

Ejemplo

- Máquina de Turing para resolver el problema de determinar si dados N y M con M > 1, N es divisible por M.



- Alfabeto T = { A, B, 0, 1, 2 }
- Se almacenan en la cinta M y N codificados en base 1, separados por el símbolo B y con el símbolo A marcando el fin del número.
- Estado inicial: q_0
- Estados posibles Q = {q_0, q_1 ,q_2,q_s, q_n}
- Estados finales: $q_{s,}$ q_n

Programa

Instrucciones

$(q_0, A, q_2, A, +1)$	$(q_1, A, q_n, *, *)$	$(q_2, A, q_s, *, *)$
$(q_0, B, q_0, B, -1)$	$(q_1, B, q_1, B, +1)$	$(q_2, B, q_2, B, +1)$
$(q_0, 0, q_0, 0, -1)$	$(q_1, 0, q_1, 0, +1)$	$(q_2, 0, q_2, 0, +1)$
$(q_0, 1, q_1, 2, +1)$	$(q_1, 1, q_0, 0, -1)$	$(q_2, 1, q_0, 1, -1)$
$(q_0, 2, q_0, 2, -1)$	$(q_1, 2, q_1, 2, +1)$	$(q_2, 2, q_2, 1, +1)$

Cómo funciona esto?.

Usar esta máquina para decidir si 5 es divisible por 3.

- ***Cómo se calcula la complejidad de un algoritmo representado por una máquina de Turing?.***

- Formalmente un algoritmo es un objeto asociado a una máquina de Turing que resuelve un determinado problema. La complejidad del mismo debería ser calculada en función del mismo.

- En la práctica sin embargo usaremos una noción un poco menos formal de algoritmo y también para determinar la complejidad de un algoritmo, que se parece más al modelo RAM.

- Desde el punto de vista práctico ambos enfoques han mostrado ser equivalentes.

Determinaremos la complejidad contando la cantidad de operaciones básicas de alto nivel que realiza el algoritmo, en función del tamaño del problema.

Es correcto esto?. Puede ser peligroso?

Cuándo decimos que un algoritmo es bueno?

Armemos esta tabla..... *supongamos que tenemos algoritmos con las siguientes complejidades y los corremos en la misma máquina*
(los tiempos están dados en segundos)

	10	**20**	**30**	**40**	**50**	**60**
n	0.00001	0.00002	0.00003	0.00004	0.00005	0.00006
n^2						
n^3						
n^5						
2^n						
3^n						

La tabla queda así…..

(los tiempos están dados en segundos salvo cuando dice otra cosa)

	10	**20**	**30**	**40**	**50**	**60**
n	0.00001	0.00002	0.00003	0.00004	0.00005	0.00006
$\mathbf{n^2}$	0.0001	0.0004	0.0009	0.0016	90.0025	0.0036
$\mathbf{n^3}$	0.001	0.008	0.027	0.064	0.125	0.216
$\mathbf{n^5}$	0.1	3.2	24.3	1.7 min	5.2 min	13.0
$\mathbf{2^n}$	0.001	1.0	17.9min	12.7días	35.6 años	366 siglos
$\mathbf{3^n}$	0.59	58 min	6.5años	3855siglos	2 E+8 siglos	1.3 E+13 siglos

Los datos de la tabla anterior corresponden a una máquina MUY vieja (son datos del libro de Garey y Johnson de 1979).

Qué pasa si tenemos una máquina 1000 veces más rápida?. Un millón de veces más rápida?.
Cuál sería el tamaño del problema que podemos resolver en una hora comparado con el problema que podemos resolver ahora?.

	actual	**computadora 1000 veces mas rápida**
n	N_1	1000 N_1
$\mathbf{n^2}$	N_2	31.6 N_2
$\mathbf{n^3}$	N_3	10 N_3
$\mathbf{n^5}$	N_4	3.98 N_4
$\mathbf{2^n}$	N_5	N_5 + 9.97
$\mathbf{3^n}$	N_6	N_6 + 6.29

Qué pasaría si tuviéramos una computadora 1 millón de veces más rápida?

Cuándo diremos entonces que un algoritmo es bueno?.

Cuándo un algoritmo es suficientemente eficiente para ser usado en la práctica?

==========

POLINOMIAL = “bueno”

EXPONENCIAL = “malo”

==========

Qué pasa si tengo complejidades como las siguientes?:

- n^{80}
- 1.001^{n}

esto no ocurre en la práctica

Puede ocurrir que un algoritmo sea "malo" en el peor caso y bueno en la práctica?.

Hay muy pocos casos.

Ejemplo clásico:

método simplex para problemas de programación lineal.

En el peor caso es exponencial pero en la práctica el tiempo de ejecución es generalmente $O(m^3)$ donde m es el número de ecuaciones del problema de programación lineal (es un método MUY usado en la industria y los servicios para resolver problemas de optimización…desde hace 60 años)

Cuándo decimos que un problema está computacionalmente bien resuelto?

Cuándo hay un algoritmo polinomial para resolverlo.

Como veremos más adelante hay problemas para los cuales no sabemos si existe o no un algoritmo polinomial para resolverlos. Saber si una gran familia de estos problemas tiene solución polinomial o no es el problema abierto más importante de teoría de la computación.....

continuará en el último capítulo del curso.....

Repasando notaciones

Dadas dos funciones f y g : $N_+ \rightarrow R$ decimos que:

- $f(n) = O(g(n))$ si $\exists\ c > 0$ y $n_0 \in N$ tal que $f(n) \leq c\ g(n)$ $\forall\ n > n_0$.
- $f(n) = \Omega(g(n))$ si $\exists\ c > 0$ y $n_0 \in N$ tal que $f(n) > c\ g(n)$ $\forall\ n > n_0$.
- $f(n) = \Theta(g(n))$ si $\exists\ c, c' > 0$ y $n_0 \in N$ tal que

$$c\ g(n) \leq f(n) \leq c'\ g(n) \qquad \forall\ n > n_0$$

Si $f(n) = O(g(n))$ se dice que f es de orden $g(n)$.

Complejidad de algunos algoritmos conocidos

- Busqueda secuencial: $O(n)$
- Busqueda binaria: $O(\log(n))$
- Bubblesort: $O(n^2)$
- Quicksort: $O(n^2)$ en el peor caso, $O(n \log(n))$ en el caso promedio.
- Heapsort: $O(n \log(n))$

Cálculo de determinantes

Sea una matriz $M = (a_{ij})$ y sea M_{ij} la submatriz de M que se obtiene sacando de M la fila i y la columna j.

Fórmula recursiva para calcular el determinante:

(desarrollo por la primera fila)

$$\det(M) = \sum_j (-1)^{j+1} a_{ij} \det(M_{1j})$$

Complejidad de este método?.

Se puede calcular usando la fórmula recursiva que se deduce de la versión recursiva que dimos del algoritmo:

$$t(n) = n\, t(n - 1) + O(n)$$

Complejidad O(n!)

Hay métodos más eficientes de calcular el determinante de una matriz?.

SI (*triangular la matriz por el método de Gauss y calcular el producto de la diagonal)*

Está este problema bien resuelto computacionalmente?.

Búsqueda secuencial

Datos: n, b, $(a_1, a_2, \ldots\ldots\ldots\ldots a_n)$

Paso 1: para $i = 1,n$ hacer

if $b = a_i$ parar

Paso 2 : parar

==========

Complejidad?

Búsqueda Binaria

```
Function Bin (T,x)
si n = 0 o x < T entonces return
i ← 1
j ← n
mientras i < j hacer
        k ← ⌊ (i+j+1)/2⌋
        si x < T (k) entonces j ← k-1
        sino i ← k
return i
==========================
```

Complejidad?

Algoritmos de ordenamiento

Algoritmo de burbujeo

Datos: $(a_1, a_2, \ldots\ldots\ldots\ldots\ldots a_n)$

Paso 0: poner $k = n$

Paso 1: mientras $k \geq 1$ hacer

Paso 2: poner $i = 1$

Paso 3: mientras $i \leq k$ hacer

Paso 4 : si $a_i > a_{i+1}$ cambiar a_i con a_{i+1}

Paso 5: poner $i = i + 1$

Paso 6 : poner $k = k - 1$

Paso 7 : parar

==========

Complejidad?

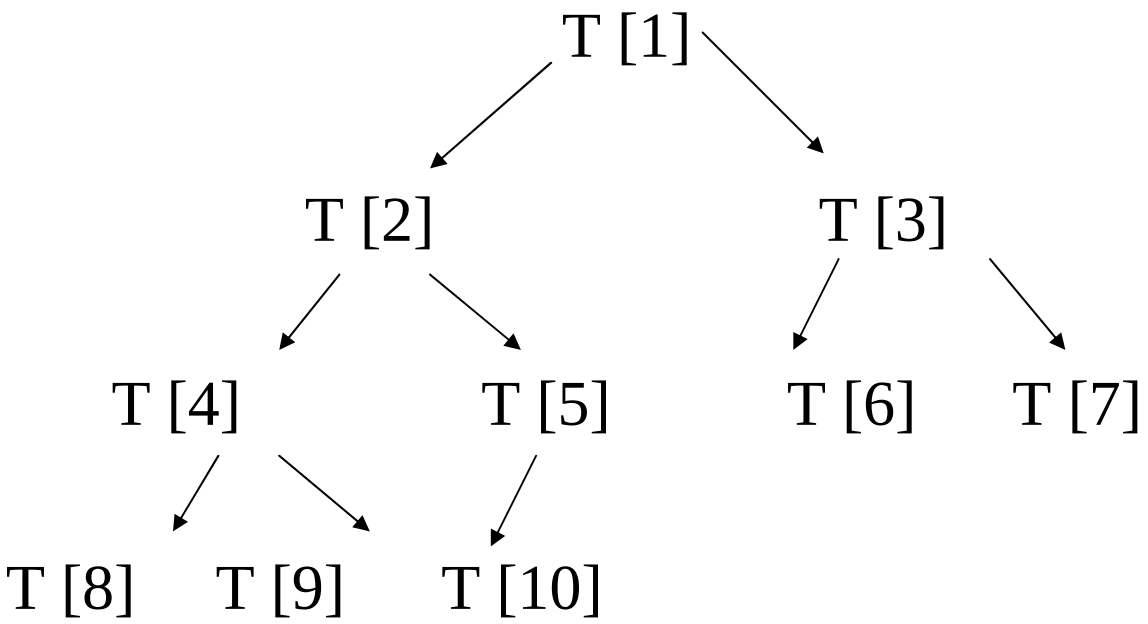
Heap sort

Heap: árbol binario casi completo, donde cada nodo tiene asignado un valor de forma que el valor de un nodo es mejor *(mayor)* que el de sus hijos.

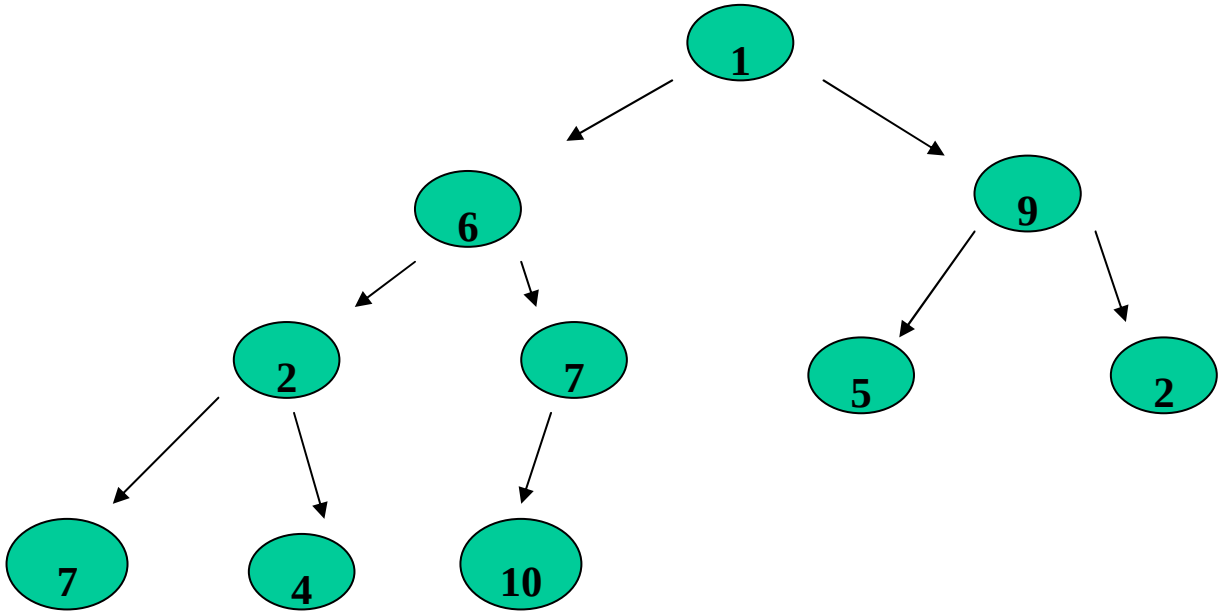
Puede ser representado por un arreglo T en el cual los nodos del nivel k se guardan en las posiciones

$$T[2^k], T[2^k+1], \ldots\ldots\ldots\ldots T[2^{k+1}-1]$$

ejemplo



Ordenar T = [1, 6, 9, 2, 7, 5, 2, 7, 4, 10] usando Heapsort.



Algoritmo Make-Heap

Dado un arreglo T los siguientes procedimientos nos permiten ordenarlo:

procedure sift-down(T [1..n], i)

("baja" el nodo i hasta que T [1..n] vuelve a ser un heap, suponiendo que T sea suficientemente grande para ser un heap (caso base?)

k ← i

repetir mientras j ≠ k

j ← k

if 2j ≤ n y **T** [2j] > **T** [k] entonces k ← 2j

if 2j < n y **T** [2j +1] > **T** [k] entonces k ← 2j + 1

intercambiar T [j] y T [k]

fin

procedure make-heap (T [1..n])

(arma el heap)

para i = ⌊n /2⌋ ....1 hacer sift-down (T,i)

fin

procedure heapsort ((T [1..n])

(ordena T)

make-heap (T)

para i = n , 2 hacer

    intercambiar T **[1] y** T **[i]**

    sift-down(T [1..i-1], 1)

fin

Límite inferior para la complejidad de un algoritmo

- *Cuándo se puede obtener?*
- *Importancia*

EJEMPLO: *algoritmos de ordenamiento basados en comparaciones. Se pueden modelar por un árbol binario de decisión. Los estados finales del algoritmos están representados en las hojas.*

Entonces nro de hojas $\geq n!$ Como el árbol es binario, nro de hojas $\leq 2^h$ (h altura del árbol) y $2^h \geq n!$. Entonces

$h \geq \log_2 (n!) = \log_2 (n\ (n-1)....2.\ 1) > \log_2 (n\ (n-1)....\lceil n/2 \rceil) > \log_2 n\ /2^{\ (n/2)} =$

$= n/2\ (\log_2 n - 1)\ ) = \Omega\ (n \log_2 n\ )$

Conclusión: HEAPSORT es “óptimo” dentro de este tipo de algoritmos.

Técnicas de diseño de algoritmos

- Dividir y conquistar
- Recursividad
- Algoritmos golosos
- Programación dinámica
- Backtracking
- Algoritmos Probabilísticos

El algoritmo se puede mejorar un poco, cambiando la forma de verificar cuando un conjunto de tareas es factible (ver libro de Brassard y Bratley).

Dividir y conquistar

- Si la instancia I es pequeña, entonces utilizar un algoritmo ad-hoc para resolver el problema.

- En caso contrario:

  i) *Dividir I en sub-instancias I1; I2; : : : ; Ik más pequeñas.*

  ii) Resolver recursivamente las k sub- instancias.

  iii) Combinar las soluciones para las k sub-instancias para obtener una solución para la instancia original I .

Dividir y conquistar

ESQUEMA GENERAL:

Función DQ (x)

Si x es suficientemente chico o simple entonces return
ADHOC(x)

Sino, descomponer x en subinstancias $x_1 \ldots\ldots x_k$.
Para i=1,k hacer $y_i = DQ\ (x_i)$
Combinar las soluciones y_i *para construir una solución y para x*

Return y

Recursividad, Caso Base

Ejemplos:

- Búsqueda binaria
- Merge Sort
- Quick Sort
- Algoritmo de multiplicación de Strassen

Cómo calcular la complejidad de estos algoritmos?.

Ecuaciones de recurrencia.

Ejemplo: Mergesort

Algoritmo divide and conquer para ordenar un arreglo A de n elementos (von Neumann, 1945).

- Si n es pequeño, ordenar por cualquier método sencillo.
- Si n es grande:

    -- A1 := primera mitad de A.
    -- A2 := segunda mitad de A.
    -- Ordenar recursivamente A1 y A2 por separado.
    -- Combinar A1 y A2 para obtener los elementos de A ordenados.

- Este algoritmo contiene todos los elementos típicos de la técnica divide and conquer.

Complejidad de MergeSort

Cómo calcular t(n), complejidad de ordenar un arreglo de longitud n?

Podemos plantear la siguiente formula recursiva para la complejidad:

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + O(n)$$

entonces

$$t(n) \text{ es } O(n \log n)$$

Algoritmo de multiplicación de Strassen

Cómo calcular el producto de dos matrices de dos por dos usando menos multiplicaciones que con el método tradicional.

Dadas dos matrices

$$A = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \qquad \text{y} \qquad B = \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix}$$

podemos poner……

$$m_1 = (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11})$$
$$m_2 = a_{11}\, b_{11}$$
$$m_3 = a_{12}\, b_{21}$$
$$m_4 = (a_{11} - a_{21})(b_{22} - b_{12})$$
$$m_5 = (a_{21} + a_{22})(b_{12} - b_{11})$$
$$m_6 = (a_{12} - a_{21} + a_{11} - a_{22})\, b_{22}$$
$$m_7 = a_{22}(b_{11} + b_{22} - b_{12} - b_{21})$$

Entonces el producto AB queda:

$$\begin{vmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{vmatrix}$$

Este procedimiento requiere 7 multiplicaciones para cacular el producto de A y B *(pero más sumas que el método tradicional!!).*

- Si reemplazamos cada elemento de A y B por una matriz de n x n, las fórmulas anteriores nos dan una forma de multiplicar dos 2n X 2n matrices.

A partir de esto tenemos un método recursivo para calcular el producto de matrices con n potencia de 2.

- *Cómo se calcula para n par, la complejidad t(n) de este algoritmo?.*

$$t(n) = 7\, t(n/2) + g(n) \qquad \text{con } g(n) \in O(n^2)$$

Se puede demostrar que $t(n) \in \Theta(n^{\log 7})$ y por lo tanto $t(n) \in O(n^{2.81})$. (ejercicio)

- Este método se puede generalizar también a matrices cuyas dimensiones no sean de la forma 2 n.

Algoritmos Golosos

- **Idea:** Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar las implicancias de esta selección en los pasos futuros ni en la solución final.

- *Se usan principalmente para problemas de optimización.*

Componentes:

- Lista o conjunto de candidatos
- Conjunto de candidatos ya usados
- Función que verifica si un conjunto de candidatos da una solución al problema.
- Función que verifica si un conjunto de candidatos es hasta el momento factible.
- Función de selección
- Función objetivo

Función GREEDY (C)

{C es el conjunto de candidatos}

S ←------ ∅

Mientras NOT solución(S) y C ≠ ∅ hacer

x ←---------- elemento que maximice select (x)

C <---------- C \ {x}

Si S U {x} es factible entonces s ←--------- S U {x}

Si solución(S) entonces RETURN S

sino RETURN "no hay solución"

Qué hacen las funciones solución(S) *y* select (x) ?

Minimizar el tiempo de espera en un sistema

Un servidor tiene n clientes para atender. Se sabe que el tiempo requerido para atender cada cliente i es t_i. Se quiere minimizar

$$T = \Sigma_i \text{ (tiempo que i está en el sistema)}$$

El algoritmo goloso que atiende al cliente que requiere el menor tiempo de atención entre los que aún están en la cola resuelve el problema. (*hay que demostrarlo)*

Supongamos que I = (i_1,i_2,.....i_n) es una permutación de los enteros {1,2....n}. Si los clientes son atendidos en el orden I entonces

$$T = t_{i1} + (t_{i1} + t_{i2}) + (t_{i1} + t_{i2} + t_{i3}) + ...$$
$$= \Sigma_k (n-k+1)\, t_{ik}$$

Usando esta expresión podemos probar que el algoritmo goloso es correcto……..

Supongamos que en I hay dos enteros a y b con a < b y tal que $t_{ia} > t_{ib}$.

O sea el cliente a es atendido antes que el b, a pesar de que el primero requiere mas tiempo de servicio. Si intercambiamos cambiamos las posiciones de a y b tenemos un nuevo orden I´ para el cual

$$T(I') = (n-a+1)\, t_{ib} + (n-b+1)\, t_{ia} + \Sigma_{k \neq a,b} (n-k+1)\, t_{ik}$$

Entonces

$$T(I) - T(I') = (n-a+1)(t_{ia} - t_{ib}) + (n-b+1)(t_{ib} - t_{ia}) =$$
$$= (b-a)(t_{ia} - t_{ib}) > 0$$

Vemos que podemos seguir mejorando el valor de T intercambiando el orden de los clientes hasta obtener el orden óptimo dado por el algoritmo goloso.

==========

Cuál es la complejidad de este algoritmo?

Problema de la mochila

Quiero decidir cuáles son los elementos que tengo que elegir para llevar en la mochila para maximizar mi beneficio respetando la restricción de capacidad de la misma.

$$\text{Max } z = \Sigma_j c_j x_j$$

sujeto a que

$$\Sigma_j a_j x_j \leq b$$
$$x_j \in N_+$$

b : capacidad de la mochila

c_j : beneficio de llevar una unidad del elemento j

a_j : peso del elemento j

b, c_j , a_j positivos

En esta primera versión del problema de la mochila vamos a suponer que podemos llevar a lo sumo un objeto de cada tipo o "partes" de los mismos, es decir que consideraremos que las variables x_j son reales positivas, o sea queremos resolver el siguiente problema:

$$\text{Max} \quad \Sigma_j \, c_j \, x_j$$

sujeto a que

$$\Sigma_j \, a_j \, x_j \leq b$$

$$0 \leq x_j \leq 1$$

Ejemplo:

n = 5, b = 100 y los beneficios y pesos están dados en la tabla

a_j	10	30	20	50	40
c_j	20	66	30	60	40

Posibles ideas para un algoritmo goloso para este problema:

- Elegir los objetos en orden decreciente de su beneficio….
- Elegir los objetos en orden creciente de su peso……..

Sirven estas ideas?. Dan el resultado correcto?

Que pasa si ordenamos los objetos por su relación beneficio/peso? O sea en orden decreciente de los c_j/a_j ?.

Lema: Si los objetos se eligen en el orden decreciente de los valores c_j/a_j entonces el algoritmo goloso encuentra la solución óptima al problema de la mochila con variables continuas.

Dem: supongamos que tenemos los elementos ordenados en orden decreciente de su relación c_j/a_j . Sea $X = (x_1,\ldots x_n)$ la solución encontrada por el algoritmo goloso. Si todos los x_i son 1 la solución es óptima. Sino sea j el menor índice tal que sea $x_j < 1$.

Es claro que $x_i = 1$ para $i < j$ y que $x_i = 0$ para $i > j$ y que

$$\sum a_i x_i = b$$

Sea $V(X) = \sum c_i x_i$ el valor de la solución dada por el algoritmo goloso.

Sea $Y = (y_1 \ldots\ldots y_n)$ otra solución cualquiera factible. Como

$$b = \sum a_i \ x_i \geq \sum a_i \ y_i$$

se puede ver que

$$\sum (x_i - y_i) \ a_i \geq 0$$

Además para todo índice i se verifica que

$$(x_i - y_i) \ c_i / a_i \geq (x_i - y_i) \ c_j / a_j$$

entonces

$$V (X) - V (Y) = \sum (x_i - y_i) \ c_i = \sum (x_i - y_i) \ a_i c_i / a_i \geq$$
$$\geq (c_j / a_j) \sum (x_i - y_i) \ a_i \geq 0$$

O sea el valor de la solución determinada por el algoritmo goloso es mejor o igual al valor de cualquier otra solución.

Veamos que ocurre con el problema de la mochila, en el caso de que las variables x_j tengan que tomar necesariamente valores enteros (como ocurre en la práctica, es decir cuando no podemos llevar un "pedazo" de un objeto).

Algoritmo goloso para el problema de la mochila con variables enteras nonegativas:

- Ordenar los elementos de forma que

  $c_j / a_j \geq c_{j+1} / a_{j+1}$

- Para j=1,n y mientras $b \neq 0$ hacer

  $x_j = \lfloor b / a_j \rfloor$

  $b = b - a_j x_j$

  $z = z + c_j x_j$

- Parar

Cuando las variables son enteras este algoritmo goloso no da siempre la solución óptima para el problema de la mochila.

Ejemplo: tenemos una mochila de tamaño 10 y 3 objetos, uno de peso 6 y valor 8, y dos de peso 5 y valor 5.

En el caso de variables enteras el algoritmo goloso que presentamos puede considerarse como una heurística para resolver el problema.

Planificación de tareas

Hay n tareas que deben ser ejecutadas cada una de las cuales requiere una unidad de tiempo. En cada momento se puede ejecutar una sola tarea.

Cada tarea i produce un beneficios g_i y debe ser ejecutada antes del tiempo d_i.

Ejemplo con n = 4

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

Algoritmo goloso para este problema

En cada paso elegir para ejecutar la tarea aún no realizada que produce el mayor beneficio, siempre y cuando el conjunto de tareas resultante sea factible.

Cómo se ve que un conjunto de tareas es factible?.

Cuando hay un ordenamiento de esas tareas que permite ejecutarlas a todas a tiempo.

Lema: sea J un conjunto de tareas y $s = (s_1, s_2, \ldots\ldots, s_k)$ una permutación de dichas tareas tal que

$d_{s1} \leq d_{s2} \leq \ldots\ldots \leq d_{sk}$

J es factible si y sólo si s también lo es.

Dem: $\Leftarrow$ obvio.

$\Rightarrow$ Si J es factible existe al menos una sucesión de los trabajos $\rho = (r_1, r_2, \ldots\ldots r_k)$ tal que $d_{ri} \geq i$, para $1 \leq i \leq k$.

Si fuera $s \neq \rho$, sea a el menor índice tal que $s_a \neq r_a$ y sea b el índice tal que $r_b = s_a$.

Claramente $b > a$ y $d_{ra} \geq d_{sa} = d_{rb.}$

Entonces el trabajo r_a podría ser ejecutado más tarde en el lugar de r_b y este último en el lugar de r_a. Si los cambiamos en ρ tenemos una sucesión factible que tiene un trabajo más en coincidencia con s. Repitiendo este procedimiento podemos llegar a obtener s y ver entonces que es factible.

Lema: Este algoritmo goloso obtiene la solución óptima para este problema de planificación de tareas.

Dem: Supongamos que el conjunto I elegido por el algoritmo goloso es distinto de un conjunto J óptimo. Sean S_I y S_j dos sucesiones factibles para estos conjuntos.

Hacemos intercambios de modo de obtener dos sucesiones factibles S'_I y S'_j tal que los trabajos comunes en ambas secuencias sean ejecutados en el mismo momento (pueden quedar gaps en alguna de las planificaciones).

O sea si tenemos las secuencias…

S_I p y q x r - - -

S_J r s t p u v q w

podemos reordenarlas de la siguiente forma

S'_I x y - p r - q -

S'_J u s t p r v q w

(En S'_I y S'_J se siguen respetando los tiempos de entrega d_i)

Como hemos supuesto que los conjuntos I y J son distintos, consideremos ahora un momento en el cual la tarea que está programada en S'_I es distinta de la programada en S'_j:

- Si una tarea a está programada en S'_I y en S'_j hay un gap (la tarea no pertenece a J) entonces $J \cup \{a\}$ sería mejor que J, absurdo porque J es óptimo.

- Si alguna tarea b está programada en S'_j y en S'_I hay un gap entonces $I \cup \{b\}$ es factible y por lo tanto el algoritmo goloso tendría que haber incluído a b y no lo hizo.

- La última posibilidad es que haya una tarea a programada en S'_I y otra b en S'_j. Esto implica que a no aparece en J y b no aparece en I :

  - si fuera $g_a > g_b$ se podría substituir a por b en J y mejorar a J (absurdo porque J era óptimo).

  - si fuera fuera $g_b > g_a$ el algoritmo goloso tendría que haber elegido a b antes de considerar a.

  - entonces $g_b = g_a$

Por lo tanto S'_I y S'_j tienen programadas en cada momento, o ninguna tarea, la misma tarea o distintas tareas que dan el mismo beneficio. Y entonces como J era óptimo I también lo es.

Algoritmo goloso para este problema de planificación de tareas

(se supone que las tareas están numeradas de modo que $g_1 \geq g_2 \geq \ldots\ldots\ldots \geq g_n$)

```
Función sequence (d[0……n])
d[0], j[0] ←--------- 0
k, j[1] ←------1
Para i= 1,n hacer
        r <-k
         mientras d[j [r]] > max ( d(i),r) hacer r = r-1
         si d[j [r]]  ≤ d[i] y d[i] > r entonces
                  para l=k-1, r+1 hacer j[l+1] = j[l]
         j[r+1] ←---------i
         k ←---- k+1
Return  k, j[1,….k]
```

Backtracking

Técnica para recorrer sistemáticamente todas las posibles configuraciones de un espacio. Puede pensarse también que es una técnica para explorar implícitamente árboles dirigidos (o grafos dirigidos en general pero sin ciclos).

- Habitualmente s utiliza un vector a = (a1; a2; : : : ; ak) para representar una solución candidata, cada a_i pertenece un dominio/conjunto ordenado y finito S_i .

- Se extienden las soluciones candidatas agregando un elemento más al final del vector a, las nuevas soluciones candidatas son sucesores de la anterior.

Procedure backtrack (v[1….k])

{v es un vector k-prometedor }

si v es una solución al problema entonces escribir v

sino para cada vector “k+1-prometedor” w tal que

w[1…k] = v[1…k] hacer backtrack (w[1…k+1])

==========

No necesariamente exploramos todas las ramas del árbol : **“poda”**

Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez sin que ninguna "amenace" a otra.

- *Cuántas operaciones tenemos que hacer si consideramos todas las combinaciones del tablero tomadas de a 8 y después vemos cuales sirven?*

$$\binom{64}{8} = 442616536$$

- *Si mejoramos esto usando la representación* $(i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8)$ *donde* i_j *indica que hay una reina en la columna j y la fila* i_j *y analizamos cuales son factibles o no cuántas operaciones tendremos?.*

$$8^8 = 16777216$$

(se encuentra una solución después de revisar "sólo" 1299 852 posiciones)

- *Si vemos que dos reinas no pueden estar ni en la misma fila ni en la misma columna vemos que en realidad podemos representar el problema mediante las permutaciones de (1,2,3,4,5,6,7,8). Cuántas operaciones tenemos en este caso?.*

  8! = 40320

(implementando este algoritmo podemos ver que hay que revisar 2830 posiciones hasta encontrar una primera solución)

Algoritmo de bactracking

Decimos que un arreglo V de los enteros 1 a 8 es k-prometedor si ninguna de las reinas ubicadas en las posiciones (1, v(1)), (2,v(2)), (3,v(3)),…..(k,v(k)), $1 \leq k \leq n$ amenaza a las otras.

O sea v es k-prometedor si para todo $i \neq j$ entre 1 y k, tenemos que $v(i) - v(j) \notin \{i-j, 0, j-i\}$.

Un vector 8-prometedor es solución del problema de las 8 reinas.

Algoritmo de backtracking

Procedimiento Queens (k, try, diag45, diag 135)
{try[1,…k] es k-prometedor, col = {try [i], 1≤ i≤ k}
diag45 = {try [i]-i +1, 1≤ i≤ k}
diag135 = {try [i] + i - 1, 1≤ i≤ k}}
Si k = 8
    entonces {un vector 8-prometedor es solución}. Return try
Sino {explorar las k+1-prometedoras extensiones de try}
    Para j=1,8
    si j ∉ col y j-k ∉ diag45 y j+k ∉ diag135 entonces
        try[k+1] ←--j
            {try [1,….k+1] es k+1-prometedor}
            Queens (k+1, col U {j}, diag45 U {j}, diag135 U {j+k}

- Se puede ver computacionalmente que este árbol de bactracking tiene 2057 nodos y que es suficiente explorar 114 para encontrar la primera solución. *(Cómo se hace para calcular esto?).*

Cuál es la complejidad de este algoritmo de bactracking?

Ejercicio: Mostrar un valor de $n \neq 2$ para el cual el problema de las n reinas no tiene solución.

Otros ejemplos

Encontrar, usando backtracking todas las soluciones enteras de

$$x_1 + 2x_2 + x_3 \leq 10$$

$$1 \leq x_i \leq 4 \qquad i = 1,2,3$$

Algoritmos probabilísticos

- Cuando un algoritmo tiene que hacer una elección a veces es preferible elegir al azar en vez de gastar mucho tiempo tratando de ver cual es la mejor elección.

Pueden dar soluciones diferentes si se aplican dos veces a la misma instancia. Los tiempos de ejecución también pueden ser muy diferentes.

- Tiempo promedio de un algoritmo determinístico: *considera todas las instancias de un mismo tamaño son "similares".* (ejemplo: quicksort)
- Tiempo esperado promedio de un algoritmo probabilístico : *es el tiempo medio de los tiempos de resolver la misma instancia del mismo problema "muchas veces".*

- Peor tiempo esperado: *tomando en cuenta el peor caso de todas las instancias de un cierto tamaño.*

Clasificación de algoritmos probabilisticos

- **Algoritmos al azar para problemas numéricos:** *la respuesta es siempre aproximada pero se espera que la solución sea mejor cuando más tiempo hay para ejecutar el algoritmo. (simulación, integración numérica).*

- **Algoritmos de Monte Carlo:** *se quiere una respuesta exacta. Por ejemplo problemas de decisión. Un algoritmo Monte Carlo da siempre una respuesta pero la respuesta puede no ser correcta. La probabilidad de suceso, es decir de respuesta correcta crece con el tiempo disponible para correr ese algoritmo. La principal desventaja es que en general no se puede decidir eficientemente si la respuesta es correcta o no.(ejemplo: decidir si un arreglo T tiene un elemento mayoritario, o sea que aparece en T más de n/2 veces)*

- **Algoritmos Las Vegas**: *nunca dan una respuesta incorrecta pero pueden no dar ninguna respuesta. También la probabilidad de suceso, es decir de tener respuesta correcta crece con el tiempo disponible para correr ese algoritmo. (ejemplo: en el problema de las 8 reinas poner las reinas al azar en las filas que aún están desocupadas, cuidando solamente que la solución en construcción sea factible)*

- **Algoritmos Sherwood** : *en este caso el algoritmo siempre da una respuesta y la respuesta es siempre correcta. Se usan cuando algún algoritmo determinístico para resolver un algoritmo es mucho más rápido en promedio que en el peor caso. Al incorporar un factor de azar el algoritmo puede llegar a eliminar la diferencia entre buenas y malas instancias. (ejemplo: Quicksort)*

En todos estos casos suponemos que tenemos disponible un generador de números al azar de costo computacional unitario.

O sea dados reales a y b con a < b, uniform (a,b) devuelve un número x elegido al azar en el intervalo $a \leq x < b$. La distribución de x es uniforme en el intervalo y llamadas sucesivas del generados dan valores independientes de x.

Para generar enteros al azar uniform (i…j) devuelve un entero k elegido al azar uniformemente en el intervalo

$i \leq k \leq j$

Ejemplo de algoritmo tipo MonteCarlo: ***ver si un arreglo tiene un elemento mayoritario.***

```
Function may(T[1,…n])
i = uniform (1,…n)
x = T[i]
k = 0
Para j =1,n hacer
   if T[j] = x then k = k +1

Return (k > n/2)
```

Cuál es la probabilidad de que este algoritmo de la respuesta correcta?

Ejemplo de algoritmo tipo *Las Vegas*: *el problema de las 8 reinas.*

```
Procedimiento Queens LV (var sol[1…8], sucess)
array ok[1…8]  (posiciones disponibles)
col, diag45, diag135 ⇐ ∅
Para k = 0,7 hacer
(sol [1…k] es k prometedor, hay que ubicar a la reina k+1)
    nb = 0
    para j = 1, 8 hacer
       si j ∉ col y j-k ∉ diag45 y j+k ∉ diag135 entonces (la columna j esta
      disponible)
      nb = nb +1
      ok[nb] = j
    si nb = 0 entonces sucess =false
    j = ok[uniform (1….nb)]
    col = col U {j}
   diag45 = diag45 U {j}
    diag 135 =  diag135 U {j+k}
    sol[k+1] = j
return success = true
```

Pueden encontrar más ejemplos de aplicación de las diferentes técnicas de diseño de algoritmos en el libro de Brassard y Bratley

Heurísticas

- Dado un problema Π , un algoritmo heurístico es un algoritmo que intenta obtener soluciones para el problema que intenta resolver pero no necesariamente lo hace en todos los casos.

- Sea Π un problema de optimización, I una instancia del problema, x*(I) el valor óptimo de la función a optimizar en dicha instancia. Un algoritmo heurístico obtiene una solución con un valor que se espera sea cercano a ese óptimo pero no necesariamente el óptimo.

- Si H es un algoritmo heurístico para un problema de optimización llamamos $x^H(I)$ al valor que devuelve la heurística.

Porqué usar algoritmos heurísticos?. En que tipo de problemas?

Cuándo decimos que una heurística es "buena"*?*

Cómo hacemos para saber si una heurística es buena?

Algoritmos aproximados

Decimos que H es un algoritmo ε- aproximado para el problema Π si para algún $\varepsilon > 0$

$$| x^H(I) - x^*(I) | \leq \varepsilon | x^*(I) |$$

O equivalentemente un algoritmo es aproximado si existe $\rho > 0$
tal que para toda instancia I

$$x^H(I) / x^*(I) \leq \rho$$

(problema de minimización)

Situación ideal, pero poco frecuente

Cómo evaluamos una heurística?

- Comportamiento medio
- Análisis probabilístico
- Comportamiento en peor caso *(algoritmos aproximados):*

  i) razón de perfomance en el peor caso :

  $r = \sup \{x^H(I) / x^*(I) \}$ ó $r = \inf \{x^H(I) / x^*(I) \}$

  según estemos minimizando o maximizando.

  ii) error relativo

  $er = | x^H(I) - x^*(I) | / | x^*(I)|$

En la práctica no podemos calcular exactamente estos valores pero si a veces obtener cotas para ellos.

Ejemplo de algoritmo aproximado

Cuando las variables son enteras, el algoritmo goloso que vimos para el problema de la mochila es un algoritmo aproximado.

Se puede demostrar que para toda instancia I de dicho problema se verifica que:

$$x^H(I) / x^*(I) \geq (c_1 \lfloor b/a_1 \rfloor) / (c_1 (b/a_1)) =$$
$$= \lfloor b/a_1 \rfloor / (b/a_1) \geq 0.5$$

(es un problema de maximización)

Metaheurísticas: métodos generales para construir heurísticas para una gran variedad de problemas.

Continuará……..