Fundamentos de Análise e Deseño de Algoritmos e Estruturas de Dados

---

TEMA 5: TÉCNICAS DE DESEÑO DE ALGORITMOS.

5.1.- Algoritmos Ávidos.

Ex. Dijkstra, Kruskal, Prim

Funcionan en etapas, en fases. En cada fase tomamos unha decisión: óptimo local. Esperamos que na derradeira fase, o algoritmo produza un óptimo global. Isto hai que demostrá-lo.

Problema: as solucións óptimas locais non sempre funcionan.

Ex. Optimización do tráfico

Ex. Problema: Planificación de un SO

traballos j1,...,jn
tempo de execución t1,...,tn
unha CPU

[1] minimizar tempo médio de terminación TMT

Ex. j1 : 15
j2 : 8
j3 : 3
j4 : 10

} consumo de tempos de CPU de cada traballo

planificación #1 (pode ser First In, First Out)

j1	j2	j3	j4
0	15	23	26 36

TMT = (15+23+26+36)/4 = 25

planificación #2 (pode ser Shortest Job First)

j3	j2	j4	j1
0	3	11	21 36

TMT = (3+11+21+36)/4 = 17’75

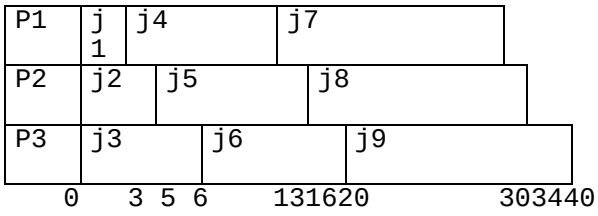
Minimizar con P procesadores

Ex. P=3

j1 : 3	j5 : 11	j9 : 20
j2 : 5	j6 : 14	
j3 : 6	j7 : 15	
j4 : 10	j8 : 18	

•planificación óptima

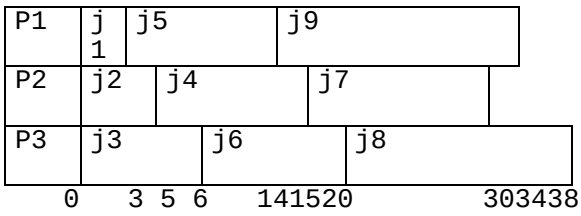
TMT = 165/9 = 18’33



Pode haber várias planificacións óptimas, se P divide nºtraballos de xeito exacto, entón temos: para cada $0 \leq i < n/p$ poden-se colocar os traballos j_{ip+1} ... $j_{(i+1)p}$ nun procesador distinto.

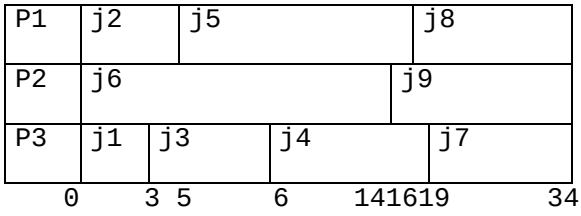
•outra planificación óptima

TMT = 165/9 = 18’33



[2] Minimizar o tempo de terminación final TTF

A segunda planificación mellora o TTF con respeito à primeira



TMT = 168/9 → TMT non óptimo

TTF = 34 → TTF óptimo

A estes algoritmos son coñecidos como “Problema de empaquetamento” ou como “algoritmo da mochila”.

5.2.- Divide e vencerás.

Pasos: -dividir (*)
-vencer (**)

(*)División do problema principal en problemas mais pequenos que se resolven recursivamente. División en duas ou mais partes. (**)Despois encontramos solución ao problema principal a partir das solucións aos subproblemas.

Ex. Subsecuência máxima (1.3.2 Algoritmo 3) en O(n log n)
Mergesort (3.4) en O(n log n) pior caso
Quicksort (3.5) en O(n log n) caso médio

Tempo de execución:

Ex. Mergesort: *2* subproblemas coa metade do vector (**n/2**) e traballo adicional O(n): $T(n) = 2T(\mathbf{n/2})+O(n)$

•TEOREMA

A solución da ecuación

$$T(n)=aT(n/b)+\theta(n^k) \qquad a\geq 1 \; b>1$$

é:

$$T(n)=\begin{cases} O(n^{\log_b a}) & \text{se } a > b^k \\ O(n^k \log n) & \text{se } a = b^k \\ O(n^k) & \text{se } a < b^k \end{cases}$$

Ex. Mergesort a=b=2, k=1 → O(n log n)
por $O(n^k \log n)$

Ex. 3 subproblemas, cada un con n/2,
+O(n) para combinar solución
a=3, b=2, k=1
por $O(n^{\log_b a}) \rightarrow O(n^{1'59})$

Ex. Se 3 subproblemas, cada un con n/2
$+O(n^2)$ para combinar solución
a=3, b=2, k=1
por $O(n^k) \rightarrow O(n^2)$

*Este teorema non cobre todos os casos de “Divide e Vencerás”.

•TEOREMA

(xeneraliza o anterior)

A solución da ecuación

$$T(n)=aT(n/b)+\theta(n^k \log^P n) \quad a\geq 1\ b>1\ p\geq 0$$

é:

$$T(n)=\begin{cases} O(n^{\log_b a}) & \text{se } a > b^k \\ O(n^k \log^{P+1} n) & \text{se } a = b^k \\ O(n^k \log^P n) & \text{se } a < b^k \end{cases}$$

•TEOREMA

Se

$$\sum_{i=1}^{k} \alpha_i < 1$$

entón a solución de

$$T(n) = \sum_{i=1}^{k} T(\alpha_i n) + O(n)$$

é

$$T(n) = O(n)$$

5.3.- Programación Dinámica.

Divide e vencerás: subexemplares → resolver e combinar (estratéxia descendente).

Existen muitos subexemplares idénticos: repetiremos chamada ao algoritmo. Entón hai ineficiência no desenrolo.

A alternativa é conservar o resultado de cada un dos cálculos, dos resultados parciais co que melloraremos a eficiência.

A ideia da **programación dinámica** é **non calcular duas veces o mesmo**.

Para isto usamos algún mecanismo, como unha tábua de resultados.

Seguimos unha **estratéxia ascendente**: primeiro os subexemplares mais pequenos (problemas mais simples) e despois combinan-se as solucións para producir solucións aos subexemplares mais grandes até chegar ao orixinal.

Este método é distinto do método descendente que segue a estratéxia de Divide e Vencerás.

Ex. Cálculo de números combinatorios.

Mediante estratéxia descendente:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ si } 0<k<n;\ 1 \text{ en outro caso}$$

O cal expresado en pseudocódigo é:

```
función c ( N , K )
      se
            K = 0
      ou
            K = N
      entón
            devolver 1
      senón
            devolver c (N-1,K-1)+C(N-1,K)
      fin-se
fin-función
```

Para muitos c(i,j) con i<n e j<k recalculan-se

$$\Omega \left(\binom{n}{k}\right) \approx \text{Fib1}$$

Alternativa: resultados intermédios → triángulo de Pascal

	0	1	2	3	...	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1			1			
...					...		
n-1	1					*	**
n	1						c(n,k)

[*non está na diagonal a pesar de como aparece na tábua]

* c(n-1,k-1) ⎫
⎬ → c(n-1,k-1) + c(n-1,k) = c(n,k)
** c(n-1,k) ⎭

Calcula-se matriz liña a liña.

Trata-se de un vector con lonxitude k, en cada paso actualizamos o vector de direita a esquerda.

Complexidade espacial: O(k)
cada fila = 1 vector de k posicións
Complexidade temporal: O(nk) n indica
n indica cantas veces hai que recalcular o vector
k indica cantos elementos ten o vector

A **programación dinámica** resolve **problemas de optimización** que satisfan o **princípio de optimalidade**: "*Nunha secuência óptima de decisións, toda subsecuência ha de ser tamén óptima*". Non todos os problemas cumpren isto.

Aplicacións: produto de várias matrices: minimizar o número de multiplicacións; árbores de busca óptimas; o problema do viaxante de comércio.

5.4.- Algoritmos aleatórios.

Algoritmos aleatórios son aqueles nos que polo menos unha vez durante a sua execución, emprega-se un número aleatório para tomar unha decisión.

O tempo de execución do algoritmo aleatório depende non só da entrada, senón tamén dos número(s) aleatório(s) producido(s).

O tempo de execución de un algoritmo aleatório para o pior caso é case sempre igual ao tempo de execución de un algoritmo non aleatório para o pior caso. A diferenza importante é que un bo algoritmo aleatório non ten entradas malas, senón malos números aleatórios (en relación coa entrada específica).

Vexamos un exemplo acerca de isto. Analisamos duas versións da ordenación rápida: no Quicksort A escollemos como pivote o primeiro elemento; no Quicksort B escollemos como pivote un elemento ao chou. En ambos casos o tempo de execución do pior caso é $\Theta(n^2)$, porque é posíbel que en cada paso se escolla como pivote o elemento mais grande.

A diferenza entre os piores casos é que hai unha entrada particular que sempre pode apresentar-se na variante A e que causa un mal tempo de execución. A variante A executará-se en tempo $\Theta(n^2)$ **sempre** que receba unha lista ordenada. En câmbio, a variante B, se recebe duas veces a mesma entrada (ordenada, como pior caso do que falamos aqui), terá dous tempos de execución distintos, dependendo de como saian os números aleatórios.

Levamos suposto sempre no cálculo de O, dos tempos de execución, que todas as entradas son equiprobábeis, pero isto non é certo. As entradas case ordenadas son muito mais probábeis do que se poda pensar en princípio, e isto causa problemas coa ordenación rápida e coas árbores binárias de busca.

Aqui está o valor dos algoritmos aleatórios: **cos algoritmos aleatórios a entrada específica perde importância.**

Asi, non falamos de *entradas malas* senón de *números aleatórios malos.* Falamos de un tempo de execución *esperado*, onde agora temos unha média de todos os números aleatórios posíbeis à vez, de todas as entradas posíbeis.

Usando a ordenación rápida con un pivote aleatório ten-se un algoritmo con tempo esperado $O(n \cdot \log n)$. Isto significa que para calquer entrada, ainda estando ordenada, espéra-se que o tempo de execución sexa $O(n \cdot \log n)$, con base nas estatísticas dos números aleatórios.

Unha cota de tempo de execución *esperado* é *algo mais forte* que unha cota do *caso médio,* pero -por suposto- é *mais feble* que unha cota para o *pior caso.*

O que pasa é que as solucións que dan a cota do pior caso non sempre son tan práticas como as solucións que dan unha cota para o caso médio; mentres que de cote os algoritmos aleatórios si son práticos.

O problema que se plantexa nos algoritmos aleatórios é como xerar os números aleatórios. En realidade *é virtualmente imposíbel* lograr a verdadeira aleatoriedade nun computador (xá que dependen de un algoritmo e polo tanto é imposíbel que sexan aleatórios).

En xeral abonda con producir números pseudo aleatórios, que son números que ***parecen aleatórios*** (cumpren a maior parte das muitas propriedades estatísticas que tên os números puramente aleatórios).

As aplicacións dos algoritmos aleatórios son: listas con saltos -busca e inserción en tempo esperado [tempo de execución que se supón para calquer secuência de entrada] O(log n)-; e primalidade [comprobar se un número é primo] de números grandes.

5.5.- Algoritmos con retroceso (backtracking).

Trata-se da exploración de un grafo orientado (frecuentemente acíclico, referido a árbores). Os algoritmos con retroceso permiten realizar unha busca exaustiva de solucións.

Aplicacións: xogos de estratéxia; xadrez; damas.

Exemplo: problemas das 8 raíñas nun taboleiro de 8x8, hai que colocá-las de xeito que non se maten mutuamente.

[solución 1] Colocar as raíñas en todas as disposicións posíbeis e verificar se cada unha de elas é solución.

$\binom{64}{8}$ posibilidades

[solución 2] Non colocar mais de unha raíña por fila nen mais de unha raíña na mesma diagonal.

vector [1.. 8] = posíbel solución

o vector representa o taboleiro

v [i] indica a fila na que está a raíña

exemplo: v=[**3**,1,*6*,**3**,8,*6*,4,7] non é solución

```
para i1 ← 1 até 8 facer
  para i2 ← 1 até 8 facer
    para i3 ← 1 até 8 facer
      para i4 ← 1 até 8 facer
        para i5 ← 1 até 8 facer
          para i6 ← 1 até 8 facer
            para i7 ← 1 até 8 facer
              para i8 ← 1 até 8 facer
                        ensaio ← (i1,i2,i3,i4,i5,i6,i7,i8)
                        se solución (ensaio)
                        entón escreber (ensaio); parar
                        finse
             finpara
           finpara
         finpara
       finpara
```

```
      finpara
    finpara
  finpara
```

8^8 posibilidades = 16.777.216
examinan-se 1.299.852

[solución 3] Evitar iguais no vector, o vector seria unha permutación de 8 elementos distintos.

```
ensaio ← permut_inicial
mentres non solución (ensaio)
e solución ≠ permut_final
facer
      ensaio ← permut_seguinte
finmentres
se solución (ensaio)
entón escreber (ensaio)
finse
```

8! posibilidades = 40.320

Detén-se segundo sexa o algoritmo de xeración de permutacións. Unha posibilidade: examinan-se 2.830

Como se traballa con permutacións as raíñas non coincidirán na mesma fila de nengún xeito, asi que para saber se unha permutación é solución abonda con chequear as diagonais

Problema que ten esta solución: que non se verifica a posición até que todas as raíñas están colocadas.

Exemplo: se se colocan r1 e r2 na diagonal principal xá non é solución. Este algoritmo ten 720 formas distintas de colocar as 6 raíñas restantes.

[solución 4] Mellora o problema da anterior mediante o retroceso. Realiza unha exploración en árbore.

Def.- V [1..k] é k-completábel (1≤k≤8) se nengunha das k raíñas se dan xaque entre si.

Isto significa que para todo par i, j de raíñas entre as k primeiras (∀i≠j entre 1 e k) cumpre-se que **v[i]-v[j] ∉ {i-j,0,j-i}**

Exemplo:

x				
	x			

3	4			

v representa a matriz

é 2-completábel?

non porque: v[1]-v[2]=3-4=-1 ∈ {-1,0,1}={1-2,0,2-1}

Segundo esta definición a solución ao problema serán os vectores 8-completábeis.

Chamamos N ao conxunto de vectores k-completábeis. G=(N,A) orientado onde (u,v)∈A ⇔ u é u-completábel, v é k+1-completábel, v[i]=u[i] ∀i∈[1..u]

Obtemos árbore na raiz: vector valeiro (k=0), nas follas solucións (k=8), ou ben 'caleixóns sen saída' (0<k<8).

Exemplo: v[1,4,2,5,8] non é 5-completábel

1	4	2	5	8

X							
		X					
	X						
			X				
				X			

Exploramos a árbore mediante percorrido en profundidade.

nº nodos < 8!

2.057 nodos posíbeis, dos que abonda explorar 114.

Como verificar que un vector é k-completábel? Abonda con comprobar a posición da derradeira raíña.

Engadir información en cada nodo:
- {colunas ocupadas}
- {diagonais / ocupadas} (1)
- {diagonais \ ocupadas} (2)

Test (0,∅,∅,∅) → chamada ao procedimento que produce todas as solucións ao problema.

```
{-------------------------------------------------------------------}
procedimento Test ( k , col , diag1 , diag2 )
      { ensaio [1..k] é k-completábel }
      { col={ensaio[i]/1≤i≤k} }
      { diag1={ensaio[i]-i+1/1≤i≤k} }
      { diag2={ensaio[i]+i+1/1≤i≤k} }
      se k = 8
      entón escreber ( ensaio )
      senón
            para i ← 1 até 8 facer
              se i ∉ col
              e i-k ∉ diag1
              e i+k ∉ diag2
              entón
```

```
                ensaio [ k+1 ] ← i {k+1-completábel}
                Test(k+1,col∪{i},diag1∪{i-k},diag2∪{i+k})
            finse
          finpara
      finse
finprocedimento
{-------------------------------------------------------------}
```

A xeneralización do problema consiste en manexar n raíñas nunha tábua de nxn posicións.

n=12

[3] 1ª solución 4.546.044-ésima permutación

[4] 1ª solución 262-ésimo nodo

<fin do tema 5>