Algoritmos : alguns esquemas gerais

- "Dividir para Conquistar"
- Tabulação
- Programação Dinâmica

"Dividir para Conquistar"

Se o problema tem dimensão suficientemente pequena a solução é directa.
Caso contrário, divide-se em sub-problemas, resolvem-se estes problemas e combinam-se as soluções.

Esquema geral:

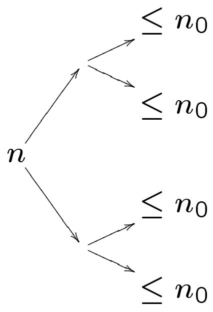
```
function prob(n){
  se n <= n0, a solução é directa
  senão
    Divide-se em sub-problemas P1,...,Pk
    Resolvem-se *recursivamente* estes problemas
    Combinam-se as soluções.
```

Relação com as provas por indução: Consegue-se resolver os problemas para o tamanho n supondo que se consegue resolver os problemas de dimensão $< n$.

Variantes do “Dividir para Conquistar”

- Número de problemas em que se sub-divide: 2, 3,... ou não constante.
- Método da divisão
- Número de sub-problemas que se resolvem: 1, 2, 3,...
- Método da combinação das soluções

Note-se que em geral a técnica só resulta se a aplicação for *recursiva*.



“Dividir para Conquistar” - Exemplos gerais

Exercício 32 *Determine a ordem de grandeza do tempo de execução correspondente à seguinte aplicação do “dividir para conquistar”:*
- O algoritmo directo tem eficiência αn^2.
- n é da forma $n_0 \times 2^p$, $p \geq 1$.
- A divisão é em 2 partes iguais.
- A eficiência da divisão em sub-problemas é $k_1 n$.
- A eficiência da combinação das soluções é $k_2 n$. ◇

Exercício 33 *Um determinado problema tem um algoritmo directo $O(n^2)$. Uma aplicação do “dividir para conquistar” exige a divisão em 3 partes (sobrepostas) de tamanho $n/2$ (estamos a supor que n é da forma $n_0 \times 2^p$). O tempo correspondente é majorado pela recorrência*

$$t(n) = \begin{cases} \alpha & \text{se } n = n_0 \\ 3t(n/2) + kn & \text{se } n > n_0 \end{cases}$$

Mostre que há de facto uma melhoria importante na eficiência que é agora de ordem $O(n^{\log 3}) \approx O(n^{1.59})$. ◇

Uso em pesquisa/ordenação - sumário

Já vimos a aplicação do "dividir para conquistar" em alguns algoritmos como o "mergesort".

Método	NS	OS	NR	OC	Pi	Md
P. binária	2	1	1		$\log n$	$\log n$
Mergesort	2	1	2	n	$n \log n$	$n \log n$
Quicksort	2	n	2	1	n^2	$n \log n$
Mediana	2	1	2	n	n	n

onde
NS: Número de sub-problemas
OS: Ordem de grandeza da eficiência da divisão em sub-problemas
NR: Número de sub-problemas que de facto se resolvem
OC: Ordem de grandeza da eficiência da combinação das soluções
Pi: Ordem de grandeza da eficiência global no pior caso
Md: Ordem de grandeza da eficiência global no caso médio

Multiplicação com precisão arbitrária

Por vezes é necessário realizar operações aritméticas com precisão arbitrária. Vamos apenas estudar um algoritmo eficiente para multiplicar inteiros muito longos. Não consideramos outras questões relacionadas com a aritmética de precisão arbitrária.

Sejam os operandos x e y e seja $n = \max(|x|, |y|)$, o comprimento do maior deles.

Exercício 34 *Mostre que o algoritmo "escolar" para determinar $x + y$ é $O(n)$.* $\diamond$

Exercício 35 *Mostre que o algoritmo "escolar" para determinar $x \times y$ é $O(n^2)$.* $\diamond$

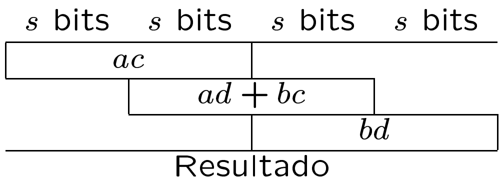
Multiplicação com precisão arbitrária

É possível multiplicar com melhor eficiência (ordem de grandeza)!

<u>Algoritmo 1</u>. Supondo que n é par, $n = 2s$, dividimos os operandos em 2 partes: bits (ou dígitos) da esquerda e da direita e consideremos o seu produto:

$$\begin{aligned} x &= a2^s + b \\ y &= c2^s + d \\ xy &= (ac)2^{2s} + (ad+bc)2^s + bd \end{aligned}$$

Note-se que os 4 produtos têm $2s$ (ou $2s+1$) de comprimento.



A adição das 4 parcelas e tratamento do "carry" vão ser $O(n)$. Vamos desprezá-las, o que vai acabar por dar uma ordem de grandeza correcta. Supomos n da forma $n_0 \times 2^p$.

Multiplicação com precisão arbitrária

Recorrência:

$$\begin{cases} t(n_0) = k \\ t(n) = 4t(n/2) \quad (n \geq 2) \end{cases}$$

Exercício 36 *Resolva a recorrência, mostrando que não houve qualquer melhoria da ordem de grandeza.*
<u>Nota</u>. Se levassemos em conta os factores constantes, veríamos que a eficiência é menor! $\diamond$

O "dividir para conquistar" não nos ajudou... Até que alguém descobriu
Boa ideia: Bastam 3 multiplicações: seja $p = (a+b)(c+d)$. Vem

$$xy = (ac)2^{2s} + (p - ac - bd)2^s + bd$$

Bastam 3 multiplicações!

Multiplicação com precisão arbitrária

O algoritmo:

```
large integer prod(x,y){
  n = max(|x|,|y|);
  if(n<=n0)
    return (xy);   // (directo)
  let x=a*2^s + b, y=c*2^s+d;
  p1=prod(a,c);
  p2=prod(b,d);
  p3=prod(a+b,c+d);                        (*)
  return( p1*2^(2s) + (p-p1-p2)*2^s + p2);  (*)
}
```

Notas:

- As contas do tipo "*2^s" não se fazem, trata-se da manipulação de bits (ou de dígitos).
- Nas linhas marcadas (*) há que efectuar adições ou subtracções!

Multiplicação com precisão arbitrária

Análise da eficiência. Recorrência:

$$\begin{cases} t(n_0) = k \\ t(n) = 3t(n/2) \quad (n \geq 2) \end{cases}$$

Se procurarmos solução da forma αn^k vem

$$\alpha n^k = 3\alpha(n/2)^k$$

o que fica satisfeito com $k = \log 3$. Usando a primeira parte da recorrência podemos determinar α. Trata-se de uma melhoria, temos agora que o tempo é $O(n^{\log 3} \approx n^{1.59}$.

Porque não usar este algoritmo no dia a dia? Porque, devido à sua complexidade só vale a pena para valores grandes de $|n|$, experimentelmente para $|n| \geq$ 500 dígitos.

Ainda é possível multiplicar de forma mais eficiente utilizando um algoritmo conhecido por FFT ("fast Fourier transform"), obtendo-se $O(n \log n \log\log n)$.

Pesquisa binária

Algoritmo conhecido que procura um valor x num vector ordenado v[ ].

```
pbin(int a, int b, int x){
  // o array v[ ] é global
  if(a>b) return(NO);
  m=(a+b)/2;  // quociente inteiro!
  if(x==v[m]) return(m);
  if(x<v[m])  return(pbin(a,m-1,x));
  return(pbin(m+1,b,x));
}
```

Recorrência para o número de comparações, supondo a forma $n = 2^p - 1$ (1 comparação pode dar 3 resultados, "==", "<" ou ">".

$$\begin{cases} c(0) = 0 \\ c(n) \leq 1 + c((n+1)/2 - 1) \quad (n \geq 1) \end{cases}$$

Solução (mudar a variável para p, $n = 2^p - 1$): $c(n) \leq \log(n+1)$. No caso geral (exercício!) temos

$$c(n) \leq \lceil \log(n+1) \rceil$$

Determinação da mediana

Seja "a[ ]" um vector não ordenado com n elementos distintos. É sabido que

1. Podemos determinar o menor valor de "a[ ]" com $n - 1$ comparações.

2. Podemos determinar o segundo menor valor de "a[ ]" com cerca de $2n$ comparações.

Como determinar o valor de ordem de ordem k, $0 \leq k < n$ eficientemente?
Seja a função "select(k)". Para $k = \lfloor n/2 \rfloor$ trata-se do problema de detrminar a **mediana**

Um processo é ordenar o vector e a eficiência para todo o k de "select(k)" é $O(n \log n)$, o que já não é mau mas... poderemos fazer melhor?

Determinação da mediana

Existe um processo semelhante ao "quicksort" com tempo médio $O(n)$!

```
int select(int i, int j, int k){
  // Selecciona o elemento de ordem k do vector a[ ]
  // que sabemos estar entre os índices i e j
  if(i==j) return(a[i]);
  m=split(i,j);
  // a[1] fica na posição m.
  // Os elementos < a[1] nas posições i,...,m-1
  // Os elementos > a[1] nas posições m+1,...,j
  if(k==m-i)
    return(a[m]);
  if(k<m-i)
    return(select(i,m-1,k));
  else
    return(select(m+1,j,k-(m-i)-1));
}
```


Eficiência da determinação da mediana

Eficiência: a análise é um pouco complicada. De modo simplista, suponhamos que em média cada chamada a select actua numa fracção $f < 1$ do vector relativamente ao tamanho anterior (por exemplo, a nova parte tem, em média, não mais de 90% da anterior). Vem, para um certo c:

$$t(n) \leq t(fn) + cn$$

Pode-se mostrat que esta recorrência permite concluir que $t(n) \in O(n)$.

No caso pior, a eficiência é má, $O(n^2)$. Mas é possível transformareste algoritmo por forma a continuar a garantir $O(n)$.

Programa completo (início)

```c
#include <stdio.h>
#define MAX  9

int a[] = {8,11,2,6,5,9,20,3,32,7};
int select(int, int, int); // globalmente em a[ ]

main(){
  int k;
  while(1){
    do{
      printf("Elemento de ordem? ");
      scanf("%d",&k);
    }
    while(k<0 || k>MAX);
    printf(" = %d\n",select(0,9,k));
  }
}
```


Programa completo (fim)

```c
int split(int, int);
int select(int i, int j, int k){
  int m;
  if(i==j)
    return(a[i]);
  m=split(i,j);
  if(k==m-i)
    return(a[m]);
  if(k<m-i)
    return(select(i,m-1,k));
  else
    return(select(m+1,j,k-(m-i)-1));
}

// Grupo da esquerda <= a[i], da direita > a[i]
// Retorna o indice de a[i], a "meio"
int split(int i, int j){
  int l=i+1, r=j, v=a[i], t;
  while(l<=r){
    while(l<=r && a[l]<=v) l++;
    while(l<=r && a[r]>v)  r--;
    if(l<r){ t=a[l]; a[l]=a[r]; a[r]=t; }
  }
  t=a[i]; a[i]=a[r]; a[r]=t;
  return(r);
}
```

Tabulação

Ideia:

- Sempre que $f(x) = y$ é calculado, guarda-se o par (x, y).

- Sempre que $f(x)$ é chamado, vê-se primeiro se já existe em memória um par (x, y). Se existir, retorna-se y; se não, calcula-se.

A tabelação pode ser generalizada a:

- Relações binárias.
- Memória com "esquecimento": devido a uma capacidade limitada só se guardam, por exemplo, os valores mais recentes.
- Programação com unificação (Prolog...).

Aplicação ao "Fibonacci"

Fibonacci, $f(x)$:
Método directo: $O(n^{(\sqrt{5}+1)/2})$
Uso da tabelação: $O(n)$

Claro que com um método iterativo simples também se consegue $O(n)$. Mas a tabelação é um método *sistemático* e *pode ser automática*.

Programa:

```
   #define NAO_SEI  (-1)
   int fib[MAX];
   ...
   for(i=0;i<MAX;i++) fib[i]=NAO_SEI;
   ...
   int f(int n){
     int t;
     if(n==0)   return(0);
     if(n==1)   return(1);
     if(n<MAX && fib[n]!=NAO_SEI) return(fib[n]);
     t=f(n-1)+f(n-2);
     if(n<MAX) fib[n]=t;
     return(t);
}
```

Exercício 37 *Considere o programa anterior e suponha que* `MAX` *é arbitráriamente grande.*

1. (a) *Simule a execução de uma chamada "*`f(5)`*".*

   (b) *Mostre que o número total de adições efectuadas numa chamada "*`f(n)`*" é* $\mathtt{n}-1$.

   (c) *Calcule o número total de chamadas a* $f()$ *causadas por a uma chamada "*`f(n)`*" (incluindo a própria chamada).*

   (d) *Qual a ordem de grandeza do algoritmo?*

◇

Considerações sobre o modo de guardar os pares (x, y) já calculados.

1. Se x tomar valores inteiros, não negativos e relativamente pequenos, pode ser tomado como índice de um vector garantindo eficiência $O(1)$ na obtenção dos valores de y. Teremos o seguinte onde x pertence ao domínio e y ao contradomínio $\mathcal{D}$:

$$[0 \cdots MAX] \rightarrow \mathcal{D}$$

2. Se não for o caso, é possível por vezes mudar de variável, $x \rightarrow x'$ por forma a que x' possa ser índice do vector.

3. No caso geral, há que usar técnicas de implementação da estrutura de dados Dicionário como, por exemplo os métodos de "hashing".

Princípio da optimalidade: Qualquer parte de uma solução óptima é óptima.

Esquema geral: Para resolver um problema de optimização de tamanho n

1. Resolvem-se todos os problemas relevantes de tamanho n_0, por exemplo $n_0 = 1$.

2. Para $k = n_0 + 1, n_0 + 2, \cdots, n$ resolvem-se todos os problemas relevantes de tamanho k. Quando for $k = n$ só há um problema resolvido que é o pedido.

Note-se que a solução é obtida por um método "bottom-up" onde, para cada $n > n_0$ se podem utilizar as soluções já calculadas de sub-problemas de tamanho $< n$.

Exercício 38 *Pretende-se caminhar do ponto* (x_1, y_1) *para o ponto* (x_2, y_2)*, por passos unitários em* x $(x_1 < x_2)$ *e nunca saindo do rectângulo que este par de pontos define (as coordenadas são inteiras). O custo de caminhar do ponto* (x, y) *para um dos 3 pontos* $(x + 1, y(\pm 1))$ *é uma função conhecida de* y*.*

Descreva um algoritmo baseado na programação dinâmica que determine o custo mínimo (e eventualmente a caminhada correspondente) de uma viajem de (x_1, y_1) *para* (x_2, y_2)*. O algoritmo deverá calcular caminhos parciais óptimos de pontos* (x, y) *para pontos* $(x + n, y')$ *com* n *sucessivamente igual a 0 (função dada), 1,. . . ,* $x_2 - x_1$*.*
$\diamond$

Multiplicar matrizes com esforço mínimo

O problema. São dadas n matrizes a multiplicar, sejam M_1, M_2,. . . , M_n. As suas dimensões são respectivamente $a_0 \times a_1$, $a_1 \times a_2$,. . . , $a_{n-1} \times a_n$ (dizemos que uma matriz tem dimensão $a \times b$ quando tem a linhas e b colunas). Pretende-se associar as matrizes (introduzindo parêntesis) por forma que o número de multiplicações elementares efectuadas seja mínimo.

O produto de matrizes é associativo mas a sua eficiência, medida pelo número de multiplicações elementares, não o é!

Exercício 39 *Mostre que a multiplicação (pelo método usual) de 2 matrizes de dimensões respectivamente $a \times b$ e $b \times c$ envolve abc multiplicações elementares (ordem $O(n^3)$ se $n = a = b = c$).* $\diamond$

Exemplo. Sejam as dimensões de M_1, M_2 e M_3 respectivamente 20×10, 10×100, 100×50. Temos

Produto	N. de multiplicações elementares
$(M_1 M_2) M_3$	$20 \times 10 \times 100 + 20 \times 100 \times 50 = 120000$
$M_1 (M_2 M_3)$	$20 \times 10 \times 50 + 10 \times 100 \times 50 = 60000$

Considerando todos os produtos possíveis

Exercício 40 *Quantos modos há de associar um produto de n factores? Por exemplo*
- Para $n = 3$, temos 2 modos: "$(ab)c$" e "$a(bc)$"
- Para $n = 4$, temos 5 modos: "$((ab)c)d$" e "$(a(bc))d$", "$(ab)(cd)$", "$a((bc)d)$" e "$a(b(cd))$".
Sugestão Seja $m = n-1$. Mostre que o número pedido, $f(m)$, satisfaz $f(0) = f(1) = 1$ e, para $m \geq 2$, $f(m) = \Sigma_{k=1}^{m-i} f(k) f(m-k)$. Verifique que $f(m) = \frac{1}{m+1} \binom{2m}{m}$ satisfaz esta recorrência. Temos, por exemplo, para $n = 4$, é $m = 3$ e temos

$$f(3) = \frac{1}{4} \binom{6}{3} = \frac{20}{4} = 5$$

$\diamond$

Exercício 41 *Mostre que $f(n)$ (ver problema anterior) é exponencial. Pode usar a aproximação se Stirling que é suficientemente exacta:*

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

$\diamond$

Usando a programação dinâmica

Vamos conseguir, com menos trabalho, obter a "parentização" óptima. Seja o produto

$$M_1 M_2 \cdots M_n$$

Vamos considerar as multiplicações óptimas 2 a 2; depois 3 a 3, etc. Em geral, o valor óptimo de um produto parcial

$$\begin{array}{cccc} M_i & M_{i+1} & \cdots & M_j \\ a_{i-1} \times a_i & a_i \times a_{i+1} & & a_{j-1} \times a_j \end{array}$$

é o mínimo

$$v(i,j) = \min_{k=i+1,\cdots,j} v(i,k-1) + v(k,j) + a_{i-1}a_k a_j$$

Usando a programação dinâmica

M_1 20×10	M_2 10×100	M_3 100×50	M_4 50×2
M_1M_2: 20000	M_2M_3: 50000	M_3M_4: 10000	
$\mathbf{M_1M_2M_3}$ Mínimo de $(M_1M_2)M_3$: $20000 + 100000$ $M_1(M_2M_3)$: $50000 + 10000$ $= 60000$		$\mathbf{M_2M_3M_4}$ Mínimo de $(M_2M_3)M_4$: $50000 + 1000$ $M_2(M_3M_4)$: $10000 + 2000$ $= 12000$	
$\mathbf{M_1M_2M_3M_4}$ Mínimo de $(M_1M_2M_3)M_4$: $60000 + 2000$ $(M_1M_2)(M_3M_4)$: $20000 + 10000 + 4000$ $M_1(M_2M_3M_4)$: $12000 + 400$ $= 12400$			

O número mínimo de multiplicações elementares é de 12400 e o produto correspondente é

$$M_1(M_2(M_3M_4))$$

Usando a programação dinâmica

Exercício 42 *Implemente este algoritmo em linguagem C.* $\diamond$

Exercício 43 *Altere o algoritmo por forma a que determine não só o custo mínimo da multiplicação mas a "parentização" correspondente. Basta, para cada sub-produto óptimo, marcar a "parentização" do topo.* $\diamond$

Exercício 44 *Mostre que a eficiência* da determinação *do modo óptimo de multiplicar matrizes (não "do modo óptimo de multiplicar matrizes") é de ordem $O(n^3)$.*
<u>*Sugestão*</u>*. O número de comparações efectuadas é*

$$n-2+2(n-3)+\cdots+(n-3)2+(n-2)$$

correspondentes sucessivamente a sub-produtos de tamanho 3, 4,..., n. Mostre que esta soma é um polinómio de grau 3. $\diamond$

O exercício anterior mostra que com a programação dinâmica conseguimos uma melhoria

$$\text{Exponencial} \to O(n^3)$$

Sobre os vários esquemas...

Aplicação ao problema das matrizes.

O "dividir para conquistar" funciona do topo para a base ("top-down").

A "programação dinâmica" funciona da base para o topo ("bottom-up"). to topo para a base.

Neste caso o "dividir para conquistar" é menos eficiente uma vez que resolve sub-problemas repetidas vezes.

Mas o "dividir para conquistar" + "tabulação" têm uma eficiência semelhante à da "programação dinâmica". Diferem essencialmente na ordem da análise ("top-down" no primeiro caso, "bottom-up" no segundo).