Estruturas de dados e algoritmos

Algumas notas

Armando B. Matos

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

2000

Estruturas de dados e algoritmos

- Disciplina do 2^o ano da licenciatura em Ciência de Computadores.

- Pressupõe conhecimentos correspondentes às disciplinas de "Programação Imperativa" e "Programação Estruturada": algoritmos, linguagem de programação C (ou outra semelhante) e alguma maturidade matemática.
  → *Aconselha-se aos alunos que não frequentem "Estruturas de Dados e Algoritmos" sem terem tido aprovação nestas duas disciplinas (ou, pelo menos, numa delas).*

Programa

- Estruturas de dados - representação da informação
- Algumas estruturas de dados, sua implementação e aplicações:
  - Listas lineares: pilhas, filas, "deques"
  - Filas de prioridade. Aplicações. "Heaps", "heapsort"
  - Dicionários. Métodos baseados em
    - Vectores ordenados e não ordenados
    - Vectores de bits
    - "Tries"
    - Árvores binárias de pesquisa
    - Árvores 2-3 e árvores B
    - "Hash"
- Introdução à análise da eficiência dos algoritmos.

Programa - continuação

- Algumas técnicas gerais de desenvolvimento de algoritmos
  - "Dividir para conquistar"
  - Tabulação
  - Programação dinâmica
- Algoritmos em domínios específicos
  - Ordenação e pesquisa
  - Grafos
  - Pesquisa em grafos ímplicitos (jogos,. . .)
  - "Strings"
- Algoritmos aleatorizados
- Complementos da linguagem C, Introdução ao C++.

Pequena (mas boa) bibliografia

- Brassard and Bratley. *Algorithmics, Theory and Practice*, Prentice-Hall International Editions, 1988.

- Aho, Hopcroft and Ullman, *Data Structures and Algorithms*, Addison Wesley, 1984.

- Cormen, Leiserson, Rivest, *Introduction to Algorithms*, McGraw-Hill, 1997.

- Apontamentos diversos fornecidos.

Avaliação

Regras de avaliação aplicáveis a todos os alunos e épocas de de exame.

- Valorização dos trabalhos práticos: 5 valores.

- Frequência na disciplina: baseada no número de faltas às aulas práticas:
  Máximo de faltas é 3

- Valorização do exame: 15 valores.

- Nota final: nota dos trabalhos + nota do exame.

(Já dado em Programação Estruturada)

Complementos da linguagem C
Macros

#define sem argumentos

Forma: **#define <ident> <xxx>**
Efeito: Daqui para a frente, **<ident>** é substituído por **<xxx>**

#define com argumentos

Forma: **#define <ident>(<P1>,...,<Pn>) <xxx>**
Efeito: Daqui para a frente,
<ident>(<A1>,...,<An>) é substituído por **<xxx>** com as ocorrências de **<P1>** em **<xxx>** substituídas por **<A1>**, as de **<P2>** por **<A2>**, etc.

Exemplos de `#define#`

Exemplos

1. `#define ATAS 1000`
   - `int a[ATAS], b[2*ATAS], BATATAS;`
2. `#define DOUBLE(X) X+X`
   - `int z=DOUBLE(y);`
   - `w=DOUBLE(2*y);`

Exercício 1 *Escreva um "macro" que teste se o parâmetro tem o bit 2 igual a 1 ou a 0.*

$$BIT(X) = \begin{cases} 1 & \textit{se o bit 2=1} \\ 0 & \textit{se o bit 2=0} \end{cases}$$

Nota O bit 0 é o menos significativo.

Uma solução

```
#define BIT2(X) (X & 4 != 0)
```

Explique porque é conveniente o uso dos parêntesis, indicando uma situação em que não se obtém o que o programador possivelmente esperaria...

O pré-processador de C

$\texttt{x.c} \xrightarrow{\text{gcc}} \texttt{a.out}$

...é de facto muito mais complexo...

$\texttt{x.c} \xrightarrow{\text{cpp}} \ldots \longrightarrow \ldots \xrightarrow{\text{“loader”}} \texttt{a.out}$

`cpp`: Pré-processador do C, trata das linhas começadas por `#` - `#define`, `#include`,...

O pré-processador de C

Como ver o efeito do `cpp`?
- Parando a compilação depois do `cpp` actuar, usando "`gcc -E...`"

```
teste.c:
--------------------------------------------------
#include <stdio.h>

#define VAL 1000
#define SVAL(A) A+1000
#define MAX(A,B)  (A>=B? A : B)

int a[VAL], b=1, c=2;

main(){
  if(b>0)
    c=MAX(b,c);
  d=2*SVAL(4);
}
--------------------------------------------------
```

O pré-processador de C

```
-------------------------------------------------
gcc -E test.c > lixo
lixo:
-------------------------------------------------
....
typedef long _G_clock_t;
...
extern int fgetc  (FILE *)  ;
...
int a[1000 ], b=1, c=2;
main(){
  if(b>0)
    c= ( b >= c ?  b  :  c ) ;
  d=2*  4 +1000 ;
}
```


Tópicos sobre macros

- Vantagens em definir constantes com "`#define`"
- Diferença entre "macros" e "funções": entre
  `#define MAX(A,B)  (A>=B? A : B)` e
  `int max(int a, int b){...}`
  Eficiência, tipo dos parâmetros,...
- Funçoes "inline".
- "const" no C++.
- Existem muitas linguagens baeadas em conceitos relacionados co o de "macro" ("patter matching", regras de reescrita...).

Operações lógicas bit a bit e "shifts"

Preferível trabalhar com "`unsigned`".

- `a | b`: "ou", bit a bit.
- `a |&b`: "e", bit a bit.
- `a ^ b`: "ou" exclusivo, bit a bit.
- `~a`: troca todos os bits.
- `a<<b`: "`a`" é deslocado ("shift") para a esquerda de "`b`" bits – é multiplicado por 2^b.
- `a>>b`: "`a` é deslocado ("shift") para a direita de "`b`" bits – é dividido (divisão inteira) por 2^b.

Exercícios

Exercício 2 *Quais os valores de "`1<<10`" e de "`1>>10`"?*

Exercício 3 *O que acontece ao valor de "`a`" após a execução da instrução "`a ^= 31`"?*

Exercício 4 *Defina uma função "`muda(char *a, int b)`" que troca o bit número "`b`" do vector "`a`", sendo o bit 0 o menos significativo de "`a[0]`",..., o bit 8 o menos significativo de "`a[1]`", etc.*

O bom estilo de programação
Algumas notas

Clareza e simplicidade. Porquê?

- O programa vai possivelmente ter de ser compreendido mais por você ou por outras pessoas, por exemplo para
  - "Debug".
  - Alterações.
  - Melhoria de comportamento.
- O programa tem de ser, tanto quanto possível, fiável. A sua *demonstração* deve ser simples.
- Tratando-se de uma equipa de vários programadores, a interface do seu programa com os dos outros deve ser clara, simples e, se possível formal.

E não se esqueça: um programa vai ter de ser comprendido:
– Pelo computador.
– Por pessoas.

Como o conseguir

- Interfaces claras e pequenas.
- Invisibilidade da implementação.
- A "programação modular ("estruturada", por objectos, etc.) pode ajudar – ou não.
- Os contextos devem ser pequenos e simples.
- Valores associados devem der agrupados em estruturas ("`struct`").

Como o conseguir

- O texto das funções deve ser curto – max 1 pág. A4.
- A função das funções deve ser clara e comentada.
- Os ciclos devem acaber de forma óbvia.
- Deve ser simples (com a eventual ajuda de comentários), olhar para um ponto do programa e fazer asserções lógicas significativas para a sua compreensão.
- O uso da recursividade é uma importante ferramenta de clarificação.
- Os comentários devem ser parcimoniosos e úteis à compreensão do programa.
- As constantes numéricas devem aparecer definidas apenas num ponto do programa.
- O programa deve ser transportável – funcionar em todas as máquinas.

Péssimo exemplo: grandes contextos

```
int i,j,k,a[100];   /* globais  */
...
void processa(void){
  for(i=0;i<1000;i++){
    ...
  }
}
```

Em C:

- Usar poucas variáveis globais (por exemplo < 10).
- Usar parâmetros.
- Nunca usar variáveis que não interessam a mais ninguém (por exemplo, contadors de ciclo) que sejam variáveis globais.
- Verificar que as variáveis que se pretendem locais estão definidas dentro da função. De outro modo, poderiam existir variáveis globais com o mesmo nome.

Péssimo exemplo: ciclos que só Deus sabe como acabam

```
  while(1){
   i=j+4;
   if(i>100)
     break;
   proc(&i);
   if(i<200)
     break;
   if(flag3==27){
     i+=4;
     goto fim;
     }
   printf("Estou aqui!\n'');
   }
fim: ...  ...
```

Um ciclo "`while`" na medida do possível, só deve acabar quando a condição se tornar falsa. Por exemplo,

```
  while((c=getc(file))!=EOF && c>'A'){
    putc(modifica(c),file);
  /* Asserção: EOF(file) || c<='A' */
```

Já agora, os valores lidos por "`getc`" são **inteiros** para permitir distinguir EOF de qualquer caracter!

Ciclos: muitos modos equivalentes

Todos os programas seguintes copiam o "standard input" para a "standard output". A variável "`c`" é um *inteiro*.
A minha classificação estética vai de `[0]` a `[5]`.

`[2]` "`while(1)`" com "`break`" no meio

```
  while(1){
   c=getchar();
   if(c==EOF)
     break;
   putchar(c);
  }
```

`[5]` Apenas "`do...while(...)`"

```
  do{
    c=getchar();
    if(c!=EOF)
      putchar(c);
  }
  while(c!=EOF);
```

Ciclos: muitos modos equivalentes - continuação

[5] Aproveitando o facto de uma condição poder ser uma expressão. . .

```
while((c=getchar())!=EOF)
  putchar(c);
```

---

[0] Métodos ilegíveis

```
while((c=getchar())!=EOF?(putchar(c),1):0);
```

---

[0] Métodos à moda do "assembler".

```
l1: c=getchar();
    if(c==EOF) goto l2;
    putchar(c);
    goto l1;
l2: continue;
```


Constantes numéricas

As constantes numéricas devem aparecer definidas apenas num ponto do programa, por exemplo, com

```
#define  X 1000
```

Maior facilidade de modificação e compreensão.

Recursividade

O uso da recursividade é uma importante ferramenta de clarificação e simplificação de programas.

Já viram algoritmos como o "quicksort", o "mergesort" ou a listagem de árvores binárias implementados com e sem recursividade?

A história do "quicksort" do seu inventor C. A. R. Hoare.

Uma função recursiva – como uma definição indutiva – tem sempre uma forma do tipo

```
 ... rec(...){
   if(...){
     ...        /* Caso base, trata dele e retorna */
     return(...);
   }
   ...
   rec(...)     /* possíveis chamadas recursivas   */
   ...          /* com argumentos mais simples     */
}
```


Exercício sobre recursividade

Exercício 5 *Para a função representada e com a chamada externa "`bo(3)`",*

1. (a) *Represente numa árvore a sucessão de chamadas à função, representando, para cada chamada os valores de "`x`" e de "`a`".*

   (b) *O que imprime o programa? Confirme no computador!*

   (c) *Mostre que qualquer chamada à função termina. Use indução no parâmetro "`x`".*

```
void bo(int x){
  int a;
  a=x+1;
  if(x<=1){
    printf("*\n");
    return;
  }
  printf("%3d%3d\n",x,a);
  x=x-1;
  bo(x-1);
  bo(x);
  printf("pi\n");
```

◇

Exercício sobre recursividade

Exercício 6 *Considere a seguinte função.*

```
int mdc(int a,int b){
  if(b==0)
    return(a);
  return(mdc(b,a%b));
}
```

1. (a) *Siga o programa para as chamadas*
   i. `mdc(5,0)`
   ii. `mdc(15,21)`

   (b) *Em geral, o que calcula a função? Justifique.*

   (c) *Escreva uma função equivalente usando iteratividade (em vez de recursividade).*

◇

Análise de algoritmos

- Introdução
- Ordens de grandeza
- Recorrências e suas soluções

Eficiência dos algoritmos

Caso geral: Tempo e espaço; consideramos só o tempo.

$$x \text{ (dados)} \xrightarrow{\text{algoritmo}} y \text{ (resultado)}$$

Para os dados x demora o tempo $t(x)$. Seja $|x|$ o comprimento de x.
Análise no pior caso:

$$T(n) = max\{t(x) \mid |x| = n\}$$

(não definido se não há nenhum x com $|x| = n$)
Análise no caso médio

$$\overline{T(n)} = \mu\{t(x) \mid |x| = n\}$$

Os parâmetros n e t têm por vezes outro significado, como veremos...

Eficiência dos algoritmos análise de um programa

```
for(i=1;i<n;i++){      t1,n t2,n-1 t3
  min=i;               n-1 t4
  for(j=i+1;j<=n;j++)  ?   t5,t6,t7
    if(a[j]<a[min])    K   t8
       min=j;          <=K t9
  t=a[i];              n-1 t10
  a[i]=a[min];         n-1 t11
  a[min]=t;            n-1 t12
}
```

$$\begin{aligned} K &= (n-1)+(n-2)+\cdots+1 = n(n-1)/2 \\ T &\leq t_1+t_2+ \\ &\quad +(n-1)(t_2+t_3+t_4+t_5+t_{10}+t_{11}+t_{12})+ \\ &\quad +K(t_7+t_8+t_9)+(K+n-1)t_6 \end{aligned}$$

Eficiência dos algoritmos
análise de um programa

$$T \leq t_1 + t_2 + (n-1)t' + \frac{n(n-1)}{2}(t'' + t_6)$$

com

$$\begin{aligned} t' &= t_2 + t_3 + t_4 + t_5 + t_6 + t_{10} + t_{11} + t_{12} \\ t'' &= t_6 + t_7 + t_8 + t_9 \end{aligned}$$

ou seja, T é da forma

$$T(n) \leq \alpha n^2 + \beta n + \gamma$$

Conclusões

1. A análise não foi completa. Isso exige métodos probabilísticos.
2. O tempo total é majorado por um polinómio do grau 2 em n.
3. O tempo correspondente ao algoritmo de ordenação dado cresce essencialmente com o quadrado de n, no sentido em que existe n_0 e constante $c > 0$ tal que

$$\forall n \geq n_0 \;\; T(n) \leq cn^2$$

Eficiência dos algoritmos
Considerações

A análise dos algoritmos raramente é completa. Porquê?

1. Há valores desconhecidos ou não fixos (t_i).
2. Uma mudança de processador/compilação alteraria os resultados!
3. Não interessa uma informação tão pormenorizada.

Em geral basta majorar o tempo por uma função simples dos parâmeros, por exemplo $T \leq cn^2 \log n$.

Ordem de grandeza $O(n)$

Dizemos que $g(n)$ é $O(f(n))$ se existem $c > 0$ e n_0 tais que

$$\forall n \geq n_0 \;\; g(n) \leq cf(n)$$

$O(f(n))$ é a classe de funções de ordem $f(n)$:

$$O(f(n)) = \{g(n) \mid \exists c > 0, \exists n_0, \; \forall n \geq n_0 \; g(n) \leq cf(n)\}$$

Assim,

$$\text{“}g(n) \text{ é } O(f(n))\text{”} \quad \equiv \quad \text{“}g(n) \in O(f(n))\text{”}$$

Ordem de grandeza $O(n)$, exercícios

Exercício 7 *Averigue se cada uma das seguintes afirmações é verdadeira. Justifique.*
(a) $2n^2 + n$ *é* $O(n^2)$ *(b)* $2n^2 + n$ *é* $O(n^3)$
(c) $2n^2 + n$ *é* $O(n)$ *(d)* $n \log n \in O(n^2)$
(e) 22 *é* $O(1)$ *(f)* $O(n) \subseteq O(n^2)$ $\diamond$

Exercício 8 *Mostre que todo o polinómio*

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$$

é de ordem n^k. $\diamond$

Exercício 9 *Indique por palavras o significado da afirmação* $f(n)$ *é* $O(1)$. $\diamond$

Exercício 10 *Averigue se a relação entre funções “$g(n)$ é $O(f(n))$” tem as propriedades*

1. *(a) Reflexiva*
   (b) Simétrica
   (c) Anti-simétrica
   (d) Transitiva

$\diamond$

Ordem de grandeza $\Omega()$ e $\Theta(n)$

– $O(\,)$: Majoração, $g(n)$ é $O(f(n))$ sse

$$\exists c > 0, \exists n_0,\ \forall n \geq n_0\ g(n) \leq cf(n)$$

$$O(f(n)) = \{g(n) \mid g(n) \text{ é de ordem } O(f(n))\}$$

– $\Omega(\,)$: Minoração, $g(n)$ é $\Omega(f(n))$ sse

$$\exists c > 0, \exists n_0,\ \forall n \geq n_0\ g(n) \geq cf(n)$$

$$\Omega(f(n)) = \{g(n) \mid g(n) \text{ é de ordem } \Omega(f(n))\}$$

– $\Theta(\,)$: Ordem exacta, $g(n)$ é $\Theta(f(n))$ sse

$$\exists c_1 > 0,\ \exists c_2 > 0,\ \exists n_0,\ \forall n \geq n_0 \quad c_1 f(n) \leq g(n) \leq c_2 f(n)$$

$$\Theta(f(n)) = \{g(n) \mid g(n) \text{ é de ordem } \Theta(f(n))\}$$

Ordens de grandeza, exercícios

Exercício 11 *Averigue se cada uma das seguintes afirmações é verdadeira. Justifique.*
(a) $2n^2 + n$ *é* $\Theta(n^2)$ *(b)* $n \log n$ *é* $\Omega(n^2)$
(c) $n \log n$ *é* $\Omega(n)$ *(d)* 2^n *é* $\Omega(n^5)$ $\diamond$

Exercício 12 *Mostre que todo o polinómio*

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$$

com $a_k > 0$ *é de ordem exacta* n^k. $\diamond$

Exercício 13 *Indique por palavras o significado da afirmação* $f(n)$ *é* $\Theta(1)$. $\diamond$

Exercício 14 *Mostre que para toda a função* $f(n)$ *é*

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

$\diamond$

Ordem de grandeza $O(n)$, exercícios

Exercício 15 *Mostre que é verdadeiro ou falso, conforme indicado*

(a) $\Omega(n^3) \subset \Omega(n^2)$ *(verdadeiro)*

(b) $\Omega(n) \subseteq \Omega(n \log n)$ *(falso)*

(c) $\Theta(n) \cap \Theta(n^2) = \emptyset$ *(verdadeiro)*

(d) Para toda a função $f(n)$*,* $\Theta(f(n)) \subseteq O(f(n))$ *Quando ocorre a igualdade? (verdadeiro)*

(e) Para toda a constante $k > 0$*,* $\Theta(f(n)) = \Theta(kf(n))$ *(verdadeiro)* $\diamond$

Panorâmica

Classificação das funções/algoritmos

Polinomial	$\exists k,\ f(n) \in O(n^k)$
Super-polinomial (e sub-exponencial)	$\forall k,\ f(n) \in \Omega(n^k)$ e $\forall a > 1, f(n) \in O(a^n)$
Exponencial	$\exists a > 1, f(n) \in O(a^n)$ e $\exists b > 1, f(n) \in \Omega(b^n)$
Hiper-exponencial	$\forall a > 1, f(n) \in \Omega(O(a^n))$

Os algoritmos com tempo de execução polinomial são considerados (nesta classificação) *eficientes* e os outros *intratáveis*.

Teorema. Seja $k \geq 0$ inteiro e $a > 1$ real. Então existe n_0 tal que

$$\forall n \geq n_0,\ \ n^k < a^n$$

Por palavras: uma função "exponencial" cresce mais rapidamente que qualquer polinómio.

Análise de algoritmos e recorrências

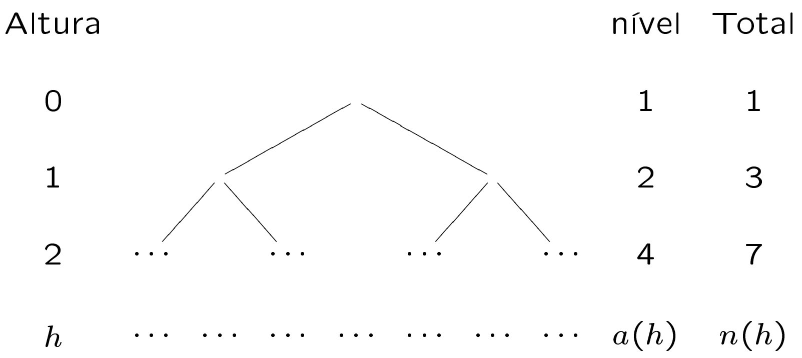
A análise de um algoritmo leva muitas vezes à definição do tempo de execução através de recorrências. Se possível, é necessário resolvê-las!

Exercício 16 *Consideremos uma árvore binária completa até uma certa altura h. Escreva recorrências e resolva-as para*

1. *(a) O número de nós* <u>*num*</u> *determinado nível, $a(h)$.*

   (b) O número de nós <u>*até um*</u> *determinado nível, $n(h)$.*

◇



Solução da alínea (b) Supomos a alínea (a) resolvida, $a(h) = 2^h$. Temos a recorrência

$$\begin{cases} n(0) = 1 \\ n(h) = n(h-1) + 2^h \quad (h \geq 1) \end{cases}$$

Temos

$$n(h) = 2^h + 2^{h-1} + \cdots + 1 = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

Confirmamos para os primeiros valores de h (confirmar!).
Mostramos que a solução está correcta *por indução* em h

- $h = 0$, $2^{0+1} - 1 = 1$, correcto segundo a recorrência.

- $h \geq 1$, supomos válido para h (hipótese indutiva) e mostramos que é válido para $h+1$. Pela recorrência,

  $$n(h+1) = n(h) + 2^{h+1} = 2^{h+1} - 1 + 2^{h+1} = 2^{h+2} - 1$$

  Ao substituir $n(h)$ por $2^{h+1} - 1$ usamos a hipótese indutiva. A forma é a mesma e fica demonstrado.

Observemos que $n(h)$ tem comportamento "exponencial", não sendo $O(h^k)$ para nenhum k. Cresce muito depressa, por exemplo uma árvore com altura 40 tem

$$2^{40} - 1 = 1099511627775 > 10^{12}$$

nós, muito mais que os habitantes da Terra!
Um algoritmo com tempo de execução $n(h)$ microsegundos, demoraria mais de um mês (!) a resolver um problema com o modesto tamanho $h = 40$.

A solução das recorrências deve ser exacta. Um método (exacto) que se usa com frequência para resolver uma recorrência que define $f(n)$ é

1. (Tabelar) Calcular os primeiros valores, $f(0)$, $f(1)$,..., $f(k)$.

2. (Suspeitar) Analisar os valores calculados à procura de uma possível forma fechada para $f(n)$.

3. (Demonstrar) Demonstrar por indução em n que a solução de que suspeitamos está correcta... ou então mostrar que não está e voltar ao passo (2).

Nota. Há também métodos directos de resolver certas classes de recorrências.

Uma recorrência (do "mergesort")

Consideramos uma recorrência que caracteriza um majorante do número de comparações efectudas pelo método de ordenação "mergesort" (breve explicação)

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

Se substituirmos "$\leq$" por "$=$", a recorrência define um majorante de $c(n)$.
$c(n)$ (ou melhor, o majorante) só está definido quando n é uma potência de 2.

Tomando o majorante e supondo $n = 2^p$ (mudança de variável)

$$\begin{cases} c(0) = 0 \\ c(p) = 2c(p-1) + 2^p - 1 \quad (p \geq 1) \end{cases}$$

1-Tabelar
A solução não é imediata... Vamos seguir os passos indicados...
Tabelar os primeiros valores (usamos p em vez de n)

p	$2^p - 1$	$c(p)$
0		0
1	1	1
2	3	5
3	7	17
4	15	49
5	31	129
6	63	321

2-Suspeitar
Bom... $c(p)$ parece crescer mais depressa que 2^p, talvez seja $p2^p$...
Por exemplo, $c(6) = 321$, $6 \times 2^6 = 384$... mas $5 \times 2^6 = 320 = c(6) - 1$.
Será $c(p) = (p-1)2^p + 1$? Funciona para $p = 0, 1, \cdots, 6$!

SUSPEITA: $c(p) = (p-1)2^p + 1$

3-Demonstrar!

Teorema. A solução da recorrência é $c(p) = (p-1)2^p + 1$

Demonstração. (por indução em p)
$p = 0$. A recorrência dá $c(0) = 0$ e a expressão $c(0) = (-1)2^0 + 1 = 0$.

$\mathcal{P}(p) \Rightarrow \mathcal{P}(p+1)$. Supomos $c(p) = (p-1)2^p + 1$ e vamos mostrar que

$$c(p+1) = ((p+1)-1)2^{p+1} + 1 = p2^{p+1} + 1$$

Temos, usando a recorrência e a hipótese indutiva

$$\begin{array}{rcll} c(p+1) & = & 2c(p) + 2^{p+1} - 1 & \text{(recorrência)} \\ & = & 2((p-1)2^p + 1) + 2^{p+1} - 1 & \text{(hip. indutiva)} \\ & = & p2^{p+1} - 2^{p+1} + 2 + 2^{p+1} - 1 & \\ & = & p2^{p+1} + 1 & \end{array}$$

que tem a forma pretendida. $\circ$

Com a variável original n, temos (logaritmo na base 2)

$$c(n) = (\log n - 1)n + 1 \in O(n \log n)$$

...ordem de grandeza dos melhores algoritmos de ordenação...

Sucessão de Fibonacci

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

Primeiros elementos:

n	0	1	2	3	4	5	6	7	8
$f(n)$	0	1	1	2	3	5	8	13	21

Exercício 17 *Mostre por indução que, para todo o* $n \geq 0$ *é* $f(n) < 2^n$. $\diamond$

Algoritmo directo

```
int f(int n){
  if(n==0)
    return(0);
  if(n==1)
    return(1);
  return(f(n-1)+f(n-2));
}
```


Análise do algoritmo

Número de adições $s(n)$, outra recorrência, parecida com a original!

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

Exercício 18 *Caracterize o número $C(n)$ de chamadas a $f(n)$ por uma recorrência. Note que há sempre uma chamada externa.* $\diamond$

Solução da recorrência de $f(n)$

O tabelamento dos primeiros valores não ajuda muito...
A sucessão cresce rapidamente com n, tentemos uma solução da forma r^n. Substituindo na recorrência (para $n \geq 2$), temos que ter

$$r^n = r^{n-1} + r^{n-2}$$

ou seja

$$r^2 - r - 1 = 0$$

Há duas soluções, $r_1 = \frac{1+\sqrt{5}}{2}$ e $r_1 = \frac{1-\sqrt{5}}{2}$ Mas como podem aparecer soluções irracionais???!!!
Notemos que a equação é homogénea, de modo que, para todo o α e β,

$$\alpha r_1^n + \beta r_2^n$$

é também solução. Será que conseguimos ajustar os coeficientes de modo a que esta combinação seja inteira (para qualquer n) e igual à sucessão de Fibonacci? Tentemos:

$$f(0) = 0 \quad \Rightarrow \quad \alpha + \beta = 0 \tag{1}$$
$$f(1) = 1 \quad \Rightarrow \quad \alpha r_1 + \beta r_2 = 1 \tag{2}$$

Solução da recorrência de $f(n)$, continuação

Exercício 19 *Determine os valores de α e β.* $\diamond$

A solução é $\alpha = \frac{1}{\sqrt{5}}$, $\beta = -\frac{1}{\sqrt{5}}$. A solução da recorrência é pois

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Exercício 20 *Confirme os valores de $f(i)$ para $i = 0, 1, 2$* $\diamond$

É espantoso como a solução de uma recorrência que define uma sucessão de valores *inteiros* tem esta forma!

Exercício 21 *Mostre que $f(n)$ tem o seguinte valor muito aproximado e majore o errro*

$$f(n) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

Escreva um programa que compare os valores exacto e aproximado de $f(n)$ para $0 \leq n \leq 10$. $\diamond$

Equações numa recorrência

Há dois tipos de equações numa recorrência

$$\text{Equações} \begin{cases} \text{Valores iniciais ou condições fronteira} \\ \text{Equação geral, para } n \geq \cdots \end{cases}$$

Exemplo Na recorrência

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

As condições fronteira são: $f(0) = 0$ e $f(1) = 1$
A equação geral é: $f(n) = f(n-1) + f(n-2)$.

Algumas técnicas de resolver recorrências

1. Tabelar, suspeitar, demonstrar

2. Diferenças de ordem k constantes Se as diferenças de ordem k são constantes, a solução é um polinómio de ordem k cujos coeficientes se determinam pelo método dos coeficientes indeterminados.
Útil na fase de "tabelação". Exemplo

n	$f(n)$	Dif 1	Dif 2
0	1		
1	3	2	
2	11	8	6
3	25	14	6
4	45	20	6
5	71	26	6

As diferenças de ordem 2 parecem ser constantes.
Propriedade Quando as diferenças de ordem k são constantes a função é um polinómio de grau k.
Neste caso tentemos a solução $f(n) = an^2 + bn + c$. Tomando os valores para $n = 0$, 1, 2, obtemos um sistema cuja solução dá: $a = ?$ $b = ?$ $c = ?$, [Exercício!].
Por indução finita demonstramos que (ou se) a solução é correcta.

3. Mudança de variável

Uma mudança de variável pode facilitar a solução.
Exemplo Seja a recorrência

$$\begin{cases} f(1) = 1 \\ f(2n) = 3f(n) + 1 \end{cases}$$

Fazemos a mudança de variável $n = 2^p$. Seja a função $f'(p)$ tal que $f(n) = f'(p)$ para $n = 2^p$. Temos, substituindo valores por valores iguais

$$\begin{cases} f'(0) = 1 \\ f'(p+1) = 3f'(p) + 1 \end{cases}$$

pois se n corresponde a p, $2n$ corresponde a $p+1$.

Exercício 22 *Resolva esta recorrência (em p), exprimindo depois a solução em termos de n.*

3. Equações lineares homogéneas

(Utilizamos este método na solução da recorrência de Fibonacci.)
Se a equação geral da recorrência é linear e homogénea, a solução toma muitas vezes a forma r^n. Seja a equação geral

$$f(n) + a_{n-1}f(n-1) + \cdots + a_m f(m) = 0$$

Se substituirmos $f(n) = r^n$, temos uma equação em r com raízes $r_1, r_2, \cdots, r_k$. Cada combinação linear de soluções é solução (porque a equação é linear e homogénea)

$$a_1 r_1^n + a_2 r_2^n + \cdots + a_k r_k^n$$

Os coeficientes determinam-se pelas condições fronteira. Se existirem raízes iguais, há que "refinar" o método.

Exercício 23 *Determine a solução da recorrência*

$$\begin{cases} f(0) = 0 \\ f(1) = 1 \\ f(n) = 7f(n-1) + 12f(n-2) \end{cases}$$

Panorâmica

Algoritmo

análise

Recorrência

Solução exacta

Solução aproximada
ordem de grandeza

Solução de recorrências

As técnicas que utilizamos para resolver tipos de recorrências (bem como outras que nãoo mencionamos) estão descritas por exemplo em
Lueker, G., *Some techniques for solving recurrences*, Computer Surveys, Vol. 12, No. 4, pag 419-436

Tipos abstractos de dados
Introdução às classes em C++

Tipos abstractos de dados

Ideia: Definir e usar as estruturas sem conhecer a sua implementação.

Exemplo: o "stack" (pilha)

Domínio $\mathcal{D}$ (por exemplo, inteiros) Conjunto $\mathcal{S}$ dos "stacks" com elementos do domínio $\mathcal{D}$.

Caracterizado pelas funções ("init" é uma constante = função de aridade 0)

push : $\mathcal{D} \times \mathcal{S} \to \mathcal{S}$ $(x, s) \to s'$	pop : $\mathcal{S} \to \mathcal{D} \times \mathcal{S}$ $s \to (x, s')$
empty : $\mathcal{S} \to \{t, f\}$ $s \to empty(s)$	init : $\to \mathcal{S}$ $\to$ "stack" vazio

Podemos caracterizar o " stack" por um conjunto de equações algébricas (onde se supõe a quantificação universal $\forall x \in \mathcal{D}$, $s \in \mathcal{S}$)

$$\begin{aligned} pop(push(x,s)) &= (x,s) \\ \neg empty(s) &\Rightarrow push(pop(s)_{\mathcal{D}}, pop(s)_{\mathcal{S}}) = s \end{aligned}$$

Uso das estruturas de informção

Podemos usar o “stack” (e prever os resultados!) sem conhecer a sua implementação.

```
/* O stack e' global... */
init();
push(2);
push(5);
print(pop());
...
```

Vantagens:

- Programação mais clara
- Modularidade
- Facilidade na divisão de trabalho
- “Debug”

Convém que o utilizador não tenha de facto acesso à implementação!

No C++

push, *pop* →		← *init*, *empty*
	Variáveis e funções públicas	
	Variáveis e funções privadas	
	Implementações	

Comecemos pelo nosso “main”

```
main() {
  stack a(3);
  int t;
  a.display();
  a.push(5);  cout << "Push 5\n";  a.display();
  a.push(3);  cout << "Push 3\n";  a.display();
  a.push(8);  cout << "Push 8\n";  a.display();
  a.push(9);  cout << "Push 9\n";
  t=a.pop();  cout << t << " (pop)\n";  a.display();
  t=a.pop();  cout << t << " (pop)\n";  a.display();
  t=a.pop();  cout << t << " (pop)\n";  a.display();
  t=a.pop();  cout << t << " (pop)\n";  a.display();
```

E os resultados. . .

```
$ a.out
   | | | |
Push 5
   |5| | |
Push 3
   |5|3| |
Push 8
   |5|3|8|
***   Error: Stack overflow
Push 9
8 (pop)
   |5|3| |
3 (pop)
   |5| | |
5 (pop)
   | | | |
***   Error: Stack underflow
0 (pop)
   | | | |
```


As declarações das variáveis e funções, públicas e privadas. . .

```
class stack {
public:
  stack(int);
  ~stack();
  void push(int);
  int pop();
  void display();

private:
  int size,      // of array
  index,         // current index
  *s;            // array
};
```

No C++

As implementações.
`stack::xxx(...)`: função "`xxx`" da classe "`stack`".

```
stack::stack(int n) {
  if(n<=0) error("Bad stack size");
  size=n;
  s = new int[n];
  index = 0;
}

stack::~stack() {
  delete[] s;
}

void stack::push(int x) {
  if(index==size)
    error("Stack overflow");
  else
    s[index++] = x;
}

int stack::pop() {
  if(index<=0)
    error("Stack underflow");
  return s[--index];
}
```


No C++

E os restos...das implementações...

```
void stack::display() {
  cout << "   |";
  int i;
  for(i=0;i<size;i++) {
    if(i<index)
      cout << s[i];
    else
      cout << " ";
    cout << "|";
    }
  cout << '\n';
}
```

...e, finalmente os restos dos restos...

```
#include <iostream.h>
//------------------------------------//
void error(char *m) {
  cout << "***   Error: " << m << '\n';
}
```

Algumas notas sobre as classes

– Em "`main`" a declaração "`stack a(3)`" cria um novo <u>objecto</u> da <u>classe</u> "`stack`"

– Funções públicas (que podem ser usadas do exterior) da classe "`stack`": "`stack`", "`~stack`", "`push`", "`pop`" e "`display`".

– A função "`stack`" (tem o nome da classe) é chamada *constructora* e é executada automaticamente quando se cria um objecto (como em "`stack a(3)`").

– A função "`~stack`" (tem o nome da classe precedido de "`~`") é chamada destructora e é executada quando o objecto deixa de existir (por exemplo, antes do retorno da função em que está definido). Esta função não tem parâmetros.

– Outros aspectos do C++ são

  – O uso de "`new`" e "`delete`" em vez de (por exemplo) respectivamente "`malloc`" e "`free`".

  – O modo como pode ser efectuado o "input/output", através de uma classe "`cout`". As funções usualmente utilizadas em C ("`printf`", "`scanf`",...) podem, evidentemente, continuar a sê-lo.

<u>Nota</u>: Em disciplinas anteriores foram estudadas as listas lineares:

- Pilhas ("stacks")
- Filas ("queues")
- "Deques"

bem como a sua implementação em memória *indexada* ("arrays") e *ligada* (com apontadores).
Esta matéria é suposta conhecida

A conversão destes algoritmos para a linguagem C++ (com o uso de classes) é deixada como exercício.

Exercício 24 *Defina a classe "fila" com uma implementação com apontadores (listas ligadas).* ◇

Uma estrutura de informação: Fila de prioridade

Uma implementação da fila de prioridade: O "heap"

Um método eficiente de ordenação: O "heapsort"

A Fila de Prioridade

Operações básicas:

- `void insere(<tipo> x)`: O valor `x` é colocado na fila de prioridade.

- `<tipo> extrai_min(void)`: o menor valor é extraído da fila de prioridade e o resultado é esse valor.

Podem ser armazenados ou não valores repetidos. Supomos que sim.

Aplicações da Fila de Prioridade

- *Simulação*: em linguagens de simulação (por exemplo, de chegada de pessoas a uma fila, lavagem de carros, etc) como o Simula. Os valores guardados são *acontecimentos* e são comparados pelo seu *tempo* de (início de) execução

  (tempo, outras características...)

  Em cada passo, o próximo (tempo mínimo) acontecimento é retirado da fila de prioridade (`extrai_min`) e executado. A sua execução pode levar à introdução de novos acontecimentos futuros (`insere`).

- *Execução de processos por um CPU.*

- *Ordenação*. Em linguagem "informal":

```
Fila de prioridade F;
for(i=0;i<n;i++)
  F.insere(x[i]);
for(i=0;i<n;i++)
  print(F.ext_min());
```


Implementações "ingénuas"

Vector não ordenado:

- `insere`: $O(1)$
- `ext_min`: $O(n)$

Complexidade total de n inserções ou eliminações (exemplo: a ordenação descrita): $O(n)$.

Vector ordenado:

- `insere`: $O(n)$
- `ext_min`: $O(1)$

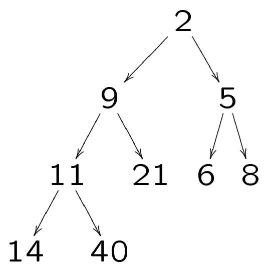
Complexidade total de n inserções ou eliminações (exemplo: a ordenação descrita): $O(n)$.

Implementação com o "heap"

<u>Heap</u>:

- Árvore binária *completa*: por níveis, da esquerda para a direita:
- Os filhos de cada nó (pode haver 0, 1 ou 2) são maiores ou igual que ele.

Um "heap" com o conjunto $\{2, 5, 6, 8, 9, 11, 14, 21, 40\}$:



Exercício 25 *Quantos "heaps" com estes mesmos elementos (2, 5, 6, 8, 9, 11, 14, 21, 40)?* $\diamond$

Uma ideia brilhante: "heap"

<u>Boa ideia</u>: um "heap" pode ser representado simplesmente num "array", preenchendo-o seguindo as linhas da árvore!

Índice	1	2	3	4	5	6	7	8	9	10	11
Valor	2	9	5	11	21	6	8	14	40		

$n = 9$

Acesso à mãe e aos filhos do nó i:

- Mãe do nó i: índice $\lfloor i/2 \rfloor$
- Filhos do nó i: índices $2i$ e $2i + 1$

Descrição resumida de $insere(x)$

1. x é colocado no fim do "array", índice $n+1$
   $n \leftarrow n+1$.

2. x *flutua*: enquanto x não estiver no topo e for menor que o pai, troca com ele.

Descrição resumida de $extrai\_min()$

1. O valor no topo da árvore é guardado em r para ser "retornado" no fim.

2. O valor t que está no fim do array é colocado no topo da árvore e o "array" encurta de uma unidade ($n \leftarrow n-1$).

3. t *mergulha*: Enquanto t for maior que o menor dos filhos (e tiver pelo menos um filho), troca t com o menor deles.

4. Retorna r, primitivo valor do topo (mínimo).

O "heapsort: `main` e resultados"

```
//------------------------------------------------------
// Heapsort (porque a fp é implementada como "heap")
//------------------------------------------------------

int main() {
  int i;
  int const n=100;
  fp a(n);
  for(i=1;i<n;i++)
    a.insert(rand()%1000);
  for(i=1;i<n;i++)
    printf("%4d",a.extract());
}
/*-------------------Resultados ------------------
  6  21  31  35  43  63  67  71  96 103
109 114 115 124 129 132 141 170 183 191
197 204 207 220 225 228 235 238 246 249
257 260 272 288 294 303 307 313 335 349
361 363 364 373 388 392 401 402 407 413
414 419 441 448 457 466 477 478 496 505
514 516 528 530 541 544 565 583 585 588
596 614 681 710 725 728 749 768 795 825
826 832 845 857 867 869 892 894 898 901
908 909 925 925 935 938 942 978 984
-------------------------------------------------*/
```

O “heapsort: declarações da classe `fp`”

```
//--------------------------------------------------
class fp {         // fila de prioridade, privadas:
public:
  fp(int);         // Publicas
  void insert(int);// Insere um valor
  int extract();   // Extrai o valor mínimo
private:
  int size,        // tamanho máximo
      index,       // pos. do último, (começa em 1)
      *a;          // o "array"
};
//--------------------------------------------------
```


O “heapsort: construtor e `insert`”

```
fp::fp(int n) {
  if(n<=0){
    printf("Illegal size\n");
    exit(1);
  }
  size=n;
  a = new int[n+1];
  index = 0;
}

void fp::insert(int x) {
  int k,t,up;
  if(index>=size){
    printf("Heap overflow\n");
    exit(1);
  }
  a[++index]=x;      // Coloca no fim;
  k=index;           // E vai subir...
  while(k>=2){       // Enquanto não for raíz...
    up=k/2;
*   if(a[up]<=a[k]) break;
    t=a[k];
    a[k]=a[up];
    a[up]=t;
    k=up;
  }
}
```

O “heapsort: `extract_min`

```
int fp::extract(){       // Extrai o valor mínimo
  int ret,f,f1,f2,k,t;
  if(index<=0){
    // (recordando: a pos. 0 não é usada!)
    printf("Heap underflow\n");
    exit(1);
  }
  ret=a[1];              // Vai retornar "ret"
  a[1]=a[index--];       // O último vai para o topo
  k=1;
  while(1){
    f1=2*k;
    f2=2*k+1;
    if(f1>index) break;  // não tem filhos...
    if(f2>index) f=f1;   // so' tem 1...
*   else if(a[f1]<a[f2]) f=f1;
    else f=f2;           // a[f] e' o menor dos 2
*   if(a[k]>a[f]){       //
      t=a[k];
      a[k]=a[f];
      a[f]=t;
      k=f;
    }
    else break;
  }
  return ret;
}
```


Análise da eficiência do “heapsort”

Vamos majorar o número de comparações $c(n)$ em que intervêm elementos a ordenar.

1. O número de elementos no “heap” é $\leq n$.

2. Há n `insert(x)`’s e n `extract_min()`’s

3. A altura h de uma árvore completa com m elementos é (Exercício!) $h = \lfloor \log m \rfloor \leq \log m$. Temos pois que é sempre $h \leq \log n$.

4. Em cada `insert(x)` fazem-se no máximo h comparações (do tipo indicado).

5. Em cada `extract_min()` fazem-se no máximo $2h$ comparações (do tipo indicado).

Temos

$$c(n) \leq nh + 2nh \leq 3n \log n$$

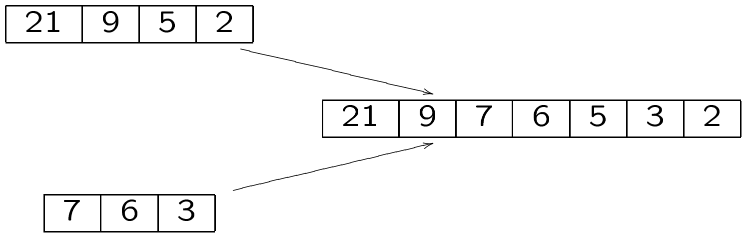
Este algoritmo é $O(n \log n)$!

O número de comparações $c(n)$ é geralmente utilizado como medida da eficiência dos algoritmos de ordenação. A ordem de grandeza da eficiência é, neste caso, $O(c(n))$ (verifique!).

Outro método eficiente de ordenação: O “mergesort”

“Merge” (fusão?): algoritmo que, a partir de 2 vectores *ordenados* com m e n elementos produz um vector *ordenado* que contém os $m+n$ elementos dos vectores iniciais.

<u>Exemplo</u>.



Algoritmo de “merge”

Algoritmo de “merge”, ideia geral:
Compara-se o mais pequeno elemento de cada vector. O menor deles impresso avançando-se de uma unidade no vector correspondente. Quando um dos vectores se tiver esgotado, o resto do outro é impresso (por ordem).

Exercício 26 *Implemente o algoritmo de “merge” numa linguagem imperativa (por exemplo, em C).* ◇

Exercício 27 *Mostre que o número c de comparações efectuadas no algoritmo de “merge” satisfaz*

$$\min(m,n) \leq c \leq m+n-1$$

◇

Algoritmo de “mergesort”

Algoritmo de “mergesort”, supondo-se que o número n de elementos a ordenar é uma potência de 2.

```
nparts=n;
while(nparts>=2){
  for(i=1;i<=nparts/2;i+=2)
    merge parts i and i+1
  // nparts is now half
}
```

Temos pois
$n/2$ “merges” de pares de vectores de tamanho 1
$n/4$ “merges” de pares de vectores de tamanho 2
...
1 “merge” de 2 vectores de tamanho $n/2$.

Exercício 28 *Usando a análise anterior, calcule um majorante do número de comparações efectadas.* $\diamond$

Algoritmo de “mergesort”, continuação

Outro método é exprimir um majorante do número total de comparações efectuadas através da recorrência:

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

Já resolvemos esta recorrência! A solução, como vimos, é

$$c(n) \leq (\log n - 1)n + 1 \in O(n \log n)$$

E se n não for uma potência de 2? Com pouco esforço podemos encontrar um majorante. Seja $2^p < n \leq 2^{p+1}$. Se substituirmos $\log n$ por $1 + \log n$ e n por $2n$ temos para o caso geral

$$c(n) \leq 2n \log n + 1 \in O(n \log n)$$

Panorâmica: O que já vimos sobre ordenação

Eficiência medida pelo número de comparações envolvendo elementos a ordenar.

1. Um algoritmo simples e relativamente ineficiente: Método de selecção, $O(n^2)$, $c(n) \leq n(n-1)/2$ Com esta eficiência existem vários como o método da bolha.

2. Dois algoritmos eficientes, $O(n \log n)$, supondo que n é potência de 2:
   (a) "heapsort", $c(n) \leq 3n \log n$
   (b) "mergesort", $c(n) \leq (\log n - 1)n + 1$

Exercício 29 *Tabele os majorantes para os 3 métodos de ordenação, para* $n = 1, 2, 4, 8, 16, 32$. $\diamond$

Ordenação: número mínimo de comparações

Em geral é muito difícil obter minorantes próximos dos resultados obtidos pelos melhores algoritmos. Mas no caso da ordenação conseguimos isso!

Supondo:
(i) Eficiência medida pelo número de comparações envolvendo elementos a ordenar.
(ii) Elementos do vector todos distintos, o resultado de uma comparação ou é "$<$" ou é "$>$".

Princípio Se a única informação que um algoritmo obtém de um vector é obtida por n comparações, o algoritmo só se pode comportar de 2^n modos distintos.

Corolário (Análise no pior caso) Um algoritmo que ordena um vector de n elementos distintos tem de efectuar no mínimo um número de comparações c tal que $2^c \geq n!$.

Ordenação: número mínimo de comparações

Vejamos se o "mergesort" está longe ou perto deste minorante.

n	Minorante $\log(\lceil n! \rceil)$	Mergesort $n(\log n - 1) + 1$
2	1	1
4	5	5
8	16	17
16	44	49

Conclusão: o mergesort (no pior caso) parece ser muito próximo do óptimo!

Se aproximarmos $n!$ usando a fórmula de Stirling (cujo erro relativo tende para 0 quando $n \to \infty$)

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

obtemos

$$n \geq \lceil \log(\sqrt{2\pi n}(n/e)^n)$$

Exercício 30 *Mostre que o minorante para o número de comparações obtido pela expressão anterior é $O(n \log n)$. Assim, em termos de ordens de grandeza, o minorante e os melhores algoritmos são idênticos!* $\diamond$

Nota sobre a aproximação de Stirling

Aproximação de Stirling:

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

Uma aproximação mais exacta corresponde a multiplicar a aproximação de Stirling por $(1 + 1/(12n))$

n	n!	Stirling	Stirling*(1+1/(12n))
1	1	0.92	1.00
2	2	1.92	2.00
3	6	5.84	6.00
4	24	23.51	24.00
5	120	118.02	119.99
6	720	710.08	719.94
7	5040	4980.40	5039.69
8	40320	39902.42	40318.07
9	362880	359537.06	362866.11
10	3628800	3598697.54	3628686.68

Exercício 31 *Escreva um programa em C (ou C++) para imprimir a tabela anterior (calculando os valores!).*
◇

Algoritmos : alguns esquemas gerais

- "Dividir para Conquistar"
- Tabulação
- Programação Dinâmica