Capítulo 1

Preliminares: fundamentos da análise de algoritmos

Este capítulo é uma exposição sumária de alguns preliminares relativos à análise de algoritmos. Depois de referirmos os modelos de computação normalmente utilizados e de mencionar os principais recursos utilizados durante a execução de um programa (tempo e espaço), estudaremos 2 aspectos fundamentais para a análise de algoritmos: ordens de grandeza das funções e solução de recorrências. Os conceitos e os resultados apresentados neste capítulo são fundamentais para a análise da eficiência dos algoritmos apresentados neste curso.

1.1 Eficiência dos algoritmos

1.1.1 Eficiência dos algoritmos: duas análises

1. Pormenorizada: mais exacta e directa mas em geral menos útil
   - Expressa em segundos.
   - Resultado da avaliação da eficiência (por exemplo: tempo gasto): único e dependente da velocidade e características do processador.
2. Através de ordens de grandeza: ignora as constantes multiplicativas e é uma análise assintótica ←
   - Expressa em *ordens de grandeza*

– Resultado da avaliação da eficiência: paramétrico (uma função do comprimento dos dados) e independente da velocidade e características do processador.

Nesta disciplina usaremos a análise 2, "através de ordens de grandeza"!

1.1.2 Recursos e modelos de computação

Em que termos é medida a "eficiência de um algoritmo"?

Resposta: pela quantidade de recursos gastos durante a sua execução como função do *comprimento dos dados*.

Recursos? O que são?

Resposta: tempo (o mais usual) e espaço (memória)

Modelos de computação mais usados:

1. Máquinas de Turing (MT)
2. Computadores de registos ("Random Access Machines")

Tempo e espaço, o que são?

Nas MT

1. Tempo: número de transições da MT durante a computação
2. Espaço: número de células visitadas pela cabeça da MT durante a computação

1.1.3 Pior caso e caso médio

Eficiência do algoritmo $\boxed{\text{A}}$ como função de quê?

Resposta: Do comprimento dos dados.

$$x \longrightarrow \boxed{\text{A}} \longrightarrow y$$

Seja $n = |x|$

Tempo de execução: depende dos dados, $t(x)$

Simplifica-se: como função do comprimento dos dados

$$t = f(n) \text{ onde } n = |x|$$

Mas... isto está errado, o tempo de execução é função de x e não de $n = |x|$!

Temos que escolher um tempo de entre todos os tempos $t(x)$

com $n = |x|$

Hipóteses mais frequentes:

1. Pior caso: $t(n) = \max\{t(x) : |x| = n\}$

2. Caso médio: $t(n) = E\{t(x) : |x| = n\}$ onde se assume uma determinada distribuição probabilística dos dados x de comprimento n; x e $t(x)$ são variáveis aleatórias. Significado de E: "valor médio".

1.1.4 Recorrências, pior caso e caso médio, um exemplo

Seja o programa

```
//-- Dado: vector v[] com n elementos, v[0,1,...,n-1]
1    int i, m
2    m=0;
3    for i=1 to n-1
4      if v[i] > v[m] then
5         m=i
6    print m
```

É fácil ver que o tempo de execução é majorado por

$$\alpha n + \beta$$

onde α e β são constantes, ver por exemplo Knuth, "The Art of Computer Programming", vol. 1.

Concentremo-nos na seguinte questão:

Quantas vezes é executada a atribuição da linha `5`?

Seja a esse número de vezes.

- Mínimo: $a = 0$ vezes
- Máximo: $a = n - 1$ vezes
- Médio: $E(a) =$???

Hipóteses sobre a distribuição probabilística dos dados

1. Todos os elementos de `v[]` são diferentes. Note que os valores do vector são irrelevantes, só é importante a sua "ordem", por exemplo, o comportamento para `v=[4,1,2,5,3]` e para `v=[41,2,5,55,30]` é idêntico.

2. Qualquer das $n!$ permutações é igualmente provável.

Calculemos alguns valores de prob $a = i$

- prob $a = 0$, probabilidade de ser $a = 0$ é $1/n = (n-1)!/n!$ = probabilidade de `v[0]` ser o maior de todos.
- prob $a = n - 1 =$ é $1/n!$ pois só numa das $n!$ permutações isso acontece.

Teremos que calcular a média

$$E(a) = 0 \times \text{prob } a = 0 + 1 \times \text{prob } a = 1 + \cdots + (n-1) \times \text{prob } a = n - 1$$

Vamos calcular $E(a)$ como uma função de n, seja $E(n)$. Temos uma recorrência:

1. $E(1) = 0$
2. Imaginemos que o vector tem $n + 1$ elementos e calculemos como é que $E(n + 1)$ depende de $E(n)$. Temos

$$E(n+1) = E(n) + 1 \times \text{prob o último elemento ser o maior}$$

Ou seja

$$\begin{cases} E(1) = 0 \\ E(n+1) = E(n) + 1/n \end{cases}$$

A solução desta recorrência é fácil

$$E(n) = 0 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} = \left(\sum_{i=1}^{n-1} \frac{1}{i} \right) - 1$$

Por exemplo, para $n = 1000$ temos

- Mínimo $= 0$
- Máximo $= 999$
- Média $E(n) \approx 6.49$

Mas isto não é uma "forma fechada"!

Podemos determinar uma forma fechada?

Ou... talvez uma boa aproximação?

Exercício 1 *Confirme o resultado anterior para um vector de 3 elementos distintos, estudando as* $6 = 3!$ *situações possíveis:*

`v[0]<v[1]<v[2],...v[2]<v[1]<v[0]`

(Fim do exemplo)

Investigar o caso geral! $E(n) = \ldots$?

1.2 Ordens de grandeza

Como "ignorar constantes e comportamento inicial"?

Resposta: **usando ordens de grandeza**.

1.2.1 Majoração, minoração e ordem exacta

Ordens de grandeza (ideia)

1. Majoração de $f(n)$ por $g(n)$:
   $f(n)$ é de ordem $O(g(n))$ – ou $f(n) \in O(g(n))$.
   Para valores suficientemente grandes de n é $f(n) \leq kg(n)$ onde k é uma constante real positiva.

2. Minoração de $f(n)$ por $g(n)$:
   $f(n)$ é de ordem $\Omega(g(n))$ – ou $f(n) \in \Omega(g(n))$.
   Para valores suficientemente grandes de n é $f(n) \geq kg(n)$ onde k é uma constante real positiva.

3. Ordem de grandeza exacta:
   $f(n)$ é de ordem $\Theta(g(n))$ – ou $f(n) \in \Theta(g(n))$.
   Para valores suficientemente grandes de n é $k_1 g(n) \leq f(n) \leq k_2 g(n)$ onde k_1 e k_2 são constantes reais positivas.

As fefinições formais das ordens de grandeza $O()$, $\Omega()$ e Θ são as seguintes.

Definições (como conjuntos):

Sejam f e g funções de $\mathbb{N}$ em $\mathbb{R}$. Seja $\mathbb{R}^+$ o conjunto dos reais positivos.

– Majoração: $g(n)$ é $O(f(n))$, f é majorante ("upper bound") de g

$$O(f(n)) = \{g(n) \ : \ \exists n_0 \in \mathbb{N}, \exists k \in \mathbb{R}^+, \ \forall n \geq n_0 : \ \ g(n) \leq kf(n)\}$$

– Minoração: $g(n)$ é $\Omega(f(n))$, f é minorante ("lower bound") de g

$$\Omega(f(n)) = \{g(n) \ : \ \exists n_0 \in \mathbb{N}, \exists k \in \mathbb{R}^+, \ \forall n \geq n_0 : \ \ g(n) \geq kf(n)\}$$

– $g(n)$ é $\Theta(f(n))$, f é da ordem de grandeza exacta de g

$$\begin{aligned}\Theta(f(n)) \quad = \quad & \{g(n) \ : \ \exists n_0 \in \mathbb{N}, \exists k_1, k_2 \in \mathbb{R}^+, \\ & \forall n \geq n_0 : \ \ k_1 f(n) \leq g(n) \leq k_2 f(n)\}\end{aligned}$$

Exercício 2 *Compare estas definições com as da página anterior; dê um significado a "suficientemente grande" por forma a que as definições coincidam.*

As 3 ordens de grandeza – notas

– Note-se que $O(f(n))$, $\Omega(f(n))$ e $\Theta(f(n))$ são conjuntos. Quando dizemos $n^2 + 3$ é de ordem $O(n^3)$ queremos dizer $n^2 + 3 \in O(n^3)$.

– Para que serve o n_0, porquê apenas para valores de n suficientemente grandes?

  – O domínio de $f(n)$ pode não ser $\mathbb{N}$. Por exemplo, diz-se que $\log n^2$ é de ordem $O(n^2)$ embora a função não esteja definida para $n = 0$.

  – $f(n)$ pode tomar valores nulos ou negativos...

– A noção de ordem pode ser estendida para funções $\mathbb{R}_0^+ \to \mathbb{R}^+$.

– Gráficos ilustrativos (no quadro!)

Exercício 3 *Verdadeiro ou falso? Justifique*

1. $\log n \in O(n)$
2. $n \in O(\log n)$
3. $2n^2$ *é de ordem* $O(n^2)$

4. $2n^2$ *é de ordem* $O(n^3)$
5. $\Omega(n) = \Omega(3n)$ *(igualdade de conjuntos)*
6. *4 é* $O(1)$
7. $O(n^2) \subseteq \Omega(n^2)$

Exercício 4 *Mostre que a seguinte função*

$$f(n) = \begin{cases} 2n & \text{se } n \text{ é par} \\ 2 & \text{se } n \text{ é ímpar} \end{cases}$$

é de ordem $O(n)$ *mas não de ordem* $\Theta(n)$.

Exercício 5

Porque é que usualmente não se diz que uma determinada função é de ordem $O(2n)$ *ou* $\Omega(3n)$ *ou* $\Theta(4n)$*? O coeficiente que se usa é sempre 1; assim, diz-se ordens* $O(n)$, $\Omega(n)$ *e* $\Theta(n)$ *respectivamente.*

Exercício 6

Considere a seguinte relação binária entre funções totais de $\mathbb{N}$ *em* $\mathbb{R}^+$*:* fRg *sse* $f(n)$ *é de ordem* $O(g(n))$*. Averigue se a relação é simétrica. Repita o exercício para a ordem de grandeza* Θ.

Exercício 7

Diz-se que $f(n)$ *é de ordem* $o(g(n))$*, quando*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Por exemplo, $1/(n^2 \log n)$ *é* $o(1/n^2)$*. Consegue encontrar alguma relação matemática entre esta definição e as "nossas" ordens de grandeza?*

1.2.2 Tempo de execução em algoritmos de ordenação e pesquisa

Para exprimir o tempo de execução usa-se muitas vezes

- o número de comparações envolvendo elementos do vector em questão

Exemplo

```
    //-- ordena v[0..n-1] pelo método da selecção do mínimo
    int i,j,m,t;
1   for i=0 to n-2
2   | m=i;
3   | for j=i+1 to n-1
4   |   if v[j]<v[m]
5   |    m=j
6   | t=v[i]; v[i]=v[j]; v[j]=m;
    // v está ordenado!
```

Exercício 8 *Quantas comparações* `v[j]<v[m]` *são efectuadas (linha* `4`*)? Exprima a sua resposta como uma função de* `n`*.*

Exemplo

```
    //-- "merge" de a[0..m-1] e b[0..n-1] (já ordenados)
    //   em v[]
    //-- Ex: a=[1,4,5,8], b=[2,3,4] --> v=[1,2,3,4,4,5,8]
1   int i=0, j=0, k=0;
2   while i<m && j<n
3     if a[i]<b[j]
4       v[k]=a[i]; i=i+1; k=k+1;
5     else
6       v[k]=b[j]; j=j+1; k=k+1;
7   while i<m: v[k]=a[i]; i=i+1; k=k+1; // só um destes ciclos
8   while j<n: v[k]=b[j]; j=j+1; k=k+1; // é executado. Porquê?
```

(Note que um dos ciclos finais (linhas `7` e `8` é "executado" 0 vezes)

Exercício 9 *Determine o número de comparações* `a[i]<b[j]` *efectuadas (na linha* `3`*)*

1. *Valor mínimo (pode supor* `m<n`*).*
2. *Valor máximo.* RESPOSTA: $m + n - 1$.

Exercício 10 *Dê um exemplo de um algoritmo de ordenação em que o número de comparações $c(n)$ envolvendo elementos do vector não é da mesma ordem que o tempo de execução do algoritmo. Assim, neste caso, $c(n)$ não "exprime" correctamente o tempo de execução do algoritmo.*

1.3 Solução de recorrências

1.3.1 Exemplos de funções definidas através de recorrências – definições indutivas

Factorial

$$n! = \begin{cases} 1 & (\text{se } n = 0) \\ n(n-1)! & (\text{se } n \geq 1) \end{cases} \qquad \text{ou...} \qquad \begin{cases} f(0) = 1 & (\text{se } n = 0) \\ f(n) = nf(n-1) & (\text{se } n \geq 1) \end{cases}$$

Fibonacci

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad (\text{se } n \geq 2) \end{cases}$$

Que será?

$$\begin{cases} f(0) = 0 \\ f(n+1) = f(n) + 3n^2 + 3n + 1 \quad (\text{se } n \geq 0) \end{cases}$$

Que será?

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

Nota: só está definido quando n é potência de 2.

1.3.2 O que é uma recorrência

O que é uma recorrência?

Resposta: um método de definir sucessões.

Uma sucessão $f : \mathbb{N} \to \mathbb{R}$ pode ser definida por um conjunto finito de equações dos seguintes tipos:

- Equações fronteira: $f(k) = c$ onde k é uma constante inteira não negativa e c é uma constante real.

  Por exemplo: $f(2) = 1$.

- Equações gerais:

  $$f(n) = \exp(\cdots) \text{ para } n\ldots$$

  onde n é uma variável inteira e $\exp(\cdots)$ é uma expressão que pode envolver valores de $f(i)$ para $i < n$.

  Por exemplo: $f(n) = f(n-1) + f(n-2)$ para $n \geq 2$.

As equações gerais e fronteira devem definir univocamente todos os valores de $f(i)$. Normalmente existe apenas uma equação geral.

Há uma relação muito próxima entre

recorrências | definições indutivas

demonstração por indução finita

Há uma relação próxima entre

recorrências $\leftrightarrow$ eq. diferenciais

Por vezes a solução é directa...

Por vezes a solução é directa...

Um exemplo:

$$\begin{cases} f(0) = 1 \\ f(n+1) = f(n) + n \end{cases}$$

Nota. "$f(n+1) = f(n) + n$ para $n \geq 0$" é equivalente a "$f(n) = (n-1) + f(n-1)$ para $n \geq 1$" (mudança de variável).

Temos

$$\begin{aligned} f(n) &= (n-1) + f(n-1) \\ &= (n-1) + (n-2) + f(n-2) \\ &= \cdots \\ &= ((n-1) + (n-2) + \cdots + 1 + 0) + 1 \\ &= n(n-1)/2 + 1 \end{aligned}$$

Portanto a solução é

$$f(n) = \frac{n^2 - n + 2}{2}$$

Neste caso a solução directa foi possível porque sabemos somar progressões aritméticas.

Nota. O aluno deve conhecer as fórmulas das somas de progressões aritméticas e geométricas (ou saber deduzi-las...)

Um exercício

Exercício 11 *Determine directamente a solução da seguinte recorrência*

$$\begin{cases} f_1 = 2 \\ f_{n+1} = 3f_n \end{cases}$$

Observações

- *Domínio.* Nem sempre uma recorrência define uma sucessão para todos os valores de n. Por exemplo, em algoritmos de ordenação pode simplificar-se a análise se se supuser que o número de elementos é uma potência de 2.
- *Existência.* para muitas recorrências não é conhecida – e possivelmente não existe – uma forma fechada para a sua solução.
- *Majoração.* Por vezes a solução de uma recorrência é muito difícil ou impossível, mas pode por vezes a partir dela definir outra mais simples cuja solução majora a função caracterizada pela recorrência; isso pode ser suficiente para, por exemplo, conhecer uma "ordem de

grandeza" da solução.

Isso pode acontecer, por exemplo, quando a função definida pela recorrência é o tempo de execução de um algoritmo.

1.3.3 Método Tabelar → suspeitar → demonstrar

O "método"

1. Tabelar: usando a recorrência tabelamos os primeiros valores da função; para ajudar no passo seguinte podem tabelar-se outras funções como n^2, 2^n, $\log n$, etc.

2. Suspeitar: eventualmente a tabela construída no passo anterior pode levar-nos a suspeitar que a solução é uma determinada função de n, seja $f(n)$.

3. Demonstrar: provamos – usualmente usando o método da indução finita – que $f(n)$ é de facto (se for!) a solução da recorrência.

Claro que se não conseguirmos avançar no passo 3. pode a suspeita estar errada e há que retroceder ao passo 2.

Um exemplo simples

$$\begin{cases} a_0 = 0 \\ a_{n+1} = a_n + 2n \end{cases}$$

Tabelar

n	a_n
0	0
1	0
2	2
3	6
4	12
5	20

Qual parece ser a solução?

Suspeitar

... coloquemos a coluna n^2

n	n^2	a_n
0	0	0
1	1	0
2	4	2
3	9	6
4	16	12
5	25	20

Agora, a suspeita é fácil!

$$f(n) = n^2 - n? \qquad \text{(suspeita)}$$

Demonstrar

O quê?

Teorema 1 *A solução da recorrência é* $n^2 - n$.

Dem. Por indução em n.

I. Caso $n = 0$: $0^2 - 0 = 0$. E da recorrência é $a_0 = 0$. $\surd$

II. Demonstrar que $\boxed{a_n = n^2 - n \;\Rightarrow a_{n+1} = (n+1)^2 - (n+1)}$.

Temos

$$\begin{array}{rcll} a_{n+1} & = & a_n + 2n & \text{(da recorrência)} \\ & = & (n^2 - n) + 2n & \text{(hipótese indutiva)} \\ & = & (n+1)^2 - (n+1) & \text{(contas simples!)} \end{array}$$

. . . e está demonstrada a implicação. □

Exercício 12 *O "mergesort" é um método de ordenação muito eficiente, mesmo no pior caso. Descrição, supondo que n é uma potência de 2:*

`mergesort(v[0...n-1],n):`

1. *Se* $n = 1$*: nada se faz*
2. *Se* $n \geq 2$*:*
   - *(a)* $\texttt{mergesort}(\texttt{v}[0 \cdots \texttt{n}/2 - 1], \texttt{n}/2)$
   - *(b)* $\texttt{mergesort}(\texttt{v}[\texttt{n}/2 \cdots \texttt{n} - 1], \texttt{n}/2)$
   - *(c)* $\texttt{merge}(\texttt{v}[0 \cdots \texttt{n}/2 - 1], \texttt{v}[\texttt{n}/2 \cdots \texttt{n} - 1]) \longrightarrow \texttt{v}[0 \cdots \texttt{n} - 1]$

Já definimos o que é o `merge`*, ver página 20.*

1. *Ilustre a execução do mergesort para o vector*

$$v[\,] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 9 & 7 & 8 & 5 & 1 & 3 & 6 & 2 \\ \hline \end{array}$$

2. *Da definição resulta a seguinte recorrência para um majorante do número de comparações efectuadas (são todas efectuadas nas chamadas de* `merge`*); use o máximo indicado na página 20 com* $m \to n/2$, $n \to n/2$*:*

Resposta: _____

3. *Resolva a recorrência pelo método "tabelar / suspeitar / demonstrar".*

1.3.4 Método das diferenças finitas constantes

Definição 1 *Seja* a_n, $n \in \mathbb{N}$, *uma sucessão. A diferença finita (ou simplesmente diferença) de* $1^{\underline{a}}$ *ordem de* a_n *é a sucessão*

$$D_n^1 = a_{n+1} - a_n \qquad \textit{(para } n \geq 0\textit{)}$$

Para $i \geq 2$ *a diferença de ordem* i *de* a_n *é a sucessão*

$$D_n^i = D_{n+1}^{i-1} - D_n^{i-1}$$

Exemplo. Para a sucessão a_n definida pela recorrência

$$\begin{cases} a_0 = 1 \\ a_n = a_{n-1} + n^2 \quad \text{para } n \geq 1 \end{cases}$$

as diferenças de ordem 1 e 2 são

n	a_n	D_n^1	D_n^2
0	1	1	3
1	2	4	5
2	6	9	$\cdots$
3	15	$\cdots$	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

Teorema 2 *Se as diferenças finitas de ordem* m *da sucessão* a_n *são uma constante não nula, a solução da recorrência é um polinómio de grau* m*, isto é*

$$a_n = c_m n^m + c_{m-1} n^{m-1} + \cdots + c_1 n + c_0$$

com $c_m \neq 0$.

Nota. A *diferença* de ordem m é como que a imagem discreta, isto é, em termos das funções de $\mathbb{N}$ em $\mathbb{R}$, da *derivada* de ordem m. No domínio "contínuo" o Teorema anterior tem a seguinte "versão": se a derivada de ordem m de uma função $f(x)$ é uma constante não nula, então $f(x)$ é um polinómio de grau m em x.

Método 1 da solução da recorrência

1. Provar que, para um determinado inteiro m, as diferenças finitas de ordem m são uma constante não nula.
2. Usar o método dos coeficientes indeterminados ($m+1$ valores de a_n) para determinar os coeficientes do polinómio.

Método 2 da solução da recorrência

1. Tabelar os primeiros valores de: a_n, D_n^1,..., D_n^m.
2. Verificar que (se) D_n^m aparenta ser uma constante.
3. Usar o método dos coeficientes indeterminados ($m+1$ valores de a_n) para determinar os coeficientes do polinómio.
4. Provar que o polinómio obtido é solução da recorrência.

O seguinte resultado permite em alguns casos a aplicação directa do método das diferenças finitas constantes.

Teorema 3 *A solução de uma recorrência da forma $t_0 = a$, $t_{n+1} = t_n + p(n)$ onde a é uma constante e $p(n)$ um polinómio de grau d é um polinómio de grau $d+1$.*

Exercício 13 *(i) Continuar a tabela do exemplo anterior (pág. 26), mostrando que D_n^3 aparenta ser uma constante positiva. Determinar a solução da recorrência (método 2). (ii) Resolver directamente a recorrência usando o Teorema 3.*

Exercício 14 *Determinar uma fórmula fechada para a soma dos primeiros n quadrados,*

$$S_n = \sum_{i=1}^{n} i^2$$

Use o exercício anterior.

1.3.5 Método da mudança de variável

Vamos usar um exemplo.

Seja a recorrência

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

Note que $a(n)$ só fica definido quando n é potência de 2, $n = 2^p$, $p \geq 0$.

É natural efectuar a mudança de variável $n = 2^p$ (ou $p = \log n$). Isto é vamos representar $a(b)$ por uma outra recorrência $b(p)$ sendo

$$a(n) \equiv b(p)$$

Fica

$$\begin{cases} b(0) = 1 \\ a(p+1) = b(p) + 1 \end{cases}$$

cuja solução é muito simples, $b(p) = p + 1$, ou seja, em termos de n:

$$a(n) = \log n + 1$$

1.3.6 Método da equação característica homogénea

Estudamos agora a solução de recorrências em f_n com uma única equação geral da forma

$$a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = 0$$

onde k é uma constante e $a_0 \neq 0$.

Exercício 15 *Mostre que as equação geral da sequência de Fibonacci é da forma indicada.*

Se experimentarmos soluções da forma $f_n = r^n$ vemos que r deve satisfazer a equação (dita característica)

$$a_0 r^k + a_1 r^{k-1} + \cdots + a_r = 0$$

Teorema 4 *Se $f_n = f_1(n)$ e $f_n = f_2(n)$ são soluções da equação $a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = 0$, então, sendo α e β quaisquer constantes, $\alpha f_1(n) + \beta f_2(n)$ também é solução dessa equação.*

Exercício 16 *Demonstre este resultado.*

Teorema 5 *Se as raízes da equação característica são todas distintas (reais ou complexas), sejam $r_1, \ldots, r_k$, a solução da equação geral é exactamente constituída pelas sucessões da forma*

$$f_n = \alpha_1 {r_1}^n + \alpha_2 {r_2}^n + \cdots + \alpha_k {r_k}^n$$

onde $\alpha_1, \alpha_2, \ldots,$ e α_k são constantes arbitrárias.

Nota. Entre as soluções possíveis está $f(n) = 0$ (solução identicamente nula) que se obtém com $\alpha_1 = \alpha_2 = \ldots = \alpha_k = 0$.

1.3.7 Equação característica homogénea – raízes distintas

Método:

1. Determine a equação característica e as suas raízes, $r_1, \ldots, r_k$.
2. Determine os coeficientes α_i, para $i = 1, 2, \cdots, k$, através de um sistema de k equações lineares da forma

   $$\alpha_1 {r_1}^n + \alpha_2 {r_2}^n + \cdots + \alpha_k {r_k}^n = f_n$$

   para k valores de f_n distintos que calculou separadamente, por exemplo, para $n = 0, 1, \ldots, k-1$.

Exercício 17 *Aplique o método descrito à solução geral da sequência de Fibonacci (ver página 21). Confirme no final 2 valores da solução obtida.*
RESPOSTA (A PREENCHER PELO ALUNO!):

- *A equação geral é...*
- *A equação característica é...*
- *As raízes da equação característica são...*
- *A solução geral é da forma...*
- *A solução é...*

1.3.8 Equação característica homogénea, caso geral: existência de raízes múltiplas

Teorema 6 *Se m é a multiplicidade de uma raíz r da equação característica, as seguintes sucessões, bem como todas as suas combinações lineares, são soluções da equação geral*

$$r^n,\ nr^n,\ n^2r^n, \cdots,\ n^{m-1}r^n$$

Mais geralmente se as raízes da equação característica forem $r_1, \ldots, r_p$ de multiplicidades respectivamente $m_1, \ldots, m_p$, a solução geral é uma qualquer combinação linear da forma

$$\sum_{i=1}^{i=p} \sum_{j=0}^{m_i-1} \alpha_{ij} n^j {r_i}^n$$

Não é difícil mostrar-se que uma combinação linear da forma indicada é sempre solução da equação geral. Também é verdade o recíproco: qualquer solução da equação geral é da forma indicada!.

Exercício 18 *Resolva a recorrência $t_n = 2t_{n-1} - t_{n-2}$, sendo*

$$\begin{cases} t_0 = 0 \\ t_1 = 1 \end{cases}$$

1.3.9 Método da equação característica não homogénea

Vamos apresentar casos particulares em que se conhece a solução.

Suponhamos que a equação geral da recorrência tem a forma

$$a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = b^n p(n)$$

onde

- b é uma constante
- $p(n)$ é um polinómio em n; seja d o seu grau

Teorema 7 *As soluções da recorrência com a equação geral indicada podem obter-se a partir das*

raízes da equação

$$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r-b)^{d+1} = 0$$

usando o método que foi explicado para o caso das equações homogéneas.

Exercício 19 *Determine a forma geral da solução da recorrência*

$$\begin{cases} t_0 = 2 \\ t_n = 2t_{n-1} + 3^n \quad (n \geq 1) \end{cases}$$

Resposta (a preencher pelo aluno!):

- *A equação geral é...*
- *A equação característica é...*
- *Portanto $b = \cdots$, $p(n) = \cdots$, $d = \cdots$.*
- *A solução geral é da forma...*
- *A solução da recorrência é...*

Uma generalização do Teorema 7, equação característica não homogénea.

Teorema 8 *Se a equação geral da recorrência tem a forma*

$$a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \cdots + b_m^n p_m(n)$$

onde os b_i são constantes e os $p_i(n)$ são polinómios em n de grau d_i, então as soluções da recorrência correspondente podem ser obtidas a partir das raízes da equação

$$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r-b_1)^{d_1+1}(r-b_2)^{d_2+1}\cdots(r-b_m)^{d_m+1} = 0$$

usando o método que foi explicado para o caso das equações homogéneas.

Exercício 20 *Determine a forma geral da solução da recorrência*

$$\begin{cases} t_0 = 2 \\ t_n = 2t_{n-1} + n + 2^n \quad (n \geq 1) \end{cases}$$

Nota. Será fornecido o enunciado do Teorema 8 nas provas em que isso for necessário.

1.4 Um exemplo de análise: tempo médio do "quicksort"

Supõe-se que o leitor conhece o algoritmo de ordenação "quicksort" que se esquematiza de seguida[1]

```
// ordena a secção v[a..b] do vector v[ ]
void quicksort(v,a,b)
  if a>=b  return
  else
    m = split(v,a,b)
    quicksort(v,a,m-1)
    quicksort(v,m+1,b)
```

O efeito da execução de `m=split(v,a,b)` é alterar o vector `v[]`, definindo um índice `m` tal que todos os elementos da secção do vector `v[a..m-1]` são menores ou iguais a `v[m]` e todos os elementos da secção do vector `v[m+1..b]` são maiores que `v[m]`.

Seja $n = \mathtt{b} - \mathtt{a} + 1$ o número de elementos a ordenar. Na execução de `m=split(v,a,b)` há $n-1$ comparações envolvendo elementos do vector.

Pretendemos mostrar que o número de comparações $c(n)$ é de ordem $O(n \log n)$; mais precisamente que é sempre $c(n) \leq kn \log n$ para uma constante k apropriada.

Admitimos que os elementos do vector são todos distintos e que todas as $n!$ permutações de elementos são igualmente prováveis. Assim, o valor de `m` dado por `m=split(v,a,b)` tem a mesma probabilidade se ser `a`, de ser `a` + 1,... e de ser `b`; essa probabilidade é $1/n$ (relembra-se que $n = \mathtt{b} - \mathtt{a} + 1$).

Obtemos a recorrência que determina o valor médio do número de comparações

$$\begin{cases} c(n) = 0 & \text{para } n \leq 1 \\ c(n) = \sum_{i=1}^{n} \frac{1}{n}(n - 1 + c(i-1) + c(n-i)) & \text{para } n \geq 2 \end{cases}$$

Note-se que para cada $i \in \{1, 2, \ldots, n\}$ o valor $c(i)$ ocorre 2 vezes na soma anterior; podemos então escrever a recorrência da forma seguinte

$$\begin{cases} c(n) = 0 & \text{para } n \leq 1 \\ c(n) = \frac{2}{n} \sum_{i=1}^{n-1}(n - 1 + c(i)) & \text{para } n \geq 2 \end{cases}$$

Pré-teorema *A solução $c(n)$ da recorrência é majorada por $kn \log n$ em que o valor de k é encontrado na demonstração.*

[1] Este algoritmo será analisado em mais detalhe durante este curso.

Caso base, $n = 1$: temos (da recorrência) $c(1) = 0 \leq k(1 \log 1)$ para qualquer $k > 0$; analogamente se trata o caso $n = 0$.

Passo de indução: suponhamos que $c(i) \leq ki \log i$ para todo o i com $0 \leq i \leq n-1$. Vamos mostrar que $c(n) \leq k(n \log n)$.

$$\begin{aligned} c(n) &= n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} c(i) \\ &\leq n - 1 + \frac{2k}{n} \sum_{i=2}^{n-1} (i \log i) \\ &\leq n - 1 + \frac{2k}{n} \int_{i=2}^{n} (i \log i) di \\ &= n - 1 + \frac{2k(n^2 \log n/2 - n^2/4 - 2ln2 + 1)}{n} \\ &= n - 1 + kn \log n - kn/2 + \alpha \end{aligned}$$

onde $\alpha = 2k(-2 \ln 2 + 1)/n$ tem valor negativo. O teorema fica então demonstrado se for $n - 1 \leq kn/2$, pois fica então $k(n \log n)$. Em particular podemos tomar $k = 2$ e enunciar a versão final do "pré-teorema" anterior.

Teorema 9 *O número médio de comparações efectuado pelo "quicksort" não excede* $2n \log n$.

Vemos pois que o número de comparações efectuadas pelo "quicksort" é, no caso médio, majorado por $2n \log n$. Admitimos que o leitor conhece o comportamento no pior caso do "quicksort" e as situações em que esse comportamento se verifica. O número de comparações efectuado é quadrático. Para mais pormenores ver por exemplo o livro de Cormen, Liederson, Rivest e Stein.

Capítulo 2

Tempo de execução dos algoritmos – complementos

2.1 Tempo de execução

Os recursos mais utilizados para medir a eficiência de um algoritmo A são o tempo de execução e o espaço gasto. Tanto um como outro são função dos dados que representamos por x

$$x \rightarrow \boxed{\text{A}} \rightarrow y$$

Destes 2 recursos consideraremos apenas o tempo de execução que representamos por $t(x)$. Em geral estamos interessados em representar o tempo de execução como função do comprimento (número de bits numa determinada codificação) dos dados, $n = |x|$. Mas pode haver dados do mesmo comprimento que originem tempos de execução muito diferentes, veja-se por exemplo o algoritmo "quick sort"; por outras palavras, t não é função de n, é função de x. O que se faz em geral é tomar uma das seguintes opções

- Considerar o maior dos tempos correspondentes a dados de comprimento n, $T(n) = \max_{|x|=n} t(x)$. A esta medida chama-se tempo no pior caso ("worst case time").
- Considerar o tempo médio correspondente a todos os dados de comprimento n, $\overline{T}(n) = E(t(x))$ supondo-se que todos os dados de comprimento n têm a mesma probabilidade de ocorrência[1]. A esta medida chama-se tempo médio ("average case time").

[1] Na verdade há considerar apenas as sequências de n bits que sejam codificações possíveis das instâncias do problema; a maioria das sequências de bits não faz normalmente sentido como codificação.

Nota. Não havendo ambiguidade, representaremos também por t o tempo no pior caso e o tempo médio.

Por outro lado, não estamos em geral interessados nos valores exactos dos tempos de execução, mas preferimos ignorar factores multiplicativos constantes e comparar os tempos através de *ordens de grandeza*. Este aspecto da questão é tratado, por exemplo nas referências em [1, 2, **?**]. O aluno deve estar familiarizado com as definições e propriedades das ordens de grandeza $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, e $o(\cdot)$. Em muitos casos práticos vai-se pretender apenas obter um limite assimptótico do tempo de execução e ignorar as constantes multiplicativas, isto é, pretende-se estabelecer algo da forma $t(n) \in O(f(n))$ onde $f(n)$ é tão pequeno quanto possível.

2.2 Sobre os modelos de computação

Outra questão que se coloca é "como vamos medir o tempo?". Não será medido certamente em segundos ou micro-segundos mas em termos do número de transições de um modelo de computação apropriado. Há diversas hipóteses; referimos as mais importantes

- *Modelo exacto.* Usamos a máquina de Turing (TM) como modelo de computação e defimos o tempo de uma computação como o número de transições que ocorrem durante a computação.

- *Modelo uniforme.* Usamos um computador de registos e representamos os programas numa linguagem de alto nível ou numa linguagem do tipo "assembler". Para obter o tempo de execução contamos 1 por cada instrução ou teste executado no programa. Dado que, por exemplo, as instruções aritméticas (adição, multiplicação,...) são efectuadas em tempo majorado por uma constante, dito de outra maneira, o seu tempo é $O(1)$, assume-se esse mesmo facto ao medir o tempo de computação.

- *Modelo dos circuitos.* Trata-se de um modelo em que o comprimento dos dados é fixo. Cada circuito "resolve" apenas os problemas cujos dados têm um determinado comprimento. Por outro lado, em vez de um algoritmo, existe um circuito constituído por componentes e ligações. Os componentes (ou circuitos elementares) funcionam em paralelo, o que permite muitas vezes tempos de execução muito mais rápidos. Trata-se de um modelo não-uniforme[2] Falaremos mais do modelo dos circuitos em (6).

O modelo uniforme é muitas vezes bastante conveniente porque está mais próximo das computações usuais, mas está basicamente errado! Não podemos supor que as operações elementares são exe-

[2]O conceito de "uniforme" é usado com 2 sentidos: a *uniformidade* no "modelo uniforme" refere-se à independência do tempo de execução relativamente ao comprimento dos dados, enquanto a *não uniformidade* dos circuitos refere-se ao facto de não existir em princípio uma descrição finita dos circuitos de uma família.

cutadas emtempo $O(1)$, como bem sabem os programadores (por exemplo na área de criptografia) que lidam com inteiros muito grandes. Assim, os algoritmos utilizados usualmente para a adição e a multiplicação têm ordem de grandeza que não é $O(1)$, mas sim $\Theta(\log n)$ e $\Theta(\log^2 n)$ respectivamente, onde n é o maior dos operandos envolvidos. O seguinte exemplo ilustra os graves erros em que se incorre quando se utiliza o modelo uniforme. Considere-se o seguinte algoritmo

```
dado n, inteiro positivo
f(n):
  t=2
  for i=1 to n:
    t = t*t
  return t
```

Usando o modelo uniforme concluimos que o tempo de execução é da forma $a + bn$ onde a e b são constantes. Todavia, se levarmos em consideração o número de bits do resultado e o facto de o processamento de cada um desses bits corresponder a pelo menos um passo da computação, vemos que tal é impossível e verificamos que o tempo real de execução é exponencial no tamanho dos dados.

Exercício 21 *Qual é o valor de* `f(n)` *(expressão matemática fechada)?*

Exercício 22 *Quantos bits tem a representação desse valor na base 2? Que pode afirmar sobre o número de dígitos da representação desse valor numa base arbitrária?*

2.2.1 O modelo externo dos dados

Diversos algoritmos, como por exemplo muitos algoritmos de ordenação e pesquisa, obtêm informação sobre os dados de uma única forma: os dados são comparados entre si. Nestes algoritmos podemos pensar que não existe leitura de dados, mas apenas os resultados das comparações referidas. Um diagrama esquemático destes algoritmos é o seguinte

Por exemplo, um algoritmo de ordenação baseado no modelo externo dos dados, comporta-se exactamente da mesma maneira – isto é, as sucessivas configurações internas são as mesmas – a ordenar `[4,1,8,5,2]` ou `[41,1,8488,55,20]`.

2.2.2 Monotonia de $t(n)$

Para facilitar a análise de um algoritmo, assume-se frequentemente que n tem uma forma específica, por exemplo, $n = 2^p$ ou $n = 2^p - 1$ para um determinado inteiro p. Por outro lado, a forma do algoritmo resulta quase sempre o facto de $t(n)$ (tempo no pior caso ou no caso médio) ser uma função monótona no sentido seguinte

$$[n' \geq n] \Rightarrow [t(n') \geq t(n)]$$

Esta monotonia de $t(n)$ facilita a determinação da ordem de grandeza de $t(n)$ no caso geral, isto é, quando n não tem a forma que se admitiu. Vejamos um exemplo. Seja[3] $t(n) = n \log n$ e $n = 2^p$. Então para um comprimento arbitrário n' (que não é necessariamente uma potência de 2), considere-se o inteiro p tal que $n = 2^p \leq n' < 2^{p+1}$, isto é, $n \leq n' < 2n$. Pela monotonia de $t(n) = n \log n$ temos

$$t(n) = n \log n \leq t(n') \leq 2n \log(2n)$$

Donde $t(n') \leq 2n \log(2n) \leq 2n' \log(2n') = 2n'(1 + \log n')$ e temos também $t(n') \geq n \log n \geq (n'/2) \log(n'/2) = (n'/2)(\log n' - 1)$ o que mostra que se verifica $t(n') \in \Theta(n' \log n')$.

Exercício 23 *Considere a seguinte definição de $t(n)$ para $n \geq 2$:*

$$t(n) = c + \sum_{i=1}^{n-1} t(i)$$

sendo c é uma constante e $t(1) = a$, um valor conhecido. Temos

$$t(n+1) - t(n) = c + \left(\sum_{i=1}^{n} t(i) \right) - c - \left(\sum_{i=1}^{n-1} t(i) \right) = t(n)$$

Resulta: $t(n+1) = 2t(n)$, donde temos a solução $t(n) = a2^{n-1}$.

Esta demonstração está errada! Explique porquê e determine a solução correcta.

[3] ln e log representam respectivamente o logaritmo natural e o logaritmo na base 2.

Exercício 24 *Considere a seguinte função para determinar o menor elemento de um vector* `a[1..n]`*:*

```
Dados: vector a[1..n]
Resultado: min{a[i] : i em {1,2,...,n}}
     min(a):
       sort(a)      -- por ordem crescente
       return a[1]
```

Do ponto de vista da eficiência, esta função é (muito) má. Explique porquê.

2.3 Análise amortizada de algoritmos

Na análise amortizada dos algoritmos considera-se uma sequência de operações e divide-se o tempo total pelo número de operações efectuadas, obtendo-se assim um "tempo médio por operação". Mais geralmente, pode haver operações de vários tipos, sendo atribuído a cada uma um seu custo amortizado.

O conceiro de "custo" generaliza o conceito de "tempo"; o custo é uma grandeza não negativa, em geral inteira, com que se pretende medir os gastos de um determinado recurso como o tempo (caso mais habitual) ou o espaço.

Definição 2 *O* custo amortizado *de uma sequência de n operações é o custo total a dividir por n.*

Quando temos uma sequência de operações op_1, op_2, op_n, podemos considerar o custo máximo de uma operação ("pior caso")

$$c_{\max} = \max\{c_i : i = 1, 2, \ldots, n\}$$

onde c_i representa o custo da operação op_i; o custo médio por operação ("caso médio") ou *custo amortizado* é

$$E(c) = \frac{\Sigma_{i=1}^{n} c_i}{n}$$

Mais geralmente podem existir vários tipos de operações numa estrutura de dados (por exemplo, pesquisa, inserção e eliminação num dicionário) e podemos atribuir a cada tipo de operação um custo amortizado.

Não se devem confundir estes custos (máximo e médio) com os tempos no pior caso e no caso médio (ver página 35), em que a média e o máximo são relativas à distribuição probabilística dos dados.

A análise amortizada é muitas vezes mais significativa do que a análise individual; por exemplo, ao efectuar um determinado processamento de dados, estamos muitas vezes mais interessados no custo total (ou no custo médio por operação) do que no custo individual de cada operação. Por exemplo, se no departamento de informática de um banco se processa, nas horas noturnas, uma longa sequência de transações de diversos tipos, não é importante se o processamento de algumas delas demorar muito tempo, desde que o tempo total não ultrapasse um limite pré-estabelecido. *Se estivessemos a utilizar sistematicamente os custos no pior caso, poderíamos chegar a majorantes do custo total muito elevados e pouco significativos.*

Por outro lado, em sistemas de tempo real (como por exemplo em jogos, nos sistemas de controlo automático associados à navegação aérea, etc.) a análise amortizada poderá não ser conveniente, uma vez que há muitas vezes que garantir majorantes do tempo de cada operação. Pelas mesmas razões, e falando agora da análise baseada na distribuição estatística dos dados, nestes casos prefere-se muitas vezes a análise no pior caso (relativamente à distribuição estatística dos dados) à análise no caso médio.

Vamos de seguida considerar em algum detalhe algumas sequências específicas de operações e fazer a sua análise amortizada.

2.3.1 "Stack" com gestão de memória

Consideremos um "stack" implementado como um vector de dimensão pré-definida

```
Implementação: vector v[0...m-1]
Inicialização: sp=0   (sp é o primeiro índice não ocupado de v)
Operações:
   push(x):    v[sp]=x;   sp++;
   pop():      sp--;      return v[sp];
```

Quando se efectua um `push` e o vector está "cheio", isto é, `sp==m`, há que efectuar previamente a seguinte operação

```
Redimensionar(p):    (onde p>m)
   1) obter um novo espaço de memória com p posições consecutivas
   2) copiar v[0..m-1] para as primeiras m posições do novo espaço
   3) fazer com que v designe para o novo espaço
```

Vamos usar o seguinte modelo de custos:

- `push`: custo 1
- `pop`: custo 1
- `redimensionar`: custo `m`; na verdade, a cópia de `m` elementos domina o tempo de execução desta operação.

A sequência de operações que vamos considerar consiste em n `push`'s. Claro que esta sequência vai eventualmente originar operações de `redimensionar`, mas vamos apenas atribuir um custo amortizado às operações de `push` (e não às de `redimensionar`). Vamos ainda supor que inicialmente o vector tem apenas uma célula disponível, isto é, que o valor inicial de `m` é 0.

Uma questão importante que pode afectar o custo amortizado é: que valor de `p` (como função de `n`) se deve escolher na operação de `redimensionar(p)`? Vamos analisar 2 hipóteses.

1) `p = n + 1`, o vector cresce de 1

Esta é uma má escolha. Vejamos:

	1	2	3		n
op:	push(.)	push(.)	push(.)	...	push(.)
custo:	1	2	3		n

Por exemplo, o custo do último `push` consiste em $n-1$ (operação `redimensionar` que envolve a cópia de $n-1$ valores) mais 1 (custo do `push`). Assim, o custo amortizado é

$$\frac{1+2+\ldots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

isto é, cada `push` tem um custo superior a $n/2$, não sendo portanto $O(1)$!

2) `p = 2n`, o vector cresce para o dobro

Esta opção é muito melhor, pois vai permitir um custo $O(1)$. Sendo $2^k < n \leq 2^{k+1}$, o custo para os redimensionamentos é

$$1+2+4+8+\ldots+2^k = 2^{k+1}-1$$

a que temos de juntar o custo n dos `push`'s. O custo amortizado é

$$\frac{n+2^{k+1}-1}{n} < \frac{n+2^{k+1}}{n} \leq \frac{3n}{n} = 3$$

pois $2^{k+1} = 2 \times 2^k < 2n$. Assim, o custo amortizado de cada `push` é inferior a 3. Este custo amortizado esconde o facto de alguns `push`'s terem um custo muito superior, $\Omega(n)$.

Exercício 25 *Mostre que com uma análise mais cuidada, podemos usar para o mesmo problema o seguintes* custos amortizados*:*

- `push`*: custo 3*
- `pop`*: custo -1*

Pretende-se mostrar que com este modelo a função potencial continua a nunca ser negativa.

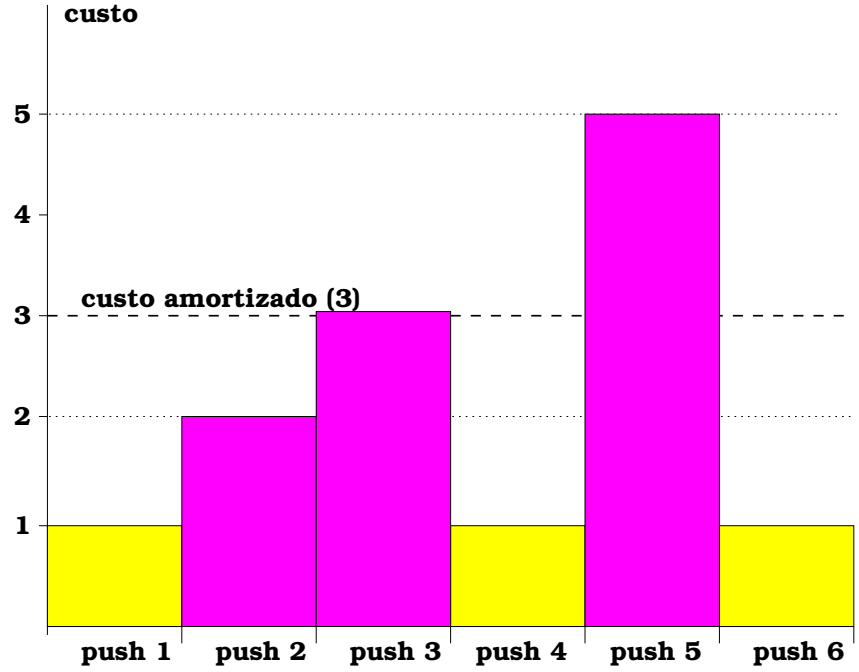
2.3.2 A função potencial

Vamos considerar de novo a sequência de operações anterior (n operações de `push`) com a opção 2 (redimensionamento que duplica o tamanho do vector). Um outro modo de analisar a o custo amortizado desta sequência de operações é o seguinte

- Por cada `push` gastamos 3 unidades (3 euros, por exemplo); dessas 3 unidades, 1 é gasta com o `push` e 2 vão para um saco de poupanças ("mealheiro"); em cada instante, o valor existente no mealheiro é o valor da função potencial.

- Quando há que fazer um redimensionamento, vai-se ao mealheiro buscar o custo respectivo.

Ao conteúdo do mealheiro chamamos "potencial", ver mais à frente a definição formal. Na figura seguinte representa-se o custo de cada operação numa sequência de 6 `push`'s.



Exercício 26 *Represente na figura anterior a função potencial (após cada operação de* `push`*).*

Como podemos concluir que no mealheiro existe sempre o dinheiro suficiente para "pagar" os redimensionamentos? Ou, por outras palavras, como mostrar que a função potencial nunca tem um valor negativo?

Resposta: usando a indução matemática no número de operações. O caso base é trivial. Quando há um redimensionamento de 2^i para 2^{i+1}, é porque $n = 2^i + 1$. Então já houve $n = 2^i$ `push`'s

cada um dos quais contribuiu com 2 unidades para o saco, isto é, para o incremento da função potencial. Mas, após o último redimensionamento (antes do actual, quando o saco continha $n/2$), houve mais $n/2$ contribuições de 2 unidades para o saco, ou seja, antes do `push` actual (que vai obrigar a um redimensionamento de custo n), o saco contém pelo menos $2 \times n/2 = n$ unidades de dinheiro. Note-se que estamos a supor que o último redimensionamento foi possível, isto é, estamos a utilizar a hipótese indutiva.

Definição 3 *A função* potencial $\Phi(s)$ *é uma função do estado s do sistema que satisfaz*

1. $\Phi(0) = 0$*, começa com o valor 0.*

2. $\forall i$, $\Phi(i) \geq 0$*, a função potencial nunca é negativa.*

A função potencial $\Phi(s)$ é usada para a análise amortizada dos algoritmos.

Sejam: a uma quantidade positiva (a vai ser o custo amortizado), c_i o custo da i-gésima operação e $\Phi(0) = 0$, $\Phi(1)$, $\Phi(2)\ldots$ os sucessivos valores da função potencial e suponhamos que o "pagamento" da i-ésima operação é efectuado da seguinte forma

– O valor a é adicionado a $\Phi(i-1)$

– De $\Phi(i-i)$ retira-se o custo da operação corrente, c_i

de modo que $\Phi(i) = \Phi(i-i) + a - c_i$. Então, e supondo que Φ nunca é negativa, o valor a é um majorante do custo amortizado. Na verdade, somando as igualdades $\Phi(i) = \Phi(i-i) + a - c_i$ para $i = 1, 2, \ldots, n$ temos

$$\begin{aligned} \Sigma_{i=1}^{n}\Phi(i) &= \Sigma_{i=1}^{n-1}\Phi(i) + na - \Sigma_{i=1}^{n}c_i \\ na &= \Sigma_{i=1}^{n}c_i + \Phi(n) \\ a &\geq \frac{\Sigma_{i=1}^{n}c_i}{n} \end{aligned}$$

Por outras palavras, a é aceitável como custo amortizado, pois garante-se já "ter pago" em cada instante n todos os custos reais até esse momento.

Em resumo, dada uma constante positiva a (custo amortizado) e os custos das operações c_1, $c_2,\ldots$, uma função potencial $\Phi(i)$ é uma função não negativa definida pela recorrência $\Phi(0) = 0$, $\Phi(i) = \Phi(i-i) + a - c_i$; ao verificarmos que, para os valores dados de a, c_1, $c_2,\ldots$, o valor de $\Phi(i)$ nunca é negativo, estamos a mostrar que a é um (majorante do) custo amortizado.

2.3.3 Outro exemplo, um contador binário

Consideremos um contador representado na base 2 e uma sequência de n operações de incrementar (de 1 unidade). O nosso modelo de custo (de cada incremento) vai ser o número de bits do contador que se modificam. Este custo representa razoavelmente o custo real e leva em particular em conta a propagação dos "carries". Por exemplo, se o contador `010111` é incrementado, fica `011000`, sendo o custo 4.

Vamos mostrar que, embora o custo de algumas operações possa ser elevado, o custo amortizado não ultrapassa 2.

Os primeiros 8 incrementos:

bit 3	bit 2	bit 1	bit 0	custo	potencial
0	0	0	0		0
0	0	0	1	1	1
0	0	1	0	2	1
0	0	1	1	1	2
0	1	0	0	3	1
0	1	0	1	1	2
0	1	1	0	2	2
0	1	1	1	1	3
1	0	0	0	4	1

Análise com base numa função potencial

Vamos definir a seguinte função potencial

$$\Phi(x) = \text{número de 1's em } x$$

(ver figura anterior, "os primeiros 8 incrementos"). É óbvio que esta função satisfaz as condições 1 e 2. da definição. Vamos ver que esta função potencial corresponde ao custo amortizado de 2. Consideremos uma operação arbitrária de incremento,

$$\texttt{xx...x0}\overbrace{\texttt{11...11}}^{m} \rightarrow \texttt{xx...x1}\overbrace{\texttt{00...00}}^{m}$$

onde `x` representa um bit arbitrário e $m \geq 0$ (número de 1's consecutivos no final da representação binária). De uma forma mais compacta, representamos esta transição por $x01^m \rightarrow x10^m$.

Para mostrar que o custo amortizado 2 corresponde exactamente à função de potencial escolhida, analisemos em primeiro lugar a variação da função potencial numa transição $x01^m \rightarrow x10^m$. Representando por $n_1(x)$ o número de 1's de x, temos

– Potencial antes da operação: $n_1(x) + m$

– Potencial depois da operação: $n_1(x) + 1$

Por outras palavras, o número de 1's varia de $1 - m$ (diminui se $m = 0$, aumenta se $m \geq 1$).

Vamos agora ver que este potencial é igual ao valor acumulado (ainda não gasto). Usamos a indução. O caso base (inicial) é trivial. Consideremos novamente a transição $x01^m \rightarrow x10^m$ e suponhamos, pela hipótese indutiva, que o valor acumulado antes da operação é $n_1(x) + m$. De quanto varia o valor acumulado? A contribuição (custo amortizado) é 2 e há $m + 1$ bits que mudam; logo, o valor acumulado varia de $2 - (m + 1) = 1 - m$ (esta variação pode ser positiva, nula ou negativa), o que é exactamente a variação do número total de 1's.

Observação. Após a primeira operação, o número de 1's nunca vai ser nulo, o que equivale a dizer que no fim, o potencial vai ser positivo.

Outra análise

Consideremos n incrementos. Quantas vezes o bit de ordem 0 se modifica? Todas, ou seja n. E o bit de ordem 1? Resposta: $\lfloor n/2 \rfloor$. No total, o número de mudanças de bit, isto é, o custo total, é

$$n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \ldots + 1 \leq n + n/2 + n/4 + \ldots \quad \text{(soma infinita)} \tag{2.1}$$

$$= 2n \tag{2.2}$$

Assim concluímos que o custo amortizado é inferior a 2.

2.3.4 Contador binário com custo exponencial na ordem do bit

Consideremos agora a operação de incremento com outro modelo de custo: modificar um bit de ordem i custa 2^i, exponencial em i. Apesar deste aumento substancial do custo[4], o custo amortizado é apenas de ordem $O(\log n)$.

Exercício 27 *Demonstre esta afirmação, utilizando uma análise análoga à efectuada anteriormente, ver a desigualdade (2.1).*

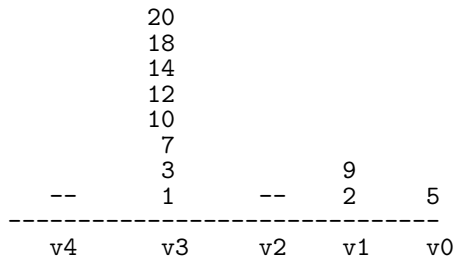
[4]Este modelo pode ser adequado noutras situações como, por exemplo, na análise do custo de um acesso à memória num sistema com uma hierarquia de memórias; além disso, este modelo vai ter uma aplicação na estrutura de dados estudada em seguida.

Aplicação: estrutura de dados com pesquisa e inserção

Vamos definir uma estrutura de informação baseada numa sequência de vectores v_0, v_1, $v_2 \ldots$ em que o vector v_i ou está vazio ou contém 2^i elementos e está ordenado.

Exercício 28 *Mostre que qualquer número de elementos pode ser representado numa estrutura deste tipo.*

Um exemplo desta estrutura



Não se impõe qualquer relação entre os valores contidos nos diferentes vectores.

Exercício 29 *Mostre que a pesquisa de um valor x numa estrutura deste tipo (isto é, x está em algum dos vectores?) pode ser efectuada em tempo $O(\log^2 m)$ onde m é o número total de elementos contidos na estrutura..*

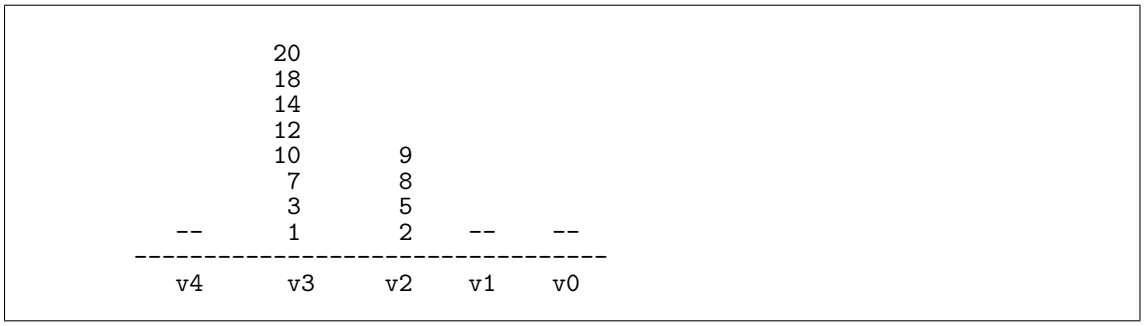
A inserção de um elemento x na estrutura utiliza operações de "merge", conforme se exemplifica para a inserção de 8; seja o vector $w = [x]$

1. Se v_0 está vazio, faça $v_0 = w$, STOP.
2. Se não: faça merge(w,v_0) $\rightarrow w$ (2 elementos). No caso presente fica $w = [5, 8]$. Depois marque v_0 como vazio.
3. Se v_1 está vazio, faça $v_1 = w$, STOP.
4. Se não, faça merge(w,v_1) $\rightarrow w$ (4 elementos). No caso presente fica $w = [2, 5, 8, 9]$. Depois marque v_1 como vazio.
5. Se v_2 está vazio, faça $v_2 = w$, STOP (caso presente).

6. Se não...

7. ...

A estrutura após a inserção de 8:



Modelo de custos:

- Criação do vector inicial: custo 1
- "Merge" de 2 vectores de tamanho k (cada): custo $2k$

Exercício 30 *(i) Qual o custo da inserção de 8 exemplificada atrás? (ii) Mostre que se o valor inserido ficar no vector v_i (inicialmente vazia), então o custo é $\Theta(2^i)$.*

Exercício 31 *Mostre que o custo amortizado da inserção de n valores numa estrutura inicialmente vazia é $O(\log(n))$. Sugestão: utilize o exercício anterior.*