Algoritmos e Complexidade

Introdução à Análise de Complexidade

José Bernardo Barros
Departamento de Informática
Universidade do Minho

1 Introdução

A análise de complexidade é o ramo das ciências da computação dedicado ao estudo dos recursos (tempo de execução, memória usada, energia consumida) necessários à execução de um determinado programa/algoritmo.
De uma forma mais geral, a análise de complexidade pode ser vista como uma ferramenta que nos permite comparar a eficiência relativa entre vários algoritmos que resolvem um dado problema.
Considere-se por exemplo o problema de ordenar um array. A análise da complexidade permite-nos, por um lado determinar quais os recursos necessários para uma função em particular o ordenar, mas também nos permitirá comparar as eficiências relativas de várias estratégias/algoritmos de ordenação.
Quando se pretende determinar os recursos necessários à execução de um determinado procedimento há que ponderar diversos factores.
Suponhamos que vamos analisar apenas o tempo necessário para que um determinado algoritmo de ordenação (de vectores) termine. Uma resposta satisfatória a essa questão deverá ter em consideração que:

- Ordenar um vector com poucos elementos demora menos do que se quisermos ordenar um vector com muitos elementos.
- O tempo gasto na ordenação de um vector varia com o grau de *desordenação* inicial do dito vector.
- O tempo necessário vai depender ainda da máquina concreta onde tal programa vai ser executado.

Vejamos então de uma forma mais geral como vamos incorporar todos estes factores na análise de comparação da complexidade de algoritmos.

O primeiro factor que vamos ter em conta é o **tamanho** do input. Em rigor, este factor expressa o número de *bits* necessários para representar o input. Veremos contudo que muitas vezes é razoável simplificar essa medida: o cálculo do comprimento de uma lista (ligada) de inteiros é tão demorado quanto o cálculo do comprimento de uma lista (ligada) de *strings*.
A forma de lidar com a influência do tamanho do input no análise de complexidade consiste em definir o custo como uma função (no sentido matemático) do tamanho.

O segundo factor que devemos ter em consideração é a **forma** (ou valor) do input: como referimos atrás, ordenar um vector já ordenado é, em princípio, mais fácil (menos complexo) do que se tal não se verificar à partida.

Vamos lidar com este factor através da análise de casos representativos. É costume analisarem-se os seguintes casos:

- **pior caso** estabelecendo um limite superior para o custo
- **melhor caso** estabelecendo um limite inferior para o custo
- **caso médio** que corresponde ao valor esperado (esperança matemática) do custo.

O terceiro factor que vamos ter em conta diz respeito à máquina onde o procedimento será executado.
A forma habitual de lidar com este factor consiste em começar por identificar todos os componentes atómicos do programa em causa. A atomicidade destes componentes não deve ser entendida no seu significado textual[1] mas antes como uma componente cuja execução tem um custo/complexidade constante.
Depois de identificadas estas componentes e fixados os respectivos custos como constantes o custo/complexidade da função a analisar será feito em função dessas constantes, que poderão ser instanciadas com os valores da máquina concreta em análise.
Neste ponto há ainda uma simplificação que é comum e que consiste em identificar uma (ou mais) operações elementares que são significativas no procedimento em causa. Essas operações são escolhidas por serem as que têm associado um maior custo ou por serem as que mais vezes são executadas.
Por exemplo, num algoritmo de ordenação é costume fazer a análise baseada apenas no número de comparações que são feitas entre elementos do vector ou ainda sobre o número de trocas efectuadas entre elementos do vector.

Sumariando o que até aqui dissemos, da análise da complexidade de uma procedimento/algoritmo vai resultar uma expressão paramétrica cujos parâmetros correspondem ao tamanho do input e aos custos das operações elementares utilizadas.

2 Procura num array desordenado

Para melhor entender os conceitos apresentados na introdução vamos prosseguir com a apresentação de um exemplo muito simples – procura de um inteiro num array de inteiros (sem qualquer ordem).
Vamos apenas fazer a análise do tempo de execução dessa função.

```
int search (int x, int N, int v[N]) {
  int i;
  i=0;
  while ((i<N) && (v[i] != x))
    i++;
  if (i==N) return (-1);
  else return i;
}
```

Tal como referimos, vamos considerar três factores nesta análise:

1. tamanho do input

[1] **átomo** deriva do grego **a** (prefixo de negação) + **tomon** (cortar) significando por isso *que não se pode partir*. A versão latina da palavra átomo é **indivíduo**: **in** (prefixo de negação) + **dividuus** (dividir).

2. forma/valor do input

3. máquina

Quanto ao primeiro ponto, trata-se de uma função cujo argumentos (input) são um inteiro (`x`) e um array (`v`). O parâmetro `N` é apenas um pormenor da linguagem C (a dimensão de um array passado como argumento não é conhecida pela função a menos que explicitamente fornecida como argumento). O tamanho do input é por isso $(N+1) * I$ em que I corresponde ao número de bits usados para representar um inteiro. Como veremos adiante, no caso em análise não é relevante ter em consideração o número de bits usados na representação dos inteiros[2].
Vamos então fazer essa análise em função apenas do número de elementos do array argumento. Ou seja, vamos definir uma função de custo $T :: N \to N$ cujo argumento é o tamanho do array em causa.

O segundo factor que vamos ter em consideração é a máquina onde esta função vai ser executada.
Tal como referimos, vamos começar por identificar as operações atómicas presentes e associar a cada uma delas uma constante que representará o custo de executar essa operação na máquina em causa (muitas vezes referida por *máquina abstracta*).
Da inspecção da definição acima resulta a seguinte lista de operações elementares:

Operação		**Custo**
atribuição	(`=` e `++`)	c_1
comparação	(`<`)	c_2
selecção em array	(`v[i]`)	c_3
teste de igualdade	(`/=` e `==`)	c_4

Quanto ao terceiro factor – valor/forma do input – vamos analisar os três casos referidos: melhor caso, pior caso e caso médio. E isto porque a função que estamos analisar pode ter realmente tempos de execução substancialmente diferentes dependendo dos valores dos seus argumentos.
Como podemos ver a variação que pode existir deve-se ao ciclo existente cujo número de iterações vai ser dependente do valor dos argumentos.
O caso em que há menos iterações é quando a condição começa por ser falsa, i.e., o valor que estamos à procura existe logo na primeira posição do array (índice `0`).
O caso em que há mais iterações acontece quando o elemento a procurar não existe no array.
É de realçar que nesta identificação nunca referimos uma outra razão para o ciclo terminar mais cedo ou mais tarde e que tem a ver com o valor de `N`. E isto tem a ver com o facto de isso não fazer parte do valor do input mas apenas do seu tamanho. Ora esse factor já foi tido em consideração e não deve de forma alguma influenciar o resto da análise.

2.1 Melhor caso

Vejamos então o custo associado a uma execução em que o valor a procurar se encontra na primeira posição do array.

$$\begin{aligned} T(N) &= \overbrace{c_1}^{\text{i = 0}} + \overbrace{c_2 + c_3 + c_4}^{\text{teste while}} + \overbrace{c_4}^{\text{teste if}} \\ &= c_1 + c_2 + c_3 + 2 * c_4 \end{aligned}$$

Vemos que neste caso o custo é constante, i.e., não depende do número de elementos no array.

[2] A única operação que depende do número de bits usados na representação de cada elemento do array é a comparação. E o custo desta operação não é substancialmente afectada por esse número (veja-se o exercício 5 na página 14).

2.2 Pior caso

Vejamos agora o que acontece no pior caso, i.e., no caso em que a condição do ciclo `(v[i] != x)` é sempre verdadeira. O custo associado é dado por:

$$\begin{aligned} T(N) &= \overbrace{c_1}^{\texttt{i = 0}} + \overbrace{\sum_{i=0}^{N-1} (\underbrace{c_2 + c_3 + c_4}_{\text{teste while}} + \underbrace{c_1}_{\texttt{i++}}) + \underbrace{c_2}_{\texttt{i<N}}}^{\text{ciclo while}} + \overbrace{c_4}^{\text{teste if}} \\ &= c_1 + (c_2 + c_3 + c_4) * \textstyle\sum_{i=0}^{N-1}(1 + c_1 + c_2) + c_4 \\ &= 2 * c_1 + c_2 + c_4 + (c_2 + c_3 + c_4) * N \\ &= K_1 * N + K2 \end{aligned}$$

Este último passo resulta de definir novas constantes[3] K_1 e K_2 que evidenciam que a função de custo é um polinómio (de grau 1) sobre N.
Este exemplo evidencia ainda que poderíamos ter chegado a esta mesma conclusão, i.e., que o custo desta função é um polinómio de grau 1 sobre N (o tamanho do input) se tivéssemos considerado apenas o custo associado a, por exemplo, aceder a uma posição do array.
Para não estarmos a refazer todos esses cálculos vamos aplicar essa simplificação apenas no cálculo do custo no caso médio.

2.3 Caso médio

Para calcular o valor esperado do custo de execução desta função teremos que identificar todas as possíveis execuções r e, para cada uma dessas determinar o custo c_r e a probabilidade p_r. Feito isto, o custo esperado é dado por

$$\overline{T}(N) = \sum_r p_r * c_r$$

No exemplo em análise é costume fazer este cálculo começando por dividir todas as possíveis execuções em dois grupos:

- casos em que a função retorna um índice do array (sucesso),
- casos de insucesso.

Esta divisão tem como único objectivo o estabelecimento das probabilidades envolvidas. Depois de determinarmos a probabilidade de cada um destes casos (p_{suc} e p_{ins} respectivamente) o custo será dado por

$$\overline{T}(N) = p_{\text{suc}} * \overline{T}_{\text{suc}}(N) + p_{\text{ins}} * \overline{T}_{\text{ins}}(N)$$

O cálculo de $\overline{T}_{\text{ins}}(N)$ corresponde sempre ao mesmo número de iterações do ciclo (no caso de insucesso o ciclo só irá terminar quando a condição `i<N` for falsa):

$$\begin{aligned} \overline{T}_{\text{ins}}(N) &= \textstyle\sum_{i=0}^{N-1} 1 \\ &= N \end{aligned}$$

[3] $K_1 = \frac{1}{2} * c_2 * c_3 * c_4$
$K_2 = 2 * c_1 + \frac{3}{2} * c_2 + \frac{1}{2} * c_3 + \frac{3}{2} * c_4$

No caso de sucesso podemos considerar que o elemento a procurar (e que será encontrado) pode ocorrer, com igual probabilidade (i.e., $\frac{1}{N}$), em qualquer das N posições do array. Daqui resulta o seguinte

$$\begin{aligned}
\overline{T}_{\text{suc}}(N) &= \sum_{i=0}^{N-1} \overbrace{\frac{1}{N}}^{\text{prob (x==v[i])}} * \overbrace{(i+1)}^{\text{custo}} \\
&= \frac{1}{N} * \sum_{i=0}^{N-1} (i+1) \\
&= \frac{1}{N} * \sum_{i=1}^{N} i \\
&= \frac{1}{N} * \frac{N*(N+1)}{2} \\
&= \frac{N+1}{2}
\end{aligned}$$

Para finalizarmos este cálculo precisamos apenas de determinar qual a probabilidade de sucesso e de insucesso.
No caso em análise, trata-se da procura de um inteiro num array de inteiros. Assumindo a completa aleatoriedade desses valores, a probabilidade de sucesso é quase nula: a probabilidade de dois números aleatórios serem iguais é muito pequena ($\frac{1}{2^b}$ em que b é o número de bits usados para representar o elemento). Consequentemente, a probabilidade de dois números serem diferentes é muito grande ($1 - \frac{1}{2^b}$). Daí que a probabilidade de o número não existir no array (ser diferente de todos) ser ainda muito próxima de 1:

$$p_{\text{ins}} = \left(1 - \frac{1}{2^b}\right)^N$$

Conversamente, a probabilidade de sucesso é quase nula pelo que,

$$\begin{aligned}
\overline{T}(N) &= p_{\text{suc}} * \overline{T}_{\text{suc}}(N) + p_{\text{ins}} * \overline{T}_{\text{ins}}(N) \\
&\approx 0 * \frac{N+1}{2} + 1 * N \\
&\approx N
\end{aligned}$$

3 Procura num array ordenado

Na secção anterior analisámos a complexidade de uma função que procura um elemento num array arbitrário.
Outras alternativas surgem se considerarmos que o array em questão está ordenado (por ordem crescente). Nesta secção vamos apresentar e analisar duas alternativas a este problema. Note-se no entanto que a função que apresentámos também está correcta quando o array argumento está ordenado.
Na análise dessas alternativas vamos apenas considerar o número de acessos ao array.

3.1 Procura linear

```
int lsearch (int x, int N, int v[N]) {
  int i;
  i=0;
  while ((i<N) && (v[i] < x))
    i++;
  if ((i==N) || (v[i] != x)) return (-1);
  else return i;
}
```

O custo de execução desta função (em termos do número de acessos ao array) vai depender do número de iterações do ciclo.
O melhor caso acontece quando a condição do ciclo começa por ser falsa, i.e., quando `v[i] <= x`. Nesse caso teremos

$$\begin{aligned} T(N) &= \overbrace{1}^{\texttt{v[i] > x}} + \overbrace{1}^{\texttt{v[i] != x}} \\ &= 2 \end{aligned}$$

O pior caso acontecerá quando o ciclo terminar apenas quando `i>=N`, ou seja, a outra parte da conjunção é sempre verdadeira. Temos por isso que

$$\forall_{0<\leq k<N} \;\; v[k] > x$$

Ou seja, que o valor a pesquisar é maior do que todos os elementos do array.
O custo desta função neste caso é

$$\begin{aligned} T(N) &= \overbrace{\sum_{i=0}^{N-1} \underbrace{1}_{\texttt{v[i] > x}}}^{\texttt{for}} + \overbrace{1}^{\texttt{v[i] != x}} \\ &= N+1 \end{aligned}$$

A análise do melhor e pior casos na comparação dos custos de execução desta função e da apresentada na secção anterior (em que não era feita qualquer assunção sobre a organização do array) parece revelar que não existe uma melhoria clara de eficiência.
Essa melhoria pode-se adivinhar pela análise das condições que caracterizam estes casos extremos. Mas só será evidente quando fizermos a análise do caso médio.
Para analisarmos o comportamento esperado desta função, consideremos que se trata de um array com valores uniformemente distribuídos e que o valor a procurar é aleatório. Nestas condições é válido afirmar que o ciclo em causa pode fazer, com igual probabilidade, de `0` até `N-1` iterações. No caso em que são feitas k iterações, o número de acessos ao array é de $k+2$ pois temos que considerar o acesso que é feito fora do ciclo.
São por isso `N` comportamentos distintos, cada um deles com probabilidade de $\frac{1}{N}$. O custo médio é

$$\begin{aligned} \overline{T}(N) &= \sum_{i=0}^{N-1} \overbrace{\frac{1}{N}}^{\text{prob}} * \overbrace{(i+2)}^{\text{custo}} \\ &= \frac{1}{N} * \sum_{i=0}^{N-1} (i+2) \\ &= \frac{1}{N} * \sum_{i=2}^{N+1} i \\ &= \frac{1}{N} * \frac{N*(N+3)}{2} \\ &= \frac{N+3}{2} \end{aligned}$$

Este resultado está mais perto da nossa intuição sobre a procura num array ordenado: serão acedidas em média metade das posições do array.

Nos cálculos acima foi usada a fórmula de cáculo da soma dos elementos de uma progressão aritmética, que apresentamos de seguida.

Pretendemos calcular a soma (com $b-a+1$ parcelas) $S = \sum_{i=a}^{b} i$

$$S = a + (a+1) + \cdots + (b-1) + b$$

Invertendo a ordem das parcelas

$$S = b + (b-1) + \cdots + (a+1) + a$$

Somando estas duas equações,

$$2 * S = (a+b) + (a+1+b-1) + \cdots + (a+1+b-1) + (a+b)$$

isto é,

$$\begin{array}{rcl} 2 * S & = & \overbrace{(a+b) + (a+b) + \cdots + (a+b) + (a+b)}^{b-a+1 \text{ vezes}} \\ & = & (b-a+1) * (a+b) \end{array}$$

Pelo que,

$$S = \sum_{i=a}^{b} i = \frac{(b-a+1) * (a+b)}{2}$$

Exercício 1 Considere a seguinte função em C que determina se um vector de inteiros contem elementos repetidos.

```
int repetidos (int v[], int N) {
int rep = 0;
int i, j;

    for (i=0; (i<N-1) && !rep; i++)
        for (j=i+1; (j<N) && !rep; j++)
            if (v[i] == v[j]) rep = 1;
    return rep;
}
```

1. Identifique o melhor e o pior casos da execução desta função.
2. Para o pior caso definido acima, calcule o número de comparações (entre elementos do vector) que são efectuadas (em função do tamanho do *array* argumento).

Exercício 2 Considere a seguinte função em C que calcula o número de elementos diferentes de um *array* de inteiros.

```
int diferentes (int v[], int N) {
int dif = 0;
int i, j;

    for (i=0; (i<N); i++){
        for (j=i+1; (j<N) && (v[i] != v[j]); j++);
                if (j==N) dif++;
    }
    return dif;
}
```

1. Identifique o melhor e o pior casos da execução desta função.

2. Para o pior caso definido acima, calcule o número de comparações (entre elementos do vector) que são efectuadas (em função do tamanho do *array* argumento).

4 Operações sobre inteiros

Considere-se a seguinte definição de uma função que calcula o produto de dois números inteiros (em que o multiplicador é não negativo).

```
int prod (int x, int y) {
  int r;
  r=0;
  while (x>0){
    r = r+y; x=x-1;
  }
  return r;
}
```

Para simplificar a análise, vamos considerar apenas o número de adições que são feitas à variável `r`.
Na análise do custo de execução desta função devemos começar por considerar o tamanho do input. Neste caso os argumentos da função são dois inteiros e por isso o tamanho N do input corresponde ao número de bits necessários para representar o inteiro `x`.
Fixado este valor, e como o número de iterações desta função depende do **valor** do parâmetro `x`, devemos fazer uma análise por casos. Para isso devemos notar que, se são necessários N bits para representar um inteiro (sem sinal), a gama de valores representável varia entre:

- 2^{N-1}, que corresponde a todos os bits excepto o mais significativo serem 0, e
- $\sum_{k=0}^{N-1} 2^k = 2^N - 1$, no caso dos bits serem todos 1.

Teremos então que o número de adições feitas à variável `r`, $T_{\text{prod}}(N)$ é:

No melhor caso, que corresponde ao menor valor de `x`, $T_{\text{prod}}(N) = 2^{N-1}$

No pior caso, que corresponde ao maior valor de `x`, $T_{\text{prod}}(N) = 2^N - 1$.

Um algoritmo alternativo para este cálculo, vai alterando tanto o valor da variável `x` (dividindo-a por 2) como de `y` (multiplicando-a por 2) mantendo como invariante que

$$r == x_0 * y_0 - \texttt{x} * \texttt{y}$$

```
int bprod (int x, int y) {
  int r=0;

  while (x>0){
    if (x & 1) r = r+y;
    x=x>>1; y=y<<1;
```

```
  }
  return r;
}
```

Mais uma vez, vamos considerar apenas o número de adições que são feitas à variável `r`. Além disso, continuam válidas as considerações feitas sobre o tamanho do input (número de bits usados na representação de inteiros) bem como a caracterização do melhor e pior casos.
Vejamos então o custo desta função no pior caso, i.e., no caso em que o valor de x_0 é $2^N - 1$ e que corresponde a estarem todos os N bits a 1.
Por cada iteração do ciclo:

- um dos bits de `x` passa a 0,
- é efectuada uma vez a adição.

Donde, no pior caso, $T_{\text{bprod}}(N) = N$

Exercício 3 Considere a seguinte definição de uma função que calcula a potência inteira de um número.

```
float pot (float base, int exp) {
  float r=1;
  while (exp>0) {
    r = r * base;
    exp = exp - 1;
  }
  return r;
}
```

Apresente uma versão alternativa desta função cujo número de multiplicações, no pior caso, seja proporcional ao número de *bits* usados para representar o expoente (Sugestão: use como inspiração as funções apresentadas no exemplo anterior para calcular o produto de dois números).

5 Análise de caso médio

Nesta secção vamos analisar com mais detalhe o custo médio de algumas funções simples.

5.1 Procura binária

```
int bsearch (int x, int N, int v[N]) {
  int i,s,m;
  i=0; s=N-1;
  while (i<s){
    m = (i+s)/2;
    if (v[m] == x) i = s = m;
    else if (v[m] > x) s = m-1;
    else i = m+1;
  }
  if ((i>s) || (v[i] != x)) return (-1);
```

```
  else return i;
}
```

Na análise que vamos apresentar do comportamento desta função tomaremos apenas em consideração o número de acessos ao vector.
Assim sendo podemos constatar que este número é determinado pelo número de iterações do (único) ciclo. Se o ciclo executar n vezes significa que o número de acessos ao array é $2 * n + 1$ (há um acesso que é feito fora do ciclo).
Concentremo-nos então em determinar o número de iterações que este ciclo efectua.

O melhor caso acontece quando o valor a procurar se encontra no (primeiro) índice acedido (i.e., `(N-1)/2`). Neste caso o ciclo efectua apenas 1 iteração.

O pior caso acontece quando o valor a procurar não se encontra no array. Para calcularmos o número de iterações que são feitas devemos ter em conta que o ciclo termina quando a diferença entre os valores de `i` e `s` deixar de ser positiva. Ora esta diferença diminui para metade por cada iteração do ciclo. Daí que, o número máximo de iterações do ciclo corresponda ao número em que se consegue dividir a diferença inicial (N) entre essas variáveis, e que corresponde a $\log_2 N$.

De forma a calcularmos o número médio de iterações do ciclo, concentremo-nos em primeiro lugar no caso em que esta procura termina com sucesso (o outro caso, quando termina em insucesso já foi analisado em cima e corresponde ao pior caso).
Assumindo que o elemento existe com igual probabilidade em qualquer posição do array, as possíveis execuções deste ciclo correspondem a

- existe 1 possibilidade de o ciclo executar uma única vez: ou seja o custo será de 1 com probabilidade $\frac{1}{N}$;
- existem 2 possibilidades de o ciclo executar duas vezes: ou seja o custo será de 2 com probabilidade $\frac{2}{N}$;
- existem 4 possibilidades de o ciclo executar três vezes: ou seja o custo será de 3 com probabilidade $\frac{4}{N}$;
- ...
- de uma forma genérica, existem 2^{k-1} possibilidades de o ciclo executar k vezes: ou seja o custo será de k com probabilidade $\frac{2^{k-1}}{N}$.

O custo esperado é então dado por

$$
\begin{aligned}
& \textstyle\sum_{k=1}^{\log_2 N-1} k * \frac{2^{k-1}}{N} \\
= \; & \tfrac{1}{N} * \textstyle\sum_{k=1}^{\log_2 N-1} k * 2^{k-1} \\
= \; & \tfrac{1}{N} * ((\log_2(N) - 1) * 2^{log_2(N)} - (\log_2(N) * 2^{\log_2(N)-1}) + 1) \\
= \; & \tfrac{1}{N} * ((\log_2(N) - 1) * N - \tfrac{1}{2} * N * \log_2(N) + 1) \\
= \; & \tfrac{1}{N} * (\tfrac{1}{2} * N * \log_2(N) - N + 1) \\
= \; & \tfrac{1}{2} * \log_2(N) - 1 + \tfrac{1}{N}
\end{aligned}
$$

O cálculo apresentado usa a seguinte propriedade:

$$\sum_{i=1}^{n} i * 2^{i-1} = n * 2^{n+1} - (n+1) * 2^n + 1$$

que passamos a demonstrar de seguida.
Comecemos por apresentar o resultado da soma dos elementos de uma progressão geométrica de razão x. Pretendemos calcular a soma

$$S = \sum_{i=0}^{n} x^i = 1 + x + x^2 + x^3 + \cdots + x^{n-1} + x^n$$

multiplicando ambos os membros desta equação por x, obtemos

$$S * x = x + x^2 + x^3 + \cdots + x^{n-1} + x^n + x^{n+1}$$

subtraindo estas duas equações,

$$\begin{aligned} S * x - S &= x^{n+1} - 1 \\ S * (x-1) &= x^{n+1} - 1 \end{aligned}$$

Pelo que,

$$S = \sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Derivemos ambos os lados desta equação (em ordem a x)

$$\begin{aligned} S' &= (1 + x + x^2 + x^3 + \cdots + x^n)' \\ &= 0 + 1 + 2 * x + 3 * x^2 + \cdots + n * x^{n-1} \\ &= \textstyle\sum_{i=1}^{n} i * x^{i-1} \end{aligned}$$

$$\begin{aligned} (\tfrac{x^{n+1}-1}{x-1})' &= \tfrac{(n+1)*x^n*(x-1)-(x^{n+1}-1)}{(x-1)^2} \\ &= \tfrac{n*x^{n+1}-(n+1)*x^n+1}{(1-x)^2} \end{aligned}$$

Pelo que

$$\sum_{i=1}^{n} i * x^{i-1} = \frac{n * x^{n+1} - (n+1) * x^n + 1}{(1-x)^2}$$

A propriedade apresentada corresponde a esta última fazendo $x = 2$.

5.2 Incremento

Considere-se a seguinte função `inc` que, recebe como argumento um vector com os *bits* de um número, e altera esse vector de forma a representar o sucessor do número original.
Assim, por exemplo, para o vector representar o número 119 (cuja representação binária é `1110111`) deve ter todos os elementos a zero excepto os 7 últimos, que devem guardar a sequência `1 1 1 0 1 1 1`. A invocação da função `inc` com esse vector, deverá alterar apenas as 4 últimas posições do vector de forma a representar o número 120 (cuja representação binária é `1111000`).

```
int inc (int b[], int N) {
int i, r=0;

    i = N-1;
    while ((i >= 0) && (b[i] == 1)) {
        b[i] = 0;
        i--;
    }
    if (i>=0) b[i] = 1;
    else r = 0;

    return r;
}
```

A complexidade desta função pode ser reduzida ao número de bits alterados, que varia entre

- 1 no (**melhor**) caso de o bit menos significativo ser `0`
- N no (**pior**) caso de os últimos $N-1$ bits serem `1`.

Para determinarmos o número médio de bits que são alterados vamos assumir que o número representado no vector de bits é um número aleatório. Isto é equivalente a dizer que cada posição do vector pode ter, com a mesma probabilidade, os valores `0` ou `1`.
Desta forma,

- a probabilidade de mudar apenas 1 bit é $\frac{1}{2}$ (corresponde à probabilidade de o bit menos significativo ser `0`).
- a probabilidade de mudar apenas 2 bits é $\frac{1}{4}$ (corresponde à probabilidade de o bit menos significativo ser `1` e o seguinte ser `0`).
- a probabilidade de mudar apenas 3 bits é $\frac{1}{8}$ (corresponde à probabilidade de os 2 bits menos significativos serem `1` e o seguinte ser `0`).
- a probabilidade de mudar apenas k bits ($0 \leq k < N$) é $\frac{1}{2^k}$ (corresponde à probabilidade de os $k-1$ bits menos significativos serem `1` e o seguinte ser `0`).
- Existem dois casos em que o número de bits mudados é N que correspondem às duas situações em que todos os $N-1$ bits menos significativos são `1`. Nestes casos fazem-se sempre N alterações.

O número médio de bits alterados é por isso dado pela expressão

$$\overline{T}(N) = (\sum_{k=1}^{N} \frac{k}{2^k}) + \frac{N}{2^N} = 2 - \frac{1}{2^{N-1}}$$

A razão de ser desta proximidade do caso médio e do melhor caso, ou por oposição, da diferença tão acentuada entre o caso médio e o pior caso, deve-se à pouca probabilidade de o pior caso acontecer.

5.3 Complemento para dois

Vamos finalizar esta secção com a apresentação de um outro exemplo de manipulação dos bits de um número – o cálculo do complemento para dois.
A seguinte função calcula o complemento para dois de um número inteiro armazenado num array de *booleanos*.

```
void complemento (char b[], int N){
  int i = N-1;
  while ((i>0) && !b[i])
    i--;
  i--;
  while (i>=0) {
    b[i] = !b[i]; i--;
  }
}
```

Para calcularmos o número médio de bits que são alterados por esta função vamos assumir que o valor do input é perfeitamente aleatório, i.e., que a probabilidade de cada posição do array ser um `0` ou `1` é 0.5.
A função em causa tem N comportamentos distintos. Para cada um deles a soma do número de iterações dos dois ciclos é $N-1$.
Como o único dos ciclos que depende do valor do array é o primeiro, o pior caso e melhor casos cada função correspondem ao melhor e pior casos deste 1^{o} ciclo respectivamente.
Para o cálculo do valor médio, podemos ter o seguinte em consideração

- o 1^{o} ciclo vai executar 0 vezes com probabilidade de 0.5; por isso o 2^{o} ciclo executa $N-1$ vezes com probabilidade de 0.5;
- o 1^{o} ciclo executa 1 vez com probabilidade $0.25 = 0.5^2$; por isso o 2^{o} ciclo executa $(N-2)$ vezes com probabilidade 0.5^2;
- ...
- o 1^{o} ciclo executa k vezes com probabilidade $(0.5)^(k+1)$; por isso o 2^{o} ciclo executa $(N-(k+1))$ vezes com probabilidade $(0.5)^(k+1)$

Somando tudo,

$$\begin{aligned}
\overline{T}(N) &= \textstyle\sum_{k=0}^{N-1} \frac{N-(k+1)}{2^{k+1}} \\
&= (N-1) * (\textstyle\sum_{k=0}^{N-1} (\frac{1}{2})^{k+1}) - \sum_{k=0}^{N-1} (k * \frac{1}{2}^{k+1}) \\
&= (N-1) * (\textstyle\sum_{k=1}^{N} (\frac{1}{2})^{k}) - \sum_{k=1}^{N-1} (k * \frac{1}{2}^{k+1}) \\
&= (N-1) * (1 - \frac{1}{2^N}) - ((N-1) * \frac{1}{2^N} - N * \frac{1}{2^{N-1}} + 1) \\
&= N - 2 - \frac{2*N-1}{2^{N-1}}
\end{aligned}$$

Para justificar este último passo, relembremos a propriedade apresentada atrás:

$$\sum_{i=1}^{n} i * x^{i-1} = \frac{n * x^{n+1} - (n+1) * x^n + 1}{(1-x)^2}$$

Note-se agora que

$$\begin{aligned}\sum_{i=1}^{n} i * x^{i+1} &= x^2 * \sum_{i=1}^{n} i * x^{i-1} \\ &= x^2 * \frac{n*x^{n+1}-(n+1)*x^n+1}{(1-x)^2}\end{aligned}$$

Exercício 4 Relembre a seguinte função de consulta de uma árvore binária de procura:

```
int elem (BTree a, int x) {
  while (a != NULL)
   if (a->info == x) break;
   else if (a->valor > x) a = a->left;
   else a = a->right;
        return (a != NULL)
}
```

Admitindo que se trata de uma árvore perfeitamente balanceada,

1. Determine o tempo médio de execução desta função, no caso de o elemento pertencer à árvore. Admita que o valor a procurar está com igual probabilidade em qualquer posição da árvore. Note que uma árvore balanceada com N nodos tem aproximadamente $\log_2 N$ níveis.

2. Calcule o tempo de execução desta função no caso de insucesso (i.e., no caso de o elemento não existir na árvore). O que pode concluir sobre o comportamento assimptótico médio desta função?

Exercício 5 Considere a seguinte definição da função `strcmp` (*Kerningham & Ritchie*).

```
int strcmp(char s[], char t[])
{ int i;
  for (i = 0; s[i] == t[i]; i++)
    if (s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```

De forma a estudar a complexidade desta função em termos do número de acessos às strings em causa

1. Identifique o melhor e pior casos da execução desta função. Uma vez que isso depende dos comprimentos das duas strings, apresente o resultado em função de um par de valores que representam o comprimento de cada uma das strings.

2. Calcule a complexidade média desta função. Para isso assuma que se tratam de duas strings aleatórias em que a probabilidade de dois caracteres serem iguais é de $\frac{1}{256}$.

Exercício 6 Considere a seguinte definição de uma função que calcula o número de bits a 1 na representação de um número inteiro (*hamming weight*).

```
int hamming (unsigned int x){
  int r=0;
  while (x!=0) {
    if (x&1) r++;
    x=x>>1;
```

```
  }
  return r;
}
```

1. Identifique o melhor e pior casos do custo da execução (número de iterações do ciclo `while`) desta função em termos do número de bits usados na representação dos números inteiros.
2. Assumindo uma amostra aleatória, diga qual a probabilidade de acontecer cada um desses casos extremos.
3. Calcule o custo médio da execução (número de iterações do ciclo `while`) desta função.

Uma forma alternativa de calcular o número de bits a `1`, da autoria de Brian Kernighan, baseia-se na relação existente entre a representação binária de um número e do seu antecessor. Dado um inteiro `x` representado em `N` bits, a representação de `x-1` pode ser obtida substituindo todos os bits menos significativos que sejam `0` por `1` e o `1` menos significativo por `0`.
Veja-se, por exemplo, as representações dos números 4044 e 4043:

4044	1	1	1	1	1	1	0	0	1	1	0	0

4043	1	1	1	1	1	1	0	0	1	0	1	1

Se aplicarmos o operador `&` (**e** bit a bit) a estas duas representações obtemos

4044 & 4043	1	1	1	1	1	1	0	0	1	0	0	0

É de notar que esta representação difere da de `x` apenas num bit: o `1` menos significativo.
Com isto em mente podemos apresentar a solução de Brian Kernighan:

```
int hamming_BK (unsigned int x){
  int r=0;
  while (x!=0) {
    x = x&(x-1);
    r++;
  }
  return r;
}
```

Exercício 7 O pior caso de execução desta última função coincide com o pior caso da primeira função apresentada.
Sabendo (por inspecção do código acima) que o número de iterações do ciclo corresponde ao número de `1`'s existentes na representação binária do argumento, calcule o número **médio** de iterações do ciclo em função do tamanho N (número de bits usados na representação de inteiros) desta função.

6 Análise Assimptótica

Tal como referimos na introdução, um dos propósitos da análise de complexidade é a comparação das eficiências relativas de várias algoritmos/procedimentos.
Dessa forma é costume agruparem-se numa mesma categoria, procedimentos que, não tendo exactamente a mesma função de custo, correspondem a procedimentos que para valores elevados do tamanho do input têm *performances* comparáveis.
A forma de fazer isso é através do estudo do crescimento assimptótico das funções.
Uma outra forma de entender as definições que apresentaremos adiante é como formas de abstraír uma função de custo de forma a preservar a sua essência.

Nas definições que a seguir se apresentam vamos assumir que as funções apresentadas são funções reais de variável real. Na maioria dos casos é ainda razoável assumir que se trata de funções monótonas, crescentes e positivas.

A forma menos abstracta de comparar duas funções é usando a igualdade extensional:

$$f = g \quad \text{sse} \quad \forall_x \; f(x) = g(x)$$

Com esta definição serão iguais as funções f e g e diferentes as funções f e h definidas por

$$\begin{aligned} f(x) &= (x+2)^2 \\ g(x) &= x^2 + 4 * x + 4 \\ h(x) &= (x+4)^2 \end{aligned}$$

Uma definição menos restritiva corresponde a

$$f \sim g \quad \text{sse} \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$$

Esta definição corresponde à comparação assimptótica de funções e é fácil de ver que no exemplo acima $f \sim g \sim h$.
Enquanto que a primeira definição relaciona funções com exactamente os mesmos valores, a segunda definição, mais abstracta, relaciona funções que têm, pelo menos a partir de um certo valor, **taxas de crescimento**[4] iguais.
As definições seguintes são usadas para caracterizar os limites superiores do crescimento de uma função.
Para uma dada função g, a classe (conjunto) de funções $o(g(x)$ define-se por

$$f \in o(g(x) \quad \text{sse} \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$$

Apesar de $o(g)$ denotar uma classe (conjunto) de funções, é costume escrever $f = o(g)$ em vez de $f \in o(g)$.

[4] É de realçar que o que está a ser comparado nesta relação não é o valor das funções mas sim a sua taxa de crescimento. Para dizermos que os valores de duas funções se aproximam diríamos antes

$$\lim_{x\to\infty} \mid f(x) - g(x) \mid = 0$$

Note-se que, pela definição de limite de uma função, esta definição é equivalente a

$$\begin{array}{lll} f \in o(g) & \text{sse} & \forall_{C>0}\exists_{n_0}\ \forall_{n\geq n_0}\ \left|\ \frac{f(x)}{g(x)}\ \right| \leq C \\ & \text{sse} & \forall_{C>0}\exists_{n_0}\ \forall_{n\geq n_0}\ \left|\ f(x)\ \right| \leq C * \left|\ g(x)\ \right| \end{array}$$

A variável C, quantificada aqui universalmente, representa a diferença da taxa de crescimento das duas funções. O que esta quantificação diz é que esta diferença pode ser arbitrariamente pequena.

Uma definição mais abrangente e mais comum em análise de complexidade é

$$f \in \mathcal{O}(g) \quad \text{sse} \quad \exists_{C>0}\exists_{n_0}\ \forall_{n\geq n_0}\ \left|\ f(n)\ \right| \leq C * \left|\ g(n)\ \right|$$

Mais uma vez, é costume escrever-se $f = \mathcal{O}(g)$ em vez de $f \in \mathcal{O}(g)$.
Note-se a semelhança entre a definição de $\mathcal{O}(g)$ e de $o(g)$: enquanto que atrás dizíamos que a diferença entra as taxas de crescimento era arbitrariamente pequena (e por isso esta diferença C surgia quantificada universalmente), nesta última definição ela surge quantificada existencialmente.

Esta relação entre funções é reflexiva ($f = \mathcal{O}(f)$) e transitiva (se $f = \mathcal{O}(g)$ e $g = \mathcal{O}(h)$ então $f = \mathcal{O}(h)$).
É então possível definir uma relação de equivalência

$$f \in \Theta(g) \quad \text{sse} \quad f \in \mathcal{O}(g)\ \wedge\ g \in \mathcal{O}(f)$$

Exemplo 1 Considerem-se as seguintes funções f, g e h assim definidas

$$\begin{array}{rcl} f(x) & = & x^2 - 3 * x - 10 \\ g(x) & = & 10 * x^2 + 20 * x + 10 \\ h(x) & = & 100 * x + 50 \end{array}$$

Comecemos por inspeccionar o valor destas três funções em alguns casos

x	$f(x)$	$g(x)$	$h(x)$
0	-10	10	50
10	60	1210	1050
100	9690	102010	10050
1000	996990	10020010	100050
10000	99969990	1000200010	1000050

De seguida vamos comparar o crescimento destas funções usando a notação o.

- $f \neq o(g)$

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{x^2 - 3 * x - 10}{10 * x^2 + 20 * x + 10} = 0.1$$

- $h = o(f)$

$$\lim_{x\to\infty} \frac{h(x)}{f(x)} = \lim_{x\to\infty} \frac{100 * x + 50}{x^2 - 3 * x - 10} = 0$$

- $h = o(g)$

$$\lim_{x\to\infty} \frac{h(x)}{g(x)} = \lim_{x\to\infty} \frac{100 * x + 50}{10 * x^2 + 20 * x + 10} = 0$$

Vamos concluir este exemplo comparando as funções f e g usando a notação $\mathcal{O}$

- Para mostrar que $g = \mathcal{O}(f)$ temos que mostrar que

$$\exists_{C>0}\exists_{x_0}\ \forall_{x\geq x_0}\ \mid g(x) \mid \leq C * \mid f(x) \mid$$

E para isso temos que apresentar uma constante positiva C que nos permita afirmar que, a partir de um ponto x_0 se verifica[5]

$$\mid g(x) \mid \leq C * \mid f(x) \mid$$
$$\Leftrightarrow$$
$$10 * x^2 + 20 * x + 10 \leq C * (x^2 - 3 * x - 10)$$

Seja $C \doteq 11$. Substituindo,

$$10 * x^2 + 20 * x + 10 \leq 11 * (x^2 - 3 * x - 10)$$
$$\Leftrightarrow$$
$$x^2 - 53 * x - 120 \geq 0$$

O polinómio $x^2 - 53 * x - 120$ tem duas raízes reais $\frac{53 \pm \sqrt{3289}}{2}$ o que significa que só é negativo entre essas raízes. Fica então descoberto o ponto $x_0 = 111 > \frac{53+\sqrt{3289}}{2} \simeq 110.34$ a partir do qual esta inequação é verdadeira.

- A prova de que $f = \mathcal{O}(g)$ é em tudo semelhante à anterior e é deixada como exercício.

O que nos permite concluir que, a menos de uma constante, as funções f e g têm a mesma taxa de crescimento. Escrevemos por isso que $f = \Theta(g)$ (e consequentemente $g = \Theta(f)$).

Este exemplo ilustra que funções polinomiais do mesmo grau (independentemente do coeficiente respectivo) têm taxas de crescimento semelhante. É por isso costume denotar, por exemplo, por $\Theta(N^2)$ a classe de funções polinomiais de grau 2, e por $\mathcal{O}(N^2)$ a classe de funções limitada superiormente por um polinómio de grau 2.
Na tabela seguinte apresentam-se alguns dos representantes canónicos de algumas classes de complexidade comuns[6]:

Classe	**Nome**
$\mathcal{O}(1)$	Constante
$\mathcal{O}(\log N)$	Logarítmico
$\mathcal{O}(N)$	Linear
$\mathcal{O}(N * \log N)$	Quasi-linear
$\mathcal{O}(N^2)$	Quadrático
$\mathcal{O}(N^c), c \geq 1$	Polinomial
$\mathcal{O}(c^N), c > 1$	Exponencial

Na tabela seguinte apresentam-se alguns valores destas funções que evidenciam a diferença nas suas taxas de crescimento.

$\mathbf{N}$	$\frac{\mathbf{1}}{\mathbf{N}} = \mathcal{O}(\mathbf{1})$	$\log_{\mathbf{2}}(\mathbf{N})$	$\mathbf{N} * \log_{\mathbf{2}} \mathbf{N}$	$\mathbf{N^2}$	$\mathbf{N^5}$	$\mathbf{2^N}$
1	1	0	0	1	1	2
10	0.1	3.01	30.10	100	100000	1024
100	0.01	6.02	602.06	10^4	10^{10}	$1.2 * 10^{30}$
1000	0.001	9.03	9030.89	10^6	10^{15}	$10.7 * 10^{300}$

Como é dificil apercebermo-nos da magnitude de certos números, recorde-se que

[5]Note-se que as funções em causa, para valores acima de um dado valor, são ambas positivas e por isso o seu valor absoluto coincide com o valor das funções.

[6]Fonte: `https://en.wikipedia.org/wiki/Big_O_notation`

- o número de átomos que se supõe existirem no universo é da ordem de 10^{82}.
- a idade do universo (tempo desde o *big-bang*) é da ordem de $4.36 * 10^{20}$ milisegundos.

Exercício 8 Dado um vector `v` de `N` números inteiros, a mediana do vector define-se como o elemento do vector em que

- existem no máximo `N/2` elementos (estritamente) menores do que ele, e
- existem no máximo `N/2` elementos (estritamente) maiores do que ele.

Se o vector estiver ordenado, a mediana corresponde ao valor que está na posição `N/2`.

1. Considere a seguinte definição de uma função que calcula a mediana de um vector.

```
int mediana (int v[], int N) {
  int i, m, M;
  for (i=0; i<N; i++) {
    quantos (v,N, v[i], &m, &M);
    if (m <= N/2) && (M <= N/2) break;
  }
  return v[i];
}
```

   Assumindo que a função `quantos` executa em tempo linear no comprimento do vector de input, identifique o melhor e pior caso de execução da função `mediana`. Para cada um desses casos determine a complexidade assimptótica da função `mediana`.

2. Calcule a complexidade média da função apresentada na alínea anterior. Para isso, assuma que os valores do vector são perfeitamente aleatórios e, por isso, que a probabilidade de o elemento numa qualquer posição do vector ser a mediana é uniforme ($= \frac{1}{N}$).

7 Definições recursivas

As funções analisadas atrás eram funções iterativas e por isso a contagem das operações envolvidas podia ser expressas directamente como o somatório (para todas as iterações desses ciclos) das operações envolvidas em cada iteração.
Há porém muitos casos em que ou o número de iterações dos ciclos não é directamente controlada por uma variável que vai sendo incrementada (veja-se por exemplo o procedimento de procura binária apresentado atrás) ou até porque a repetição é resultado de uma definição recursiva.
Nestes casos é costume começar por definir a função de custo de uma forma recursiva.
A estas definições é costume chamar-se **equações de recorrência**.
Considere-se por exemplo a seguinte definição recursiva da função `maxInd` que determina, num array de inteiros o índice do maior elemento do array.

```
int maxInd (int v[], int N) {
  int i;

  if (N==1) i = 0;
```

```
  else {
    i = maxInd (v,N-1);
    if (v[N-1] > v[i]) i = N-1;
  }
  return i;
}
```

Vamos agora definir $T(N)$ como o número de comparações entre elementos do array que esta função efectua, assumindo que o array passado como argumento tem tamanho N. Da inspecção da definição acima resulta a seguinte definição (recursiva) de T:

$$T(N) = \begin{cases} 0 & \text{se } N = 1 \\ \underbrace{T(N-1)}_{\texttt{maxInd (v,N-1)}} + 1 & \text{se } N > 1 \end{cases}$$

Esta definição recursiva (muitas vezes referida como uma **relação de recorrência**) pode ser *resolvida*, i.e., podemos determinar uma definição alternativa (e não recursiva).
Expandindo os primeiros termos podemos fácilmente induzir a solução geral

$$\begin{aligned} T(1) &= 0 \\ T(2) &= 1 + T(1) = 1 + 0 = 1 \\ T(3) &= 1 + T(2) = 1 + 1 + 0 = 2 \\ T(4) &= 1 + T(3) = 1 + 1 + 1 + 0 = 3 \\ &\cdots \\ T(N) &= \underbrace{1 + 1 + \cdots + 1}_{N-1 \text{ vezes}} + 0 \\ &= N - 1 \end{aligned}$$

Exemplo 2 Considere-se a seguinte definição recursiva da procura binária num array ordenado.

```
int bsearch (int x, int v[], int N){
  int i;
  if (N<=0) i = -1;
  else {
    m = N/2;
    if (v[m] == x)
      i = m;
    else if (v[m] > x)
      i = bsearch (x,v,m);
    else {
      i = bsearch (x,v+m+1, N-m-1);
      if (i != -1) i = i+m+1
    }
    return i;
}
```

Calculemos então $T(N)$ como o número de comparações feitas por esta função quando se procura um elemento que não existe (pior caso) num array com `N` elementos.

Da inspecção da definição acima no caso identificado resulta a seguinte relação de recorrência:

$$T(N) = \begin{cases} 0 & \text{se } N = 0 \\ \\ T(\frac{N}{2}) + 2 & \text{se } N > 0 \end{cases}$$

Mais uma vez, expandindo alguns termos podemos fácilmente induzir a solução geral

$$\begin{aligned} T(0) &= 0 \\ T(1) &= T(2^0) = 2 \\ T(2) &= T(2^1) = 2 + T(\tfrac{2^1}{2}) = 2 + 2 = 4 \\ T(4) &= T(2^2) = 2 + T(\tfrac{2^2}{2}) = 2 + 2 + 2 = 6 \\ T(8) &= T(2^3) = 2 + T(\tfrac{2^3}{2}) = 2 + 2 + 2 + 2 = 8 \\ &\cdots \\ T(2^i) &= \underbrace{2 + 2 + \cdots + 2}_{i \text{ vezes}} + 2 = 2 * i + 2 \end{aligned}$$

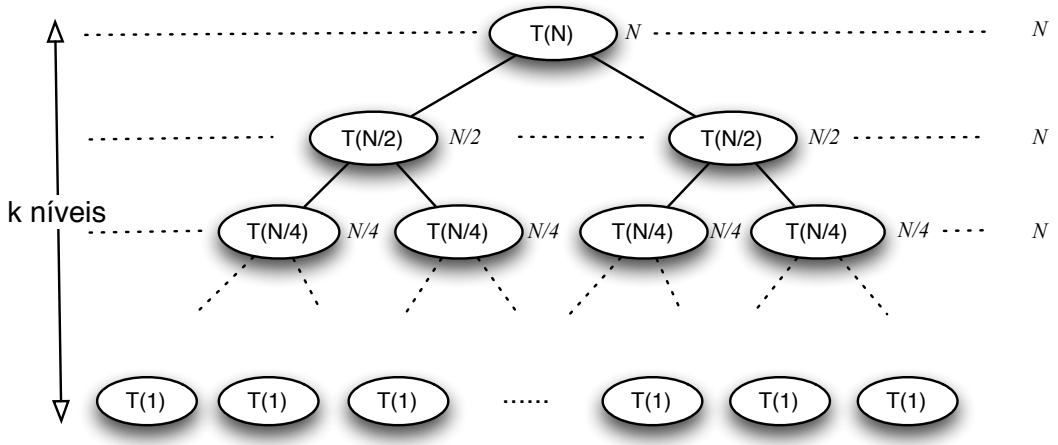
E por isso,

$$T(N) = T(2^{\log_2(N)}) = 2 * \log_2(N) + 2$$

As relações de recorrência resultantes da análise dos progrmas não são sempre tão lineares.
Uma forma de ajudar a induzir o resultado consiste em *desenhar* a expansão dos termos sob a forma de uma árvore (muitas vezes denominada **árvore de recursão**).
Veja-se por exemplo a seguinte relação de recorrência.

$$T(N) = \begin{cases} c_1 & \text{se } N = 0 \\ \\ N + 2 * T(\frac{N}{2}) & \text{se } N > 0 \end{cases}$$

Para um valor de N genérico (e suficientemente grande) podemos apresentar a expansão de $T(N)$ da seguinte forma



Esta forma de apresentar a expansão da relação de recorrência permite-nos organizar os cálculos de uma outra forma. Neste caso podemos constatar que em cada nível da árvore, a soma dos vários custos é N. Se a árvore tiver k níveis, a soma de todos esses níveis será então dada por $k * N$. Ora a altura desta árvore (i.e., o número de níveis) corresponde ao número de vezes que se pode dividir N por 2, i.e., $\log_2(N)$.
Destas observações resulta a seguinte solução:

$$\begin{aligned} T(N) &= \log_2(N) * N + 2^{1+\log_2(N)} * c_1 \\ &= \log_2(N) * N + 2 * N * c_1 \end{aligned}$$

Exercício 9 Utilize uma árvore de recorrência para encontrar limites superiores para o tempo de execução dados pelas seguintes recorrências (assuma que para todas elas $T(0)$ é uma constante):

1. $T(n) = n + T(n-1)$
2. $T(n) = n + T(n/2)$
3. $T(n) = k + 2 * T(n-1)$ com k constante.
4. $T(n) = n + 2 * T(n/2)$
5. $T(n) = k + 2 * T(n/3)$ com k constante.

Exercício 10 Considere a função `pot` que calcula a potência inteira de um número.

```
int pot (int x, int n) {
  int r = 1;

  if (n>0) {
    r = pot (x*x,n/2);
    if (m%2 == 1) r = r * x;
  }
  return r;
}
```

Assumindo que as operações elementares sobre inteiros (multiplicação, divisão e resto da divisão) de dois números executam em tempo linear no número de bits da representação binária dos números, faça a análise da complexidade (usando uma recorrência) desta função para o pior caso (não se esqueça de caracterizar esse pior caso), em função do número de bits usados na representação dos números inteiros.

8 Ordenação

Nesta secção vamos aplicar alguns dos conceitos e técnicas apresentadas atrás a vários algoritmos de ordenação de vectores.
Para simplificar a apresentação vamos apresentar funções de ordenação de vectores de inteiros e vamo-nos apenas preocupar com o número de comparações feitas (c) e com o número de escritas (w) no array.
Muitas vezes usaremos nos acessos ao array a função `swap` definida abaixo e que na nossa análise assumiremos que corresponde a duas escritas no array.

```
void swap (int a[], int p, int q){
  int t;
  t=a[p]; a[p]=a[q]; a[q]=t;
}
```

8.1 Bubble-sort

```
void bubble_sort (int N, int v[N]){
  int i, j;

  for(i=N-1;i>1;i--)
    for (j=0; j<i; j++)
      if (v[j]>v[j+1])
        swap (v,j,j+1);
}
```

A análise da definição acima mostra que:

- o número de iterações dos ciclos (e consequentemente de comparações entre elementos do array) depende **apenas** do tamanho do array.
- o número de vezes que a função `swap` será invocada depende também dos valores armazenados no array:
  - o melhor caso verifica-se quando a condição `(v[j]<v[j+1])` é sempre falsa. Ora isto corresponde ao array estar ordenado à partida.
  - o pior caso verifica-se quando a condição `(v[j]<v[j+1])` é sempre verdadeira. Isto verifica-se se o array estiver ordenado por ordem inversa à partida.

Destas observações resulta a seguinte análise.

Melhor caso

$$\begin{aligned} T_{\text{BubS}}(N) &= \sum_{i=2}^{N-1} \left(\sum_{j=0}^{i-1} c \right) \\ &= \sum_{i=2}^{N-1} i * c \\ &= c * \sum_{i=2}^{N-1} i \\ &= c * \tfrac{1}{2} * (2 + N - 1) * (N - 1 - 2 + 1) \\ &= c * \tfrac{1}{2} * (N + 1) * (N - 2) \\ &= \Theta(N^2) \end{aligned}$$

Pior caso

$$\begin{aligned} T_{\text{BubS}}(N) &= \sum_{i=2}^{N-1} \left(\sum_{j=0}^{i-1} c + 2 * w \right) \\ &= \sum_{i=2}^{N-1} i * (c + 2 * w) \\ &= (c + 2 * w) * \sum_{i=2}^{N-1} i \\ &= (c + 2 * w) * \tfrac{1}{2} * (2 + N - 1) * (N - 1 - 2 + 1) \\ &= (c + 2 * w) * \tfrac{1}{2} * (N + 1) * (N - 2) \\ &= \Theta(N^2) \end{aligned}$$

Como podemos ver o custo desta função, no pior e melhor casos, tem uma taxa de crescimento quadrática. Por isso é válido afirmar que, embora o melhor e o pior caso correspondam a custos diferentes,

$$T_{\text{BubS}}(N) = \Theta(N^2)$$

Exercício 11 É costume usar uma versão optimizada deste algoritmo que, para cada iteração do ciclo mais exterior verifica se o array já está ordenado.

```
void bubble_sort (int N, int v[N]){
  int i, j;
  int sorted=0;

  for(i=N-1;i>1 && !sorted;i--){
    sorted = 1;
    for (j=0; j<i; j++)
      if (v[j]<v[j+1]){
        swap (v,j,j+1);
        sorted = 0;
      }
    }
}
```

Identifique o melhor caso da execução desta função e determine o seu comportamento assimptótico nesse caso.

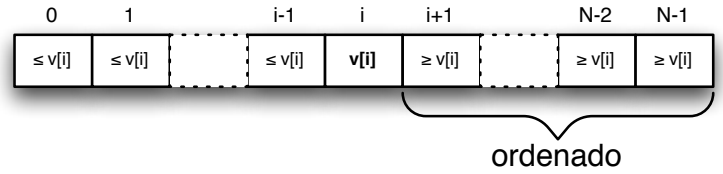
A compreensão de um algoritmo é fundamental para a análise de complexidade. Em algoritmos iterativos esta compreensão passa essencialmente pela identificação dos invariantes dos ciclos.
Vamos terminar esta breve apresentação deste algoritmo de ordenação pela apresentação informal dos dois ciclos envolvidos.

- Cada iteração do ciclo mais exterior tem como objectivo calcular o maior elemento do array e colocá-lo na última posição. Esta propriedade pode ser expressa pelo seguinte predicado:

  $$(\forall_{0 \leq k < i} \forall_{i < p < N}\ v[k] \leq v[p]) \ \wedge \ (\forall_{i \leq k < N-1}\ v[k] \leq v[k+1])$$

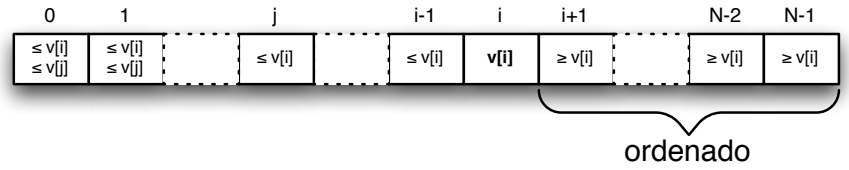
  O primeiro predicado da conjunção garante que os elementos à direita da posição `v[i]` são maiores (ou iguais) do que todos os elementos anteriores a `i`.

  O segundo garante que a partir da posição `i` o array está ordenado. Note-se que isto garante que a ordenação do array já não vai modificar a parte do array acima de `i`.



- Cada iteração do ciclo mais exterior tem como objectivo garantir que na posição `j` está o maior dos elementos desde a posição `0` até `j`.

$$(\forall_{0\leq k<i}\forall_{i<p<N}\ v[k]\leq v[p])\ \wedge\ (\forall_{i\leq k<N-1}\ v[k]\leq v[k+1])\ \wedge\ (\forall_{0\leq k<j}\ v[k]\leq v[j])$$



8.2 Ordenação por troca directa

```
void swap_sort (int N, int v[N]){
  int i, j;
  for (i=0;i<N-1;i++)
    for (j=i+1;j<N;j++)
      if (v[i]>v[j])
        swap (v,i,j);
}
```

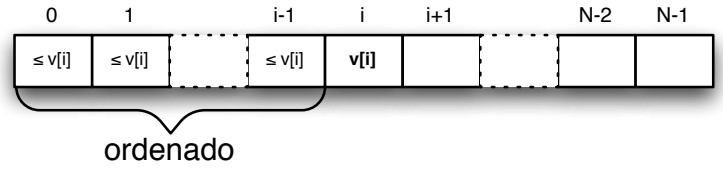
Os invariantes associados aos ciclos deste algoritmo são semelhantes aos apresentados para o *bubble-sort*:

- Cada iteração do ciclo mais exterior tem como objectivo calcular o menor elemento do array de `i` a `N-1` e colocá-lo na posição `i`. Esta propriedade pode ser expressa pelo seguinte predicado:

$$(\forall_{0\leq k<i}\forall_{i\leq p<N}\ v[k]\leq v[p])\ \wedge\ (\forall_{0\leq k<i-1}\ v[k]\leq v[k+1])$$

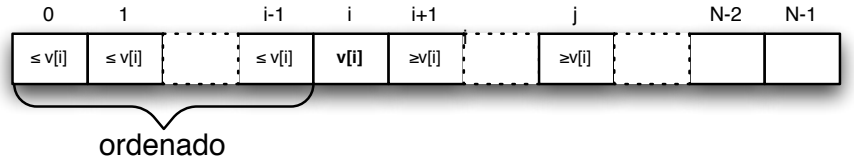
  O primeiro predicado da conjunção garante que os elementos à esquerda da posição `v[i]` são menores (ou iguais) do que todos os elementos à direita de `i`.

  O segundo garante que até à posição `i` o array está ordenado. Note-se que isto garante que a ordenação do array já não vai modificar a parte do array à esquerda de `i`.



- Cada iteração do ciclo mais exterior tem como objectivo garantir que na posição `i` está o menor dos elementos desde a posição `i` até `j`.

$$(\forall_{0\leq k<i}\forall_{i\leq p<N}\ v[k]\leq v[p])\ \wedge\ (\forall_{0\leq k<i-1}\ v[k]\leq v[k+1])\ \wedge\ (\forall_{i\leq k<j}\ v[i]\leq v[k])$$



A análise da definição acima mostra que:

- o número de iterações dos ciclos (e consequentemente de comparações entre elementos do array) depende **apenas** do tamanho do array.
- o número de vezes que a função `swap` será invocada depende também dos valores armazenados no array:
  - o melhor caso verifica-se quando a condição `(v[i]>v[j])` é sempre falsa. Ora isto corresponde ao array estar ordenado à partida (pois `j` é maior do que `i`).
  - o pior caso verifica-se quando a condição `(v[i]>v[j])` é sempre verdadeira. Isto verifica-se se o array estiver ordenado por ordem inversa à partida.

Destas observações resulta a seguinte análise.

Melhor caso

$$\begin{aligned} T_{\text{SwapS}}(N) &= \sum_{i=0}^{N-2}\left(\sum_{j=i+1}^{N-1} c\right) \\ &= \sum_{i=0}^{N-2}(N-1-i)*c \\ &= \left(\sum_{i=0}^{N-2}(N-1)*c\right) - c*\left(\sum_{i=0}^{N-2} i\right) \\ &= (N-1)*c*(N-1) - c*\frac{(N-1)*(N-2)}{2} \\ &= \frac{c}{2}*N*(N-1) \\ &= \Theta(N^2) \end{aligned}$$

Pior caso

$$\begin{aligned} T_{\text{SwapS}}(N) &= \sum_{i=0}^{N-2}\left(\sum_{j=i+1}^{N-1}(c+2*w)\right) \\ &= \frac{c+2*w}{2}*(N-2)*(N-3) \\ &= \Theta(N^2) \end{aligned}$$

Como podemos ver o custo desta função, no pior e melhor casos, tem uma taxa de crescimento quadrática. Por isso é válido afirmar que, embora o melhor e o pior caso correspondam a custos diferentes,

$$T_{\text{SwapS}}(N) = \Theta(N^2)$$

Exercício 12 Considere o seguinte procedimento de ordenação de arrays (ordenação por inserção ordenada).

```
void i_sort (int N, int v[N]){
  int i, j;
  int t;
  for (i=1; i<N; i++){
    t = a[i];
    for (j=i-1; j>=0 && a[j] > t; j--)
      a[j+1]=a[j];
    a[j+1] = t;
  }
```

Apresente informalmente os invariantes dos ciclos envolvidos. Identifique ainda o melhor e pior casos da execução deste procedimento em termos de número de comparações e escritas no array.

8.3 Merge-sort

O algoritmo de *merge-sort* é um exemplo de uma estratégia *divide and conquer*: dado um problema de dimensão N

1. começa-se por dividir problema em k problemas de dimensão substancialmente menor do que N
2. resolve-se cada um desses problemas (eventualmente usando essa mesma estratégia)
3. combinam-se as k soluções parciais de forma a obter a solução do problema original

A complexidade das soluções baseadas nesta estratégia é normalmente definida (rescursivamente) à custa da complexidade das operações de divisão do problema e de fusão das soluções parciais.

```
void merge_sort (int N, int v[N]){
  int m;
  if (N>1) {
    m = N/2;
    merge_sort (m, v);
    merge_sort (N-m, v+m);
    merge (N,v,m);
  }
}
```

A operação de divisão consiste apenas em seleccionar as duas metades do array em causa. Cada uma das invocações recursivas ordenará um array com (sensivelmente) metade dos elementos.
A operação fundamental deste algoritmo é por isso a fusão das soluções parciais. A função `merge` tem esse objectivo: construir um array ordenado a partir de um array em que os elementos nas posições `0..m` e nas posições `m+1..N-1` estão ordenadas.
Para fazer isso é necessário usar um array auxiliar onde se começa por construir a fusão dessas duas porções do array.

```
void merge (int N, int v[N], int m){
  int f, s, i;
  int aux[N];
  f=0; s=m; i=0;
  while ((f<m)&&(s<N))
```

```
    if (v[f]<v[s])
      aux[i++]=v[f++];
    else
      aux[i++]=v[s++];
  while (f<m)
    aux[i++]=v[f++];
  while (s<N)
    aux[i++]=v[s++];
  for (i=0;i<N;i++)
    v[i]=aux[i];
}
```

A complexidade desta função depende do tamanho do array em causa. Mais do que isso corresponde ao número de iterações dos vários ciclos. O número de iterações do último destes ciclos é `N`. Quanto aos três primeiros ciclos, apesar de não conseguirmos determinar o número de iterações de cada um, podemos afirmar que, no conjunto, correspondem também a `N` iterações[7].
Podemos por isso definir a complexidade desta função como

$$T_{\text{merge}}(N) = 2 * N = \Theta(N)$$

A complexidade da função de ordenação é definida pela seguinte relação de recorrência:

$$\begin{aligned} T_{\text{MSort}}(N) &= \begin{cases} 1 & \text{se } N \leq 1 \\ \\ T_{\text{merge}}(N) + 2 * T_{\text{MSort}}(\frac{N}{2}) & \text{se } N > 1 \end{cases} \\ &= \begin{cases} 1 & \text{se } N \leq 1 \\ \\ 2 * N + 2 * T_{\text{MSort}}(\frac{N}{2}) & \text{se } N > 1 \end{cases} \end{aligned}$$

[7]Note-se para isso que em cada iteração de cada um dos três ciclos a variável `i` é incrementada uma única vez e que no final dos três ciclos ela tem o valor `N`.

A solução desta recorrência pode ser fácilmente induzida a partir da expansão de alguns termos:

$$
\begin{aligned}
T_{\mathrm{MSort}}(0) &= 1 \\
T_{\mathrm{MSort}}(1) &= 1 \\
T_{\mathrm{MSort}}(2) &= T(2^1) \\
&= 2*2 + 2*T_{\mathrm{MSort}}(\tfrac{2^1}{2}) \\
&= 2*2+2 \\
T_{\mathrm{MSort}}(4) &= T(2^2) \\
&= 2*4 + 2*T_{\mathrm{MSort}}(\tfrac{2^2}{2}) \\
&= 2*4 + 2*(2*2+2) \\
&= 2*4 + 2*4 + 4 \\
T_{\mathrm{MSort}}(8) &= T(2^3) \\
&= 2*8 + 2*T_{\mathrm{MSort}}(\tfrac{2^3}{2}) \\
&= 2*8 + 2*(2*4 + 2*4 + 4) \\
&= 2*8 + 2*8 + 2*8 + 8 \\
T_{\mathrm{MSort}}(16) &= T(2^4) \\
&= 2*16 + 2*T_{\mathrm{MSort}}(\tfrac{2^4}{2}) \\
&= 2*16 + 2*(2*8 + 2*8 + 2*8 + 8) \\
&= 2*16 + 2*16 + 2*16 + 2*16 + 16 \\
&\cdots \\
T_{\mathrm{MSort}}(2^i) &= \underbrace{2*2^i + 2*2^i + \cdots + 2*2^i}_{i \text{ vezes}} + 2^i \\
&= i*2*2^i + 2^i \\
&= 2^i*(2*i+1)
\end{aligned}
$$

E por isso,

$$
\begin{aligned}
T_{\mathrm{MSort}}(N) &= T_{\mathrm{MSort}}(2^{\log_2(N)}) \\
&= 2^{\log_2(N)} * (2*(\log_2(N))) + 1) \\
&= N*(2*\log_2(N)+1 \\
&= \Theta(N*\log(N))
\end{aligned}
$$

Exercício 13 A seguinte alternativa para a função `merge` definida acima tenta minimizar as escritas no array argumento.

```
void merge (int N, int v[N], int m){
  int aux [N];
  int i0, i, s, k;

  for (i0=0; (i0<m) && (v[i0]<=v[m]); i0++);
```

```
  for (i=i0, s=m, k=i0; (i<m) && (s<N); k++)
    if (v[i]<=v[s]) aux[k]=v[i++];
      else          aux[k]=v[s++];
  for (;i<m;i++) aux[k++] = v[i];
  for (k=i0; k<s;k++)
    v[k] = aux[k];
}
```

Identifique o melhor caso em termos do número de escritas no array argumento (v). Para isso, comece por identificar os predicados que caracterizam o estado dos arrays envolvidos após cada um dos ciclos.

Exercício 14 A definição da função `merge` apresentada no exercício anterior depende, ao contrário da definição apresentada na página 27, do estado do array argumento.
Apresente uma definição alternativa que execute, no melhor caso, em tempo constante.
Usando essa definição, qual a complexidade da função `merge_sort` no melhor caso?

A inspecção do funcionamento da função `merge_sort` apresentada acima revela que as várias chamadas recursivas da função têm como único propósito a definição dos intervalos do array onde deve ser sucessivamente aplicada a função `merge`.
Vejamos por exemplo o que acontece quando invocamos esta função com um array com 8 elementos (`merge-sort (8,v)`).

1. `merge-sort (4,v)`
   (a) `merge-sort (2,v)`
      i. `merge-sort (1,v)`
      ii. `merge-sort (1,v+1)`
      iii. `merge (2,v,1)`
   (b) `merge-sort (2,v+2)`
      i. `merge-sort (1,v+2)`
      ii. `merge-sort (1,v+2+1)`
      iii. `merge (2,v+2,1)`
   (c) `merge (4,v,2)`
2. `merge-sort (4,v+4)`
   (a) `merge-sort (2,v+4)`
      i. `merge-sort (1,v+4)`
      ii. `merge-sort (1,v+4+1)`
      iii. `merge (2,v+4,1)`
   (b) `merge-sort (2,v+4+2)`
      i. `merge-sort (2,v+4+2)`
      ii. `merge-sort (1,v+4+2)`
      iii. `merge-sort (2,v+4+2+1,1)`

   (c) `merge (4,v+4+2,2)`

3. `merge (8,v,4)`

Podemos listar as chamadas à função `merge` obtendo a seguinte sequência:

1. `merge (2,v,1)`
2. `merge (2,v+2,1)`
3. `merge (2,v+4,1)`
4. `merge (2,v+4+2,1)`
5. `merge (4,v,2)`
6. `merge (4,v+4,2)`
7. `merge (8,v,4)`

Uma definição alternativa passa por pré-determinar esses intervalos de forma a começando com uma amplitude de 2, serem sucessivamente duplicados e acabarem por englobar todo o array.

```
void merge_sortI (int N, int v[N]) {
  int size, base, s;

  for (size = 2; size < 2*N; size*=2)
    for (base = 0; base < N; base+=size) {
      s = (size < N - base) ? size : N-base;
      merge(s,v+base,size/2);
  }
}
```

A análise da complexidade desta função, embora iterativa e definida à custa de `for...` não é tão imediata quanto os outros exemplos iterativos apresentados.
Isto acontece porque as variáveis de controlo dos ciclos envolvidos não são incrementadas por cada iteração do ciclo. Mas como são apenas alteradas uma vez por iteração podemos fazer as seguintes associações:

- O ciclo mais exterior podia ser antes controlado por uma variável `i`, com valor inicial 1, com valor final $\log_2(N)$ e incrementada uma vez por iteração.

  Note-se que dessa forma, a variável que realmente controla o ciclo (`size`) nada mais é do que 2^i.

- O ciclo mais interior podia ser antes controlado por uma variável `j`, com valor inicial 0, com valor final $\frac{\mathrm{N}}{\mathrm{size}}$ e incrementada uma única vez por iteração.

  Note-se que desta forma, a variável que realmente controla o ciclo (`base`) nada mais é do que $\mathrm{size} * j$.

Com estas observações podemos Definir a complexidade da função como

$$\begin{aligned}
T(N) &= \sum_{i=1}^{\log_2(N)} \sum_{j=0}^{\frac{N}{\text{size}}} T_{\text{merge}}(\text{size}) \\
&= \sum_{i=1}^{\log_2(N)} \sum_{j=0}^{\frac{N}{2^i}} T_{\text{merge}}(2^i) \\
&= \sum_{i=1}^{\log_2(N)} \sum_{j=0}^{\frac{N}{2^i}} (2^i) \\
&= \sum_{i=1}^{\log_2(N)} 2^i * (\tfrac{N}{2^i} + 1) \\
&= \sum_{i=1}^{\log_2(N)} (N + 2^i) \\
&= \left(\sum_{i=1}^{\log_2(N)} N\right) + \left(\sum_{i=1}^{\log_2(N)} 2^i\right) \\
&= (N * \log_2(N)) + \left(2 * 2^{\log_2(N)} - 2\right) \\
&= (N * \log_2(N)) + 2 * N - 2 \\
&= \Theta(N * \log_2(N))
\end{aligned}$$

8.4 Quick-Sort

O algoritmo de quick-sort (Hoare, 1969) segue uma estratégia de *divide and conquer*, começando por segmentar o array a ordenar em duas partes que serão ordenadas separadamente.
A peça chave deste algoritmo é esta operação de separação do array.

```
int partition (int N, int v[N]){
  int i, j;
  for (i=j=0; i<N-1; i++)
    if (v[i]<v[N-1]) swap (v,i,j++);
  swap (v,N-1,j);
  return j;
  }
```

O elemento que é usado para *partir* o array é muitas vezes referido como *pivot* e na versão aqui apresentada é o elemento que se encontra na última posição.
Se o resultado da invocação desta função retornar um valor p então verifica-se que

$$(\forall_{0<k<p} v[k] \leq v[p]) \ \wedge \ (\forall_{p<k<N} v[p] \leq v[k])$$

O número de comparações efectuadas entre elementos do array depende apenas do número de elementos do array: quando invocada com um array de tamanho N são feitas $N-1$ comparações.
Quanto ao número de escritas no array, podemos ver que é feito pelo menos uma troca (`swap`) fora do ciclo. Para um array com N elementos, o número total de trocas pode ser N se a condição `v[i]<v[N-1]` for sempre verdadeira, i.e., se o pivot escolhido for o maior elemento do array.

```
void quick_sort (int N, int v[N]){
  int p;
```

```
  if (N>1) {
    p = partition (N,v);
    quick_sort (v,p);
    quick_sort (v+p+1,N-p-1);
  }
}
```

A relação de recorrência que traduz o número de comparações entre elementos do array feito por esta função pode ser escrita como:

$$T(N) = \begin{cases} 1 & \text{se } N = 1 \\ \\ N - 1 + T(p) + T(N - 1 - p) \ \text{ com } 0 \leq p < N & \text{se } N > 1 \end{cases}$$

O caso extremo em que o valor de p é uma das extremidades do array (e que acontece por exemplo quando o array a ordenar já se encontra ordenado) resulta numa complexidade quadrática. De facto nesse caso a recorrência acima passa a ser escrita como

$$T(N) = \begin{cases} 1 & \text{se } N = 1 \\ \\ N - 1 + T(N - 1) & \text{se } N > 1 \end{cases}$$

que corresponde à função

$$T(N) = \sum_{i=1}^{N-1} i = \frac{N * (N - 1)}{2} = \Theta(N^2)$$

O outro caso extremo ocorre quando o valor de p corresponde ao ponto médio do array. Nesse caso a recorrência acima resulta em

$$T(N) = \begin{cases} 1 & \text{se } N = 1 \\ \\ N - 1 + 2 * T(\frac{N-1}{2}) & \text{se } N > 1 \end{cases}$$

A solução desta recorrência pode ser fácilmente induzida a partir da expansão de alguns termos:

$$
\begin{aligned}
T_{\mathrm{QSort}}(0) &= 1 \\
T_{\mathrm{QSort}}(1) &= 1 \\
T_{\mathrm{QSort}}(3) &= T_{\mathrm{QSort}}(2^2-1) \\
&= 2 + 2 * 1 \\
&= (2^2-2) + 2 \\
T_{\mathrm{QSort}}(7) &= T_{\mathrm{QSort}}(2^3-1) \\
&= 6 + 2 * ((2^2-2) + 2) \\
&= (2^3-2) + (2^3-2^2) + 2^2 \\
T_{\mathrm{QSort}}(15) &= T_{\mathrm{QSort}}(2^4-1) \\
&= 14 + 2 * ((2^3-2) + (2^3-2^2) + 2^2) \\
&= (2^4-2) + (2^4-2^2) + (2^4-2^3) + 2^3 \\
T_{\mathrm{QSort}}(31) &= T_{\mathrm{QSort}}(2^5-1) \\
&= 30 + 2 * ((2^4-2) + (2^4-2^2) + (2^4-2^3) + 2^3) \\
&= (2^5-2) + (2^5-2^2) + (2^5-2^3) + (2^5-2^4) + 2^4 \\
&\cdots \\
T_{\mathrm{QSort}}(2^k-1) &= (2^k-2) + (2^k-2^2) + \cdots (2^k-2^{k-1}) + 2^{k-1} \\
&= (k-1) * 2^k - \left(\textstyle\sum_{i=1}^{k-1} 2^i\right) + 2^{k-1} \\
&= (k-1) * 2^k - (2^k-2) + 2^{k-1} \\
&= 2^k * \left(k - \tfrac{3}{2}\right) + 2
\end{aligned}
$$

E por isso,

$$
\begin{aligned}
T_{\mathrm{QSort}}(N) &\sim T_{\mathrm{QSort}}(2^{\log_2(N)}-1) \\
&\sim 2^{\log_2(N)} * \left(\log_2(N) - \tfrac{3}{2}\right) + 2 \\
&\sim N * \left(\log_2(N) - \tfrac{3}{2}\right) + 2 \\
&\sim \Theta(N * \log(N))
\end{aligned}
$$

Para calcularmos o valor médio do número de comparações efectuado pelo algoritmo de quick-sort vamos assumir que a probabilidade de cada um dos valores possíveis da função de partição tem uma distribuição uniforme (i.e., se o vector tiver N elementos, a probabilidade de cada um dos N valores possíveis é $\frac{1}{N}$). Da recorrência acima obtemos

$$
\overline{T}(N) = (N-1) + \sum_{p=0}^{N-1} \frac{1}{N} \left(\overline{T}(p) + \overline{T}(N-p-1)\right) \quad \text{para } N > 1
$$

Esta expressão pode ser simplificada, se expandirmos o somatório:

$$\begin{aligned}
&\textstyle\sum_{p=0}^{N-1} \frac{1}{N} \left(\overline{T}(p) + \overline{T}(N-p-1)\right) \\
&\quad = \tfrac{1}{N} \left((\overline{T}(0) + \overline{T}(N-1)) + (\overline{T}(1) + \overline{T}(N-2)) + \cdots + (\overline{T}(N-1) + \overline{T}(0))\right) \\
&\quad = \tfrac{1}{N} \left((\overline{T}(0) + \overline{T}(0)) + (\overline{T}(1) + \overline{T}(1)) + \cdots + (\overline{T}(N-1) + \overline{T}(N-1))\right) \\
&\quad = \tfrac{2}{N} \left(\overline{T}(0) + \overline{T}(1) + \cdots + \overline{T}(N-1)\right) \\
&\quad = \tfrac{2}{N} \textstyle\sum_{p=0}^{N-1} \overline{T}(p)
\end{aligned}$$

Temos então que

$$\overline{T}(N) = (N-1) + \frac{2}{N} \sum_{p=0}^{N-1} \overline{T}(p)$$

Multiplicando ambos os membros por N,

$$N * F(N) = N * (N-1) + 2 * \sum_{p=1}^{N-1} \overline{T}(p)$$

Calculando esta expressão para $N-1$

$$(N-1) * \overline{T}(N-1) = (N-1) * (N-2) + 2 * \sum_{p=1}^{N-2} \overline{T}(p)$$

Subtraindo estas equações (membro a membro), obtemos

$$N * F(N) - (N-1) * \overline{T}(N-1) = N * (N-1) - (N-1) * (N-2) + 2 * \overline{T}(N-1)$$

O que, depois de simplificar, vem

$$\overline{T}(N) = \left(\frac{2 * N - 1}{N}\right) + \left(\frac{N+1}{N}\right) * \overline{T}(N-1)$$

Que já está na forma canónica de uma equação de recorrência de 1ª ordem, e por isso

$$\begin{aligned}
\overline{T}(N) &= \tfrac{2}{1} * \tfrac{3}{2} * \tfrac{4}{3} * \cdots * \tfrac{N+1}{N} * \left(1 + \textstyle\sum_{i=1}^{N} \frac{\frac{2*i-1}{i}}{\frac{2}{1} * \frac{3}{2} * \frac{4}{3} * \cdots * \frac{i+1}{i}}\right) \\
&= (N+1) * \left(1 + \textstyle\sum_{i=1}^{N} \frac{\frac{2*i-1}{i}}{i+1}\right) \\
&= (N+1) * \left(1 + \textstyle\sum_{i=1}^{N} \frac{2}{i+1} + \sum_{i=1}^{N} \frac{1}{i*(i+1)}\right) \\
&= \Theta(N) * (\Theta(1) + \Theta(\log(N)) + \Theta(1)) \\
&= \Theta(N * \log(N))
\end{aligned}$$

Exercício 15 A função de ordenação `quick_sort` apresentada acima pode ser ligeiramente melhorada no caso de o array a ordenar ter elementos repetidos. Nesse sentido,

1. Comece por definir uma variante da função `partition` que secciona o array em três partes ((1) os elementos que são estritamente menores do que o pivot, (2) os elementos que são iguais ao pivot e (3) os elementos que são estritamente maiores do que o pivot). Certifique-se que a sua função executa em tempo linear no tamanho do array argumento,

2. Altere a definição da função de ordenação de forma a usar essa nova definição da função de partição.

Do mesmo autor do algoritmo de *quick sort* (Tony Hoare) existe um algoritmo de selecção do k-ésimo elemento de um array e que pode ser definido da seguinte forma:

```
int quickSelect (int k, int N, int v[N]){
  int r;
  if (N==1) r=v[0];
  else {
    p = partition (N,v);
    if (p==k) r=v[p];
    else if (p>k) r=quickSelect(k,p-1,v);
    else r=quickSelect(k-p-1, N-p-1, v+p+1);
  }
  return r;
}
```

A relação de recorrência que traduz o esforço envolvido nesta definição é dado por

$$T(N) = \begin{cases} 1 & \text{se } N = 1 \\ \\ N + T(p) \;\; \text{com } 1 \leq p < N & \text{se } N > 1 \end{cases}$$

Em que a quantidade p corresponde ao tamanho do array sobre o qual é feita a chamada recursiva.
No melhor caso, que acontece quando o valor retornado pela função `partition` corresponde ao índice `k`, esta função executa em tempo linear (o tempo de executar a função `partition`).
O pior caso acontece quando o tamanho do array que é passado na chamada recursiva difere apenas de uma unidade relativamente ao array original. Neste caso esta função tem uma complexidade quadrática.
Assumindo que todas os possíveis resultados de p são igualmente prováveis, o tempo médio será dado por:

$$\overline{T}(x) = \begin{cases} 1 & \text{se } x = 1 \\ \\ (x-1) + \frac{1}{x-1} \sum_{p=1}^{x-1} \overline{T}(p) & \text{se } x > 1 \end{cases}$$

O que, particularizando para $x = N$ e $x = N - 1$ vem

$$\begin{aligned} \overline{T}(N) &= (N-1) + \tfrac{1}{N-1} \textstyle\sum_{p=1}^{N-1} \overline{T}(p) \\ (N-1) * \overline{T}(N) &= (N-1)^2 + \textstyle\sum_{p=1}^{N-1} \overline{T}(p) \\ (N-1) * \overline{T}(N) &= (N-1)^2 + \overline{T}(1) + \overline{T}(2) + \cdots + \overline{T}(N-1) \end{aligned}$$

$$\begin{aligned} \overline{T}(N-1) &= (N-2) + \tfrac{1}{N-2} \textstyle\sum_{p=1}^{N-2} \overline{T}(p) \\ (N-2) * \overline{T}(N-1) &= (N-2)^2 + \textstyle\sum_{p=1}^{N-2} \overline{T}(p) \\ (N-2) * \overline{T}(N-1) &= (N-2)^2 + \overline{T}(1) + \overline{T}(2) + \cdots + \overline{T}(N-2) \end{aligned}$$

Subtraindo as duas igualdades

$$\begin{aligned}(N-1)*\overline{T}(N)-(N-2)*\overline{T}(N-1) &= (N-1)^2-(N-2)^2+\overline{T}(N-1)\\ (N-1)*\overline{T}(N) &= (N-1)^2-(N-2)^2+(N-1)*\overline{T}(N-1)\end{aligned}$$

$$\begin{aligned}\overline{T}(N) &= \tfrac{(N-1)^2-(N-2)^2}{N-1}+\overline{T}(N-1)\\ &= \tfrac{2*N-3}{N-1}+\overline{T}(N-1)\\ &= 2-\tfrac{1}{N-1}+\overline{T}(N-1)\\ &= \underbrace{2+2+\cdots+2}_{N \text{ vezes}}-\textstyle\sum_{i=1}^{N-1}\frac{1}{i}\\ &= 2*N-\textstyle\sum_{i=1}^{N-1}\frac{1}{i}\\ &= \Theta(N)-\Theta(\log(N))\\ &= \Theta(N)\end{aligned}$$

9 Análise amortizada

Relembre a função `inc` que, recebe como argumento um vector com os *bits* de um número, e altera esse vector de forma a representar o sucessor do número original.
Assim, por exemplo, para o vector representar o número 119 (cuja representação binária é `1110111`) deve ter todos os elementos a zero excepto os 7 últimos, que devem guardar a sequência `1 1 1 0 1 1 1`. A invocação da função `inc` com esse vector, deverá alterar apenas as 4 últimas posições do vector de forma a representar o número 120 (cuja representação binária é `1111000`).

```
int inc (int b[], int N) {
int i, r=0;

    i = N-1;
    while ((i >= 0) && (b[i] == 1)) {
        b[i] = 0;
        i--;
    }
    if (i>=0) b[i] = 1;
    else r = 0;

    return r;
}
```

A complexidade desta função pode ser reduzida ao número de bits alterados, que varia entre

- 1 no (**melhor**) caso de o bit menos significativo ser `0`
- N no (**pior**) caso de os últimos $N-1$ bits serem `1`.

A análise do caso médio já apresentada revela-nos que

$$\overline{T}(N)=(\sum_{k=1}^{N}\frac{k}{2^k})+\frac{N}{2^N}=2-\frac{1}{2^{N-1}}$$

É de notar que $\lim \overline{T}(N) = 2$ e que por isso

$$\overline{T}(N) = \Theta(1)$$

A razão de ser desta proximidade do caso médio e do melhor caso, ou por oposição, da diferença tão acentuada entre o caso médio e o pior caso, deve-se à pouca probabilidade de o pior caso acontecer.

Esta observação motiva o uso de uma outra forma de analisar a complexidade de algoritmos conhecida genericamente por **análise amortizada** e que consiste em, ao invés de analisar o custo de uma operação, analisar o custo de uma sequência de operações.
É costume utilizar como sequência de operações a analisar, a sequência de pior custo.

Dada então uma sequência de N operações $\{op_i\}_{1 \leq i \leq N}$ cada uma com um custo c_i, pretendemos determinar o custo amortizado da operação i, denotado por $\widehat{c}_i$ de tal forma que a soma dos custos amortizados seja um limite superior da soma dos custos reais, i.e.,

$$\sum_{i=1}^{N} \widehat{c}_i \geq \sum_{i=1}^{N} c_i$$

9.1 Análise Agregada

Nesta abordagem à análise amortizada, aquilo que se faz é começar por calcular a soma dos custos reais das operações da sequência a analisar. Com este valor calculado, podemos estimar o custo amortizado de cada operação da sequência como

$$\widehat{c}_i = \frac{1}{N} \sum_{i=1}^{N} c_i$$

Com esta definição, a propriedade acima é trivialmente verificada:

$$\begin{array}{ll} \sum_{i=1}^{N} \widehat{c}_i & = \sum_{i=1}^{N} (\frac{1}{N} \sum_{j=1}^{N} c_j) \\ & = (\frac{1}{N} \sum_{j=1}^{N} c_j) \sum_{i=1}^{N} 1 \\ & = (\frac{1}{N} \sum_{j=1}^{N} c_j) N \\ & = \sum_{j=1}^{N} c_j \end{array}$$

Vejamos então como proceder para o caso que estamos a analisar (a função `inc` apresentada acima). Consideremos então uma sequência de N invocações desta função a um mesmo vector `b` com um número suficientemente grande de posições e que está inicialmente todo preenchido a `0` (veremos que esta assumpção dos valores iniciais não é realmente importante na conclusão final).
A tabela sequinte (que já foi apresentada acima), mostra o custo das várias operações desta sequência.

i	input	output	c_i
1	$\cdots$ 0 0 0 0 0 0 0	$\cdots$ 0 0 0 0 0 0 0 1	1 = 1
2	$\cdots$ 0 0 0 0 0 0 1	$\cdots$ 0 0 0 0 0 0 1 0	2 = 1+1
3	$\cdots$ 0 0 0 0 0 1 0	$\cdots$ 0 0 0 0 0 1 1	1 = 1
4	$\cdots$ 0 0 0 0 0 1 1	$\cdots$ 0 0 0 0 1 0 0	3 = 1+1+1
5	$\cdots$ 0 0 0 0 1 0 0	$\cdots$ 0 0 0 0 1 0 1	1 = 1
6	$\cdots$ 0 0 0 0 1 0 1	$\cdots$ 0 0 0 0 1 1 0	2 = 1+1
7	$\cdots$ 0 0 0 0 1 1 0	$\cdots$ 0 0 0 0 1 1 1	1 = 1
8	$\cdots$ 0 0 0 0 1 1 1	$\cdots$ 0 0 0 1 0 0 0	4 = 1+1+1+1
...			

O que pretendemos calcular é a soma dos elementos da última coluna $\sum c_i$ para depois calcular a sua média.
Neste caso, e como já indicado na coluna, podemos ver que:

- todas as linhas têm a parcela 1
- metade das linhas tem também a parcela +1
- um quarto das linhas têm também mais uma parcela +1
- ...

Generalizando, a soma dos valores da última coluna pode ser calculada por:

$$N + \frac{N}{2} + \frac{N}{4} + ... = \sum_{i=0}^{?} \frac{N}{2^i}$$

O limite superior deste somatório corresponde ao número de vezes que conseguimos dividir n por 2, i.e., $\log_2 N$ (em rigor, este limite superior corresponde ao maior inteiro que seja menor ou igual a $\log_2 N$, i.e., $\lfloor \log_2 N \rfloor$).
Temos então o custo C de fazer N incrementos (sem overflow)

$$C_N = \sum_{i=0}^{\log_2 N} \frac{N}{2^i} = N \sum_{i=0}^{\log_2 N} \frac{1}{2^i} = N\,(2 - \frac{1}{2^{\log_2 N}}) = N\,(2 - \frac{1}{N}) = 2\,N - 1$$

Para sabermos agora o custo (amortizado) de cada operação, apenas temos que dividir este custo pelo número de operações em causa:

$$\widehat{c_i} = \frac{C_N}{N} = \frac{1}{N}(2\,N - 1) = 2 - \frac{1}{N} = \Theta(1)$$

É de notar que, apesar do custo médio aqui calculado ser diferente do que calculámos acima, em termos assimptóticos, obtivemos exactamente o mesmo resultado. A diferença deve-se ao factor excepcional do caso de overflow.
É também por nos focarmos na análise assimptótica que a assumpção de partirmos de um vector inicializado a `0` não é relevante. Se partirmos de um outro valor inicial, obteremos um valor diferente mas assimptoticamente igual.

9.2 Método Contabilístico

Muitas vezes não é fácil fazer o cálculo agregado dos custos de uma sequência de operações. Existem por isso outros métodos que ajudam, se não a calcular essa soma, pelo menos a determinar um limite superior razoável.
No método contabilístico da análise amortizada aquilo que vamos tentar é estimar um custo $\widehat{c_i}$ para a operação i de tal forma que a soma dos custos das operações da sequência seja menor do que a soma destes custos esperados, i.e., devemos estimar o custo $\widehat{c_i}$ de tal forma que

$$\sum_{i=1}^{N} c_i \leq \sum_{i=1}^{N} \widehat{c_i}$$

Esta desigualdade pode ser escrita sob a forma

$$(\sum_{i=1}^{N} \widehat{c_i}) - (\sum_{i=1}^{N} c_i) \geq 0$$

Uma forma (mas não a única) de garantir que esta inequação é válida, é garantirmos que, para todo o K,

$$B_K \doteq (\sum_{i=1}^{K} \widehat{c_i}) - (\sum_{i=1}^{K} c_i) \geq 0$$

A analogia que vamos usar é a de uma conta bancária em que se pretende garantir um saldo positivo:

- o termo $D_K \doteq \sum_{i=1}^{K} \widehat{c_i}$ corresponde à soma de todos os *depósitos* (até à operação K).
- o termo $W_K \doteq \sum_{i=1}^{K} c_i$ corresponde à soma de todos os *levantamentos* (até à operação K).

Visto desta perspectiva, podemos definir B_k como o *saldo* após a operação k como

$$\begin{aligned} B_k &= D_k - W_k \\ &= (\textstyle\sum_{i=0}^{k} \widehat{c_i}) - (\sum_{i=0}^{k} c_i) \\ &= (\widehat{c_k} + \textstyle\sum_{i=0}^{k-1} \widehat{c_i}) - (c_k + \sum_{i=0}^{k-1} c) \\ &= (\textstyle\sum_{i=0}^{k-1} \widehat{c_i} - \sum_{i=0}^{k-1} c_i) + (\widehat{c_k} - c_k) \\ &= B_{k-1} + (\widehat{c_k} - c_k) \end{aligned}$$

O que nos dá uma forma de calcular o *saldo* após uma operação a partir do *saldo* antes e dos custos reais e estimados dessa operação.
Por razões de completude (e uma vez que a fórmula que obtivemos é uma recorrência) devemos indicar o valor inicial B_0. É costume usar 0 para este valor, embora não seja necessário para uma análise assimptótica (este valor inicial corresponde à assumpção feita na análise agregada de começar a sequência a analisar com um determinado valor).

Vejamos então como este método pode ser aplicado ao problema que já apresentámos (incremento).
A tabela apresentada atrás contem já a coluna correspondente aos custos reais (levantamentos). Na tabela seguinte vamos acrescentar duas novas colunas que correspondem aos custos estimados e ao saldo.
Para isso vamos estimar que o custo de cada operação é constante, i.e., $\widehat{c_i} = 2$ (nesta altura da exposição do problema já não há muitas surpresas sobre qual o custo amortizado de cada operação!).

Esta estimativa deve ser feita de forma a prever futuros custos adicionais das próximas operações. Neste caso, cada operação de `inc` transforma uma sequência de `1`s em `0`s e finalmente transforma um `0` em `1`. Se admitirmos que o custo de passar os `1`s a `0` se faz à custa do valor *amealhado*, devemos amealhar 1, para mais tarde o podermos usar para converter esse `1` (que acabou de ser produzido).

$\mathbf{i}$	input								$\mathbf{c_i}$	$\widehat{\mathbf{c}}_\mathbf{i}$	$\mathbf{B_i}$
0											0
1	$\cdots$	0	0	0	0	0	0	0	1	2	1
2	$\cdots$	0	0	0	0	0	0	1	2	2	1
3	$\cdots$	0	0	0	0	0	1	0	1	2	2
4	$\cdots$	0	0	0	0	0	1	1	3	2	1
5	$\cdots$	0	0	0	0	1	0	0	1	2	2
6	$\cdots$	0	0	0	0	1	0	1	2	2	2
7	$\cdots$	0	0	0	0	1	1	0	1	2	3
8	$\cdots$	0	0	0	0	1	1	1	4	2	1
9	$\cdots$	0	0	0	1	0	0	0	1	2	2
10	$\cdots$	0	0	0	1	0	0	1	2	2	2
11	$\cdots$	0	0	0	1	0	1	0	1	2	3
12	$\cdots$	0	0	0	1	0	1	1	3	2	2
13	$\cdots$	0	0	0	1	1	0	0	1	2	3
14	$\cdots$	0	0	0	1	1	0	1	2	2	3
15	$\cdots$	0	0	0	1	1	1	0	1	2	4
16	$\cdots$	0	0	0	1	1	1	1	5	2	1
17	$\cdots$	0	0	1	0	0	0	0	1	2	2

Esta tabela mostra-nos que o valor estimado ($\widehat{c}_i = 2$) para cada operação é suficiente para garantir que o saldo nunca toma valores negativos. De facto, e tal como vimos atrás, em metade dos casos o custo real é 1, permitindo-nos *poupar* 1 por cada uma dessas operações (continuando na analogia contabilística, um *depósito* de 2 e um *levantamento* de 1 permite-nos *poupar* 1).

9.3 Método do Potencial

O método de análise amortizada visto na secção anterior baseia-se na estimativa de um custo de cada operação de tal forma que o somatório dos custos estimados seja superior à soma dos custos reais dessas operações.
No método do potencial vamos tentar calcular estes custos estimados a partir de uma função (dita de potencial) sobre a estrutura de dados (estado) em questão.

Suponhamos então que definimos uma função Φ que, mapeia cada estado (no caso que temos vindo a analisar, um vector de `0`s e `1`s) num número (real) e que satisfaz as seguintes propriedades:

- $\Phi(S) \geq 0$ para todos os possíveis estados S

- $\Phi(S_0) = 0$ em que S_0 corresponde ao estado inicial.

Usando esta função de potencial, vamos definir o custo estimado de cada operação da sequência $(\widehat{c_i})$ como

$$\widehat{c_i} = c_i + (\Phi(S_i) - \Phi(S_{i-1}))$$

em que c_i corresponde ao custo real da operação i e S_{i-1} e S_i correspondem aos estados antes e depois de executar a operação i.

Por simplicidade de notação, vamos passar a denotar $\Phi(S_i)$ apenas por Φ_i

Vejamos agora como estas definições nos permitem obter um majorante para o custo de N operações.

$$\begin{aligned}
\sum \widehat{c_i} &= \widehat{c_1} + \widehat{c_2} + \cdots + \widehat{c_{N-1}} + \widehat{c_N} \\
&= (c_1 + \Phi_1 - \Phi_0) + (c_2 + \Phi_2 - \Phi_1) + \cdots + (c_N + \Phi_N - \Phi_{N-1}) \\
&= c_1 + \Phi_1 - \Phi_0 + c_2 + \Phi_2 - \Phi_1 + \cdots + c_N + \Phi_N - \Phi_{N-1} \\
&= c_1 + c_2 + \cdots + c_N + \Phi_N - \Phi_0 \\
&= (\sum c_i) + \Phi_N - \Phi_0
\end{aligned}$$

Da igualdade obtida

$$\sum_{i=1}^{N} \widehat{c_i} = (\sum_{i=1}^{N} c_i) + \Phi_N - \Phi_0$$

e das propriedades da função de potencial acima enumeradas, podemos concluir que

$$\sum_{i=1}^{N} \widehat{c_i} - (\sum_{i=1}^{N} c_i) = \Phi_N - \Phi_0 = \Phi_N - 0 \geq 0$$

E por isso, que a soma dos custos estimados $(\sum \widehat{c_i})$ é maior ou igual à soma dos custos reais $(\sum c_i)$.
(É de notar que uma formulação alternativa das propriedades da função de potencial poderia ser que o potencial de cada estado é sempre superior ao potencial do estado inicial.)

A definição da função de potencial nem sempre é muito fácil ou intuitiva. Esta função deve caracterizar o esforço envolvido em processar um dado estado. Daí que muitas vezes se associe o potencial de um estado com o seu estado de desordem.
No caso concreto que temos vindo a analisar, e depois de todo este foco nesse exemplo, a solução surge mais naturalmente.
Vamos definir o potencial de um array de bits como o número de `1`s desse array.
Note-se que esta função é independente da função `inc` que estamos a analisar. Depende apenas do valor do seu argumento (estado).
Definida esta função, calculemos o valor esperado do custo de cada operação. Este é dado, como foi dito acima, pela fórmula

$$\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1}$$

É relativamente fácil, neste contexto, caracterizar o custo real de cada operação. De facto, o que a função `inc` está a fazer é a converter em `0`s o maior prefixo de `1`s do vector, seguido de converter um bit adicional.

- Assumindo que o array b tem pelo menos um bit a `0`, seja `k` a posição do `0` mais à direita. Então, $\Phi(b) = l + r$ em que l corresponde à soma do número de `1`s até à posição `k` e r corresponde ao número de `1`s da posição `k` em diante (que não é mais do que o comprimento do maior prefixo de `1`s de b).

- No estado após a execução de `inc`, o potencial passa a ser $\Phi(b') = l + 1$ uma vez que os r bits a `1` foram substituidos por `0`s.

O custo estimado de cada operação é então

$$\widehat{c_i} = (l+1) + (r+1) - (l+r) = 2$$

Obtendo (sem grande surpresa!) o mesmo resultado que obtivemos pela aplicação dos outros métodos.

9.4 Tabelas dinâmicas

Considere-se uma estrutura de dados implementada num array em que o custo de fazer uma inserção é constante. Por simplicidade da apresentação, admitamos que esse custo é 1.
Vamos enriquecer essa implementação, permitindo que, quando queremos inserir um novo elmento e o array esgota a sua capacidade, (1) é realocado um novo array com o dobro da capacidade, (2) é copiado o conteúdo do array antigo para este novo array e, finalmente (3) é adicionado o novo elemento.
A análise do custo de cada inserção mostra-nos que

- no melhor caso (o array ainda não esgotou a sua capacidade) o custo da inserção é igual ao custo anterior, i.e., 1.
- no pior caso (a capacidade do array já foi atingida), temos um custo adicional de N, que corresponde à cópia dos elementos para o novo array.

Vejamos então a análise amortizada desta nova operação de inserção.

9.4.1 Análise Agregada

A tabela seguinte mostra o estado da tabela após a inserção das primeiras 12 entradas numa tabela inicialmente vazia (e com um tamanho inicial de 1), bem como os custos das várias inserções.

i	**Resultado**	**c_i**
0	[ ]	
1	[1]	1
2	[1][2]	2 = 1+1
3	[1][2][3][ ]	3 = 1+2
4	[1][2][3][4]	1
5	[1][2][3][4][5][ ][ ][ ]	5 = 1+4
6	[1][2][3][4][5][6][ ][ ]	1
7	[1][2][3][4][5][6][7][ ]	1
8	[1][2][3][4][5][6][7][8]	1
9	[1][2][3][4][5][6][7][8][9][ ][ ][ ][ ][ ][ ][ ]	9 = 1+8
10	[1][2][3][4][5][6][7][8][9][10][ ][ ][ ][ ][ ][ ]	1
11	[1][2][3][4][5][6][7][8][9][10][11][ ][ ][ ][ ][ ]	1
12	[1][2][3][4][5][6][7][8][9][10][11][12][ ][ ][ ][ ]	1

A inspecção dos vários custos associados a cada uma das operações põe em evidência que o somatório dos custos de uma sequência de N inserções pode ser calculado como

$$C_N = N + \sum_{i=0}^{?} 2^i$$

Mais uma vez, o limite superior deste somatório corresponde ao número de vezes que conseguimos dividir N por 2 (sem chegar a 0), i.e., $\log_2 N$.
Desenvolvendo,

$$C_N = N + \sum_{i=0}^{\log_2 N} 2^i = N + (2^{1+\log_2 N} - 1) = N + 2\,N - 1 = 3\,N - 1$$

Do que resulta um custo amortizado $\widehat{c}_i$ de

$$\widehat{c}_i = \frac{1}{N}\,C_N = \frac{1}{N}\,(3\,N - 1) = 3 - \frac{1}{N} = \Theta(1)$$

9.4.2 Método Contabilístico

Admitindo que o saldo mínimo é atingido depois de uma duplicação da tabela, cada nova inserção deverá ter como custo amortizado, para além do custo unitário de uma inserção, um custo suplementar de 2, que será usado para a próxima duplicação da tabela. Isto porque se a tabela passar a ter um tamanho X, a nova duplicação acontecerá após $X/2$ inserções. Por isso cada inserção deverá *amealhar* 2.
Vejamos então a tabela da análise anterior, aumentada com o custo estimado $\widehat{c}_i = 3$ e o saldo no fim de cada operação.

i	**Resultado**	**c_i**	**$\widehat{c}_i$**	**B_i**
0	[ ]			0
1	[1]	1	3	2
2	[1 2]	2	3	3
3	[1 2 3]	3	3	3
4	[1 2 3 4]	1	3	5
5	[1 2 3 4 5]	5	3	3
6	[1 2 3 4 5 6]	1	3	5
7	[1 2 3 4 5 6 7]	1	3	7
8	[1 2 3 4 5 6 7 8]	1	3	9
9	[1 2 3 4 5 6 7 8 9]	9	3	3
10	[1 2 3 4 5 6 7 8 9 10]	1	3	5
11	[1 2 3 4 5 6 7 8 9 10 11]	1	3	7
12	[1 2 3 4 5 6 7 8 9 10 11 12]	1	3	9

9.4.3 Método do Potencial

De forma a definirmos o valor do potencial de cada estado, devemos atentar no seguinte:

- O potencial deverá aumentar à medida que o número de elementos da tabela aumenta.
- Após uma operação que envolva a duplicação da tabela, o potencial deve diminuir proporcionalmente ao número de elementos na tabela (de forma a absorver o custo adicional de cópia dos elementos).

Uma forma de obedecer a estes dois requisitos será definir o potencial de um estado i como

$$\Phi_i = U_i - F_i = 2 * U_i - S_i$$

em que S_i, U_i e F_i correspondem ao tamanho do array, ao número de posições ocupadas e número de posições livres respectivamente.
De forma a garantir uma das propriedades da função de potencial, diremos ainda que $\Phi_0 = 0$.
A segunda propriedade, que o potencial nunca se torna negativo, é um invariante da estrutura em causa: quando se duplica o tamanho, pelo menos metade estará ocupada.
Vejamos então qual o valor do custo estimado de cada inserção. Para isso analisaremos os dois comportamentos alternativos da função.

- No caso de **não se fazer a duplicação** da tabela: o custo real da operação é $c_i = 1$; o tamanho do array permanece inalterado ($S_i = S_{i-1}$); o número de posições livres diminui uma unidade ($F_i = F_{i-1} - 1$); o número de posições ocupadas aumenta uma unidade ($U_i = U_{i-1} + 1$).

$$\begin{aligned}\widehat{c_i} &= c_i + (\Phi_i - \Phi_{i-1}) \\ &= 1 + (U_i - F_i) - (U_{i-1} - F_{i-1}) \\ &= 1 + (U_{i-1} + 1) - (F_{i-1} - 1) - (U_{i-1} - F_{i-1}) \\ &= 1 + U_{i-1} + 1 - F_{i-1} + 1 - U_{i-1} + F_{i-1} \\ &= 3\end{aligned}$$

- No caso de **se fazer a duplicação** da tabela: o custo real da operação é $c_i = 1 + U_{i-1}$ (note-se que U_k corresponde ao número de posições usadas após a operação k); o tamanho do array aumenta para o dobro ($S_i = 2 * S_{i-1}$); o número de células usadas aumenta uma unidade ($U_i = U_{i-1} + 1$); o número de células livres aumenta de $F_{i-1} = 0$ para $F_i = U_{i-1} - 1$.

$$\begin{aligned}\widehat{c_i} &= c_i + (\Phi_i - \Phi_{i-1}) \\ &= 1 + U_{i-1} + (U_i - F_i) - (U_{i-1} - F_{i-1}) \\ &= 1 + U_{i-1} + (U_{i-1} + 1 - (U_{i-1} - 1)) - (U_{i-1} - 0) \\ &= 1 + U_{i-1} + U_{i-1} + 1 - U_{i-1} + 1 - U_{i-1} + 0 \\ &= 1 + U_{i-1} + U_{i-1} + 1 - U_{i-1} + 1 - U_{i-1} + 0 \\ &= 3\end{aligned}$$

Vemos então que, para ambos os casos, $\widehat{c_i} = 3$.

9.4.4 Remoção

Consideremos agora que adicionalmente à operação de inserção definida atrás, pretendemos implementar um procedimento de remoção que, no caso da tabela atingir um determinado valor mínimo da sua taxa de ocupação, ela é realocada para uma tabela menor.
Vamos realocar a tabela para metade do seu valor apenas quando a tabela tiver apenas um quarto das suas entradas preenchidas. A tabela seguinte parte duma tabela em que foram feitas 9 inserções e onde se vão sucessivamente removendo elementos.

i	Resultado	c_i
0	1 2 3 4 5 6 7 8 9	
1	1 2 3 4 5 6 7 8	1
2	1 2 3 4 5 6 7	1
3	1 2 3 4 5 6	1
4	1 2 3 4 5	1
5	1 2 3 4	5
6	1 2 3	1
7	1 2	3
8	1	1

No modelo contabilístico podemos associar a cada remoção um custo estimado de 2, correspondendo ao custo real 1 acrescido de uma *poupança* de 1 para a futura cópia: note-se que só após removermos (pelo menos) N elementos é que teremos que copiar outros tantos N elementos para a tabela menor.
O caso em que a remoção é *cara* corresponde a uma diminuição do tamanho da tabela. E nesse caso devemos ter *amealhado* o suficiente para cobrir esses gastos.

i	Resultado	c_i	$\widehat{c}_i$	B_i
0	1 2 3 4 5 6 7 8 9			0
1	1 2 3 4 5 6 7 8	1	2	1
2	1 2 3 4 5 6 7	1	2	2
3	1 2 3 4 5 6	1	2	3
4	1 2 3 4 5	1	2	4
5	1 2 3 4	5	2	1
6	1 2 3	1	2	2
7	1 2	3	2	1
8	1	1	2	2

Em termos de função de potencial, o potencial deve aumentar à medida que a tabela se vai esvaziando (o número de células livres aumenta) máximo deve ser atingido quando a tabela fica apenas com 25% da sua capacidade preenchida, baixando nessa altura para o seu valor mínimo com a taxa de ocupação a 50%.
Seja então a função de potencial definida como

$$\Phi_i = \max(0, F_i - \frac{S_i}{2})$$

em que F e S representam o número de posições na tabela e o número de posições desocupadas (a expressão $\max(0, \_)$ garante apenas que o potencial se mantem não negativo).
Analizemos então os custos estimados face a esta definição

- No caso em que o número de posições desocupadas se mantem a menos de metade da tabela, o potencial mantém-se a zero.

$$\widehat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + 0 - 0 = 1$$

- Nos restantes casos em que a dimensão da tabela não é alterada ($S_i = S_{i-1}$)

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (F_{i-1} + 1) - \tfrac{S_i}{2} - (F_{i-1} - \tfrac{S_{i-1}}{2}) \\ &= 1 + F_{i-1} + 1 - \tfrac{S_i}{2} - F_{i-1} + \tfrac{S_i}{2} \\ &= 2\end{aligned}$$

- Finalmente, quando a tabela é redimensionada para metade ($S_i = \frac{S_{i-1}}{2}$), o número de posições livres passa a ser metade do tamanho da tabela $F_i = \frac{S_i}{2} = \frac{S_{i-1}}{4}$. Nesse caso, o custo real c_i contem também o custo de copiar tantos elementos quanto as posições que ficaram desocupadas ($c_i = 1 + F_i$).

  Note-se que para que a tabela seja realocada é necessário que após a remoção, 3/4 da tabela estejam desocupados, i.e., $F_{i-1} = \frac{3}{4}\, S_{i-1} - 1 = \frac{3}{2}\, S_i - 1 = 3\, F_i - 1$

$$\begin{aligned}&\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + F_i + (F_i - \tfrac{S_i}{2}) - (F_{i-1} - \tfrac{S_{i-1}}{2}) \\ &= 1 + F_i + F_i - \tfrac{S_i}{2} - F_{i-1} + \tfrac{S_{i-1}}{2} \\ &= 1 + F_i + F_i - F_i - (3\, F_i - 1) + 2\, F_i \\ &= 2\end{aligned}$$

9.4.5 Tabelas de Hashing

Um caso em que as tabelas dinâmicas são muito utilizadas é na implementação de tabelas de hashing com open addressing.
As operações de inserção e consulta numa destas tabelas (admitindo um bom comportamento da função de hash) permanecem constantes desde que o factor de carga (i.e., a taxa de ocupação da tabela) permaneça suficientemente baixo.
Supunhamos então que pretendemos manter o factor de carga abaixo de 0.5; sempre que esse valor for ultrapassado a tabela é duplicada.
Vamos assumir que a inserção sem duplicação é feita em tempo constante (1) e que a duplicação da tabela tem um custo proporcional ao tamanho da tabela (N).
Para mostrarmos que o custo amortizado da operação de inserção permanece constante vamos assumir que a função de potencial varia linearmente com o número de posições livres e ocupadas, isto é, vamos tentar determinar uma função de potencial da forma

$$\Phi(s) = a\,.\,L_s + b\,.\,O_s$$

em que L_s e O_s representam o número de posições livres e ocupadas no estado s.
Para calcular as constantes a e b vejamos o que acontece quando uma inserção provoca uma duplicação da tabela.

- No estado anterior devemos ter um potencial que nos permita *suportar o custo adicional* da duplicação da tabela. Este estado i é então caracterizado por
  - $L_i = O_i = \frac{N}{2}$
  - $\Phi(i) = N$

Daí que

$$\begin{aligned} \Phi(i) &= a\,.\,L_i + b\,.\,O_i \\ N &= a\,.\,\tfrac{N}{2} + b\,.\,\tfrac{N}{2} \\ 2 &= a + b \end{aligned}$$

- No estado seguinte $i+1$ esse potencial deverá ser usado na totalidade (passando a zero). Teremos então
  - $\Phi(i+1) = 0$
  - $O_{i+1} = O_i + 1$
  - $L_{i+1} = \underbrace{2 * N}_{\text{novo tamanho}} - (L_i - 1)$

Daí que

$$\begin{aligned} \Phi(i+1) &= a\,.\,L_{i+1} + b\,.\,O_{i+1} \\ 0 &= a\,.\,((2 * N - L_i + 1) + b\,.\,(O_i + 1) \\ 0 &= a\,.\,(2 * N - \tfrac{N}{2} + 1) + b\,.\,(\tfrac{N}{2} + 1) \end{aligned}$$

A Relações de recorrência de 1ª ordem

De forma a podermos resolver relações de 1ª ordem (da forma $x_{n+1} = a_{n+1} \ + b_{n+1}\,x_n$) vamos começar por apresentar os casos mais simples:

1. $x_{n+1} = b\,x_n$

   Expandindo os primeiros elementos desta série, podemos induzir facilmente o caso geral:

   $$\begin{aligned} &x_1 = b\,x_0 \\ &x_2 = b\,x_1 = b^2\,x_0 \\ &x_3 = b\,x_2 = b^3\,x_0 \\ &\ldots \\ &x_n = b^n\,x_0 \end{aligned}$$

2. $x_{n+1} = b_{n+1}\,x_n$

   Também aqui, expandindo os primeiros elementos desta série, podemos induzir o caso geral:

   $$\begin{aligned} &x_1 = b_1\,x_0 \\ &x_2 = b_2\,x_1 = b_1\,b_2\,x_0 \\ &x_3 = b_3\,x_2 = b_1\,b_2\,b_3\,x_0 \\ &\ldots \\ &x_n = x_0\,\Pi_{i=1}^{n}\,b_i \end{aligned}$$

3. $x_{n+1} \ = x_n + c_{n+1}$ Mais uma vez, vamos expandir os primeiros elementos da série:

   $$\begin{aligned} &x_1 = x_0 + c_1 \\ &x_2 = x_1 + c_2 = x_0 + c_1 + c_2 \\ &x_3 = x_2 + c_3 = x_0 + c_1 + c_2 + c_3 \\ &\ldots \\ &x_n = x_0 + \textstyle\sum_{i=1}^{n}\,c_i \end{aligned}$$

4. $x_{n+1} = b_{n+1}\, x_n + c_{n+1}$

   Neste caso, a expansão dos primeiros elementos da série dá apenas uma leve intuição sobre o resultado:

$$\begin{aligned}
x_1 &= b_1\, x_0 + c_1 \\
x_2 &= b_2\, x_1 + c_2 \\
&= b_2\, (b_1\, x_0 + c_1) + c_2 \\
&= b_1\, b_2\, x_0 + c_1\, b_2 + c_2 \\
x_3 &= b_3\, x_2 + c_3 \\
&= b_3\, (b_1\, b_2\, x_0 + c_1\, b_2 + c_2) + c_3 \\
&= b_1\, b_2\, b_3\, x_0 + c_1\, b_2\, b_3 + c_2\, b_3 + c_3 \\
x_4 &= b_4\, x_3 + c_4 \\
&= b_1\, b_2\, b_3\, b_4\, x_0 + c_1\, b_2\, b_3\, b_4 + c_2\, b_3\, b_4 + c_3\, b_4 + c_4 \\
&\ldots
\end{aligned}$$

   Uma forma de resolver esta recorrência consiste em definir uma nova série $\{y_n\}_{n>0}$ de tal forma que

$$x_n = b_1\, b_2 \ldots b_n\, y_n \qquad (\text{com } y_0 = x_0)$$

   Com esta série, podemos reescrever a equação $x_{n+1} = b_{n+1}\, x_n + c_{n+1}$ como

$$b_1\, b_2 \ldots b_{n+1}\, y_{n+1} = b_{n+1}\, (b_1\, b_2 \ldots b_n)\, y_n + c_{n+1}$$

   Dividindo ambos os membros por $b_1\, b_2 \ldots b_{n+1}$ obtemos a recorrência

$$y_{n+1} = y_n + \frac{c_{n+1}}{b_1\, b_2 \ldots b_{n+1}}$$

   Que tem a forma já vista acima. Seja d_i definido por

$$d_i = \frac{c_i}{b_1\, b_2 \ldots b_i}$$

   Então temos,

$$y_n = x_0 + \sum_{i=1}^{n} d_i = x_0 + \sum_{i=1}^{n} \frac{c_i}{b_1\, b_2 \ldots b_i}$$

   Substituindo agora na definição de y_n temos a solução para a recorrência:

$$x_n = b_1\, b_2 \ldots b_n\, (x_0 + \sum_{i=1}^{n} \frac{c_i}{b_1\, b_2 \ldots b_i})$$

B Exercícios de Testes

1. *(Teste, 2007/2008)*

   Faça a análise assimptótica do tempo de execução da função `minpairs`.

```
typedef struct {int p; int s;} Pair;

void minpairs(Pair a[], char v[], int n) {
   int i,j;
   for (i=0; i<n, i++) v[i]=1;
   for (i=n-1; i>=0; i--)
      for (j=i-1; j>=0; j--)
         if (a[i].s <= a[j].s) v[j]=0;
}
```

2. *(Teste, 2007/2008)*
   Considere a seguinte função recursiva.
   Sabendo que $T_{\text{processa}}(N) = N$ em que N corresponde ao tamanho do array recebido pela função `processa`, escreva uma recorrência que descreva o comportamento temporal da função `exemplo` e indique, justificando, a solução dessa recorrência.

```
void exemplo(int a[], int n) {
   int x = n/2;
   if (n>0) {
      exemplo(a,x);
      processa(a,n);
      exemplo(a+x,n-x);
   }
}
```

3. *(Exame de recurso, 2007/2008)*
   Analise a complexidade da função `bsort` identificando o melhor e pior casos.

```
int bubble(int a[], int n) {
   int i, k;
   // n > 0
   i = n-1; k = 0;
   // n > 0 && i = n-1 && k = 0
   while (i > 0) {// i >= 0 && k < n
      if (a[i] < a[i-1]) {
         swap(a,i,i-1); k++}
      i--;
   }
   // k < n
   return k;
}
```

```
void bsort(int a[], int n) {
   while (bubble(a,n));
}
```

4. *(Época especial, 2007/2008)*
   Analise a complexidade da função `factoriais`.

```
int factorial(int n) {
   int i, k;
   // n >= 0
   k = 1; i=0;
   // n > 0 && k = 1
   while (i < n) {
     // i <= n 0 && k = i!
     i++; k*=i;
   }
   // k = n!
   return k;
}
```

```
void factoriais (int a[], int n) {
   int i;
   for (i=0; (i<n); i++)
      a[i] = factorial (i);
}
```

5. *(Época especial, 2007/2008)*

Analise a complexidade da função `mod` em função do número de bits dos seus argumentos. Nomeadamente: identifique o melhor e pior casos, e para este último, diga qual o número de iterações do ciclo `while`.

```
int mod (int x, int y) {
// x >= 0 && y > 0
int r = x;
// x >= 0 && y > 0  && r == x
while (r>=y) {
   // r >= 0 && (\exists q >= 0 : q * y + r = x)
   r = r - y;
   }
// 0 <= r < y && (\exists q >= 0 : q * y + r = x)
return r;
```

6. *(Exame de recurso, 2008/2009)*
   Analise o tempo de execução no melhor e no pior caso da seguinte função, que determina, para cada posição i do array A, se $A[i]$ é inferior a todos os elementos do array B até à posição $i-1$.

```
void minimos (int A[], int B[], int  C[]) {
  for (i = N-1; i >= 0; i--) {
    C[i] = 1;
    j = 0;
    while (C[i] && j < i) {
      if (B[j] <= A[i]) C[i] = 0;
      j++;
    }
  }
}
```

7. *(Exame de recurso, 2008/2009)*
   Analise o tempo de execução da seguinte função recursiva, utilizando para isso uma recorrência.

```
int binarymin (int a[], int first, int last) {
  if (first == last) return first;
  if (first < last) {
    int mid = (first + last) / 2;
    int m1 = binarymin (a, first, mid);
    int m2 = binarymin (a, mid+1, last);
    if (a[m1] <= a[m2]) return m1; else return m2;
   }
}
```

8. *(Teste, 2009/2010)*
   Considere a operação de rotação circular "shift-rotate" de um *array* de dimensão n. A seguinte função recursiva efectua k operações dessas recursivamente. Analise o seu tempo de execução utilizando uma recorrência. Considere que $0 \leq k \leq n$.

```
void shift (int u[], int n, int k)
{ if (k > 0) {
    int i = n-1, aux = u[n-1];
    while (i>0) {
      u[i] = u[i-1];
      i--;
    }
    u[0] = aux;
    shift (u, n, k-1);
  }
}
```

9. *(Teste, 2009/2010)*

Apresente uma versão sem recursividade da função `shift` que execute em tempo $O(n)$ utilizando espaço também em $O(n)$.

10. *(Exame de recurso, 2009/2010)*
Considere a seguinte função que, dado um vector de booleanos que representa (os bits de) um número inteiro (armazenados por ordem decrescente da sua significância), faz o incremento a esse número.

```
int inc (int bits [], int n) {
   while (n>=0) {
     n--;
     if (bits [n]) bits [n] = 0;
     else break;
   }
   if (n<0) return 0;
   bits [n] = 1;
   return 1;
}
```

(a) Faça a análise assimptótica do tempo de execução desta função (em função do tamanho do array). Não se esqueça de identificar o melhor e pior casos de execução desta função.

(b) Mostre que o tempo amortizado de execução desta função é constante. Para isso analise o tempo de execução de `N` incrementos ao vector que representa 0 (os bits todos a 0).

11. *(Exame de recurso, 2009/2010)*
A função seguinte calcula o tamanho da maior sequência de bits 1 na representação de um número inteiro. Faça a análise assimptótica do seu tempo de execução. Explicite o comportamento no melhor e no pior caso, e utilize notação assimptótica para expressar as suas conclusões sobre o comportamento do algoritmo.

```
int longest(int n) {
    int c,l = 0;
    while(n!=0) {
       c = 0;
       while (n % 2 == 1) {
          c = c + 1; n = n / 2;
       }
       if (c>l) { l = c; };
       n = n/2;
    }
    return l;
}
```

12. *(Exame de recurso, 2009/2010)*
Escreva uma equação de recorrência para o tempo de execução da seguinte função, que calcula valor máximo de um array `a` (não vazio) com `n` valores. Com base nessa equação, efectue uma análise assimptótica do tempo de execução da função.

```
int maxArray(int a[], int n) {
    int m,l,r;
    if (n == 1) return a[0];
    m = n/2;
    l = maxArray(a,m);
    r = maxArray(a+m,n-m);
    if (l>r) return l
    else return r;
}
```

13. *(Época especial, 2009/2010)*

Considere a seguinte função sobre uma árvore binária, que devolve o elemento alcançável através de um determinado caminho.

Analise o seu tempo de execução, no melhor e no pior caso, tendo em conta que o tamanho do input é dado pelo número de elementos da árvore N e pelo comprimento do caminho L. Utilize equações de recorrência na sua análise.

```
int walk(TreeNode *root, ListNode *path) {
      if (! root) return (-1);
     if (! path) return (root->val);
     if (path -> direction == LEFT) {
        return walk(root->left,path->next);
      } else {
          return walk(root->right,path->next);
      }
}
```

14. *(Teste, 2010/2011)*

Considere o seguinte algoritmo recursivo para cálculo do máximo numa árvore binária com inteiros positivos (não necessariamente ordenada).

```
typedef struct tree {
     int val;
     struct tree *left;
     struct tree *right;
} *Tree;
```

```
int maximo(Tree t) {
     int dir, esq, res;
     if (!t) return -1;
     esq = maximo(t->left);
     dir = maximo(t->right);
     res = t->val;
     if  (res < esq) res = esq;
     if (res < dir) res = dir;
     return res;
}
```

(a) Escreva uma equação de recorrência para o tempo de execução deste algoritmo, desenhe a respectiva árvore, e represente o seu tempo de execução utilizando notação assimptótica.

(b) Suponha que se trata de uma árvore binária de pesquisa. Apresente um algoritmo que resolva o mesmo problema de forma mais eficiente, e repita a análise que efectuou na alínea anterior.

15. *(Teste, 2010/2011)*

Considere a seguinte definição de uma função que testa se uma dada função `teste` é válida para algum subconjunto de `v`.

Sabendo que a função `teste` tem uma complexidade linear no tamanho do vector de entrada, apresente uma relação de recorrência que traduza a complexidade da função `f` em função do tamanho `n` do vector recebido como argumento.

Apresente ainda uma solução dessa recorrência, justificando-a informalmente.

```
int f (int v[], n) {
int i, r;

r = 0;
if (n==0) return 0;
if (teste (v,n)) r = n;
else
  for (i=0;(r==0)&&(i<n-1);i++) {
        swap (v,i,n-1);
        r = f(v,n-1));
        swap (v,i,n-1);
     }
return r;
}
```

16. *(Exame de recurso, 2010/2011)*
Considere a seguinte definição de uma função que calcula a altura de uma árvore AVL.

```
int altura (AVLTree t) {
   int r = 0;
   while (t) {
     if (t->Bal == E) t = t->esq
     else t = t->dir
     r++
   }
 return r;
}
```

Apresente uma versão recursiva da mesma função e efectue a respectiva análise de complexidade em função do tamanho (i.e., número de elementos) da árvore recebida como argumento. Assuma que a árvore em causa está balanceada e com os factores de balanço correctamente calculados.

17. *(Época especial, 2010/2011)*
Considere a sequinte definição de uma função que calcula a altura de uma árvore.

```
typedef struct nodo {
    int v;
    struct nodo *esq, *dir;
  } Nodo, *BTree;
```

```
int altura (BTree a) {
   if (a == NULL) return 0;
   else return (1 + max (altura (a->esq),
                         altura (a->dir)));
}
```

Para cada um dos casos (extremos) de a árvore estar perfeitamente equilibrada ou perfeitamente desiquilibrada, apresente relações de recorrência que traduzam o tempo de execução desta função em função do tamanho da árvore de entrada (i.e., do número de nodos da árvore). Apresente os resultados da análise global em notação assimptótica.

18. *(Teste, 2011/2012)*
Considere a seguinte função que gera (imprimindo no `stdout`) todas as combinações de N bits.

Faça a análise assimptótica do seu tempo de execução. Para isso, defina uma recorrência que exprima essa complexidade, desenhe a árvore de recursão e apresente uma solução para a recorrência apresentada.

```
void bitsSeq (int N, char seq[], char *end) {
    if (N==0) {
        *end = 0;
        printf ("%s\n", seq);
    } else {
        *end = '0';
        bitsSeq (N-1, seq, end+1);
        *end = '1';
        bitsSeq (N-1, seq, end+1);
    }
}
```

19. *(Teste, 20111/2012)*
Seja `V` um vector com `N` números inteiros diferentes. Uma forma de representar um subconjunto `X` de `V` consiste em usar um vector `x` com `N` valores booleanos em que `x[i] == 1` sse `V[i]` $\in$ `X`.

Por exemplo para `V = {1,2,3,4,5}` o array `x={1,0,0,1,0}` representa o subconjunto $\{1, 4\}$.

Suponha que existe a função `int teste(int V[], int N, int x[])` que testa se um determinado subconjunto `x` de `V` satisfaz uma dada propriedade. Assuma ainda que a função executa em tempo linear em `N`.

(a) Defina uma função `int forall(int V[], int N, int x[])` que testa se a dita popriedade é válida para todos os subconjuntos de `V`. No caso de insucesso, a função deve ainda preencher o vector `x` com um conjunto que não satisfaça a propriedade.

(b) Identifique o pior caso de execução da função apresentada na alínea anterior e analise a complexidade dessa função para esse caso. Assuma que a função `teste` executa em tempo linear relativamente ao tamanho do array `V`.

20. *(Exame de recurso, 2011/2012)*
Considere a seguinte função que constrói uma árvore binária a partir de uma *min-heap*.

```
BTree heapToTree (MinHeap h) {
   return hToTAux (h, 0);
}
BTree hToTAux (MinHeap h, int r){
   BTree new;
   if (r >= h->used) return NULL;
   new = (BTree) malloc (sizeof (struct btree));
   new->value = h->values[r];
   new->left = hToTAux (h, 2*r+1);
   new->right = hToTAux (h, 2*r+2);
   return new;
}
```

Faça a análise da complexidade da função `hToAux`, em função do tamanho da *min-heap*.

21. *(Época especial, 2011/2012)*
A função `rep` ao lado testa se um vector de inteiros não tem elementos repetidos.
Identifique o melhor e pior casos do seu tempo de execução.
Apresente ainda uma relação de recorrência que traduza o tempo de execução da função no pior caso, bem como uma solução para essa recorrência.

```
int nrep (int v[], int n) {
   int r = 1, i;
   if (n>0) {
      for (i=1; (r && (i<n)); i++)
          if (v[0] == v[i]) r = 0;
      r = r && nrep (v+1, n-1);
   }
   return r;
}
```

22. *(Época especial, 2011/2012)*

Suponha que se usam vectores de coeficientes para representar polinómios. O polinómio $4.2\,x^5 - 3.2\,x^2 - 0.5$ será representado por um array em que as primeiras seis componentes são `-0.5`, `0`, `-3.2`, `0`, `0` e `4.2`.
A função apresentada calcula o valor de um polinómio num ponto.
Analise a complexidade desta função em função do tamanho do polinómio e apenas em termos do número de adições (A) e mutiplicações (M) efectuadas.

```
float valor (float p[], int n, float x) {
int i; float r;
i = n; r = 0;
while (i>0) {
    i = i-1;
    r = r * x + p[i];
   }
return r;
}
```

23. *(Teste, 2012/2013)*
Defina uma função `break_list` que parta uma lista ligada em dois segmentos de igual comprimento, devolvendo o endereço da lista correspondente ao segundo. Se a lista inicial tiver comprimento ímpar, o segundo segmento terá mais um elemento do que o primeiro. Por exemplo, se a lista original tiver os elementos `1,2,3,4,5,6,7`, após a execução da função, a lista passa a ter apenas os valores `1,2,3` e o resultado da função deverá ser a lista com os elementos `4,5,6,7`.

A função a definir só terá que produzir resultados válidos para listas de comprimento superior ou igual a 2.
Certifique-se que a solução apresentada executa em tempo linear no tamanho da lista e que não aloca memória adicional.

```
typedef struct lnode {
  int info;
  struct lnode *next;
} Lnode, *List;
```

24. *(Teste, 2012/2013)*
Escreva uma recorrência e diga qual o tempo de execução assimptótico da função seguinte, que constrói uma árvore binária a partir de uma lista ligada de inteiros (usando a função `break_list` da alínea anterior).

```
typedef struct tnode {
  int info;
  struct tnode *left, *right;
} Tnode, *BTree;
```

```
Tnode* mkTree (Lnode *l) {
  Tnode *new;
  Lnode *l2;
  if (!l) return NULL;
  new = malloc(sizeof(Tnode));
  if (!l->next) {
    new -> left = new -> right = NULL;
    new -> info = l -> info;
    free (l); return new;
  }
  l2 = break_list(l);
  new -> info = l2 -> info;
  new -> left = mkTree (l);
  new -> right = mkTree (l2->next);
  free (l2); return new;
}
```

25. *(Teste, 2012/2013)*
Relembre a seguinte função de consulta de uma árvore binária de procura:

```
int elem (BTree a, int x) {
```

```
    while (a != NULL)
        if (a->info == x) break;
        else if (a->valor > x) a = a->left;
        else a = a->right;
    return (a != NULL)
}
```

Admitindo que se trata de uma árvore perfeitamente balanceada,

(a) Determine o tempo médio de execução desta função, no caso de o elemento pertencer à árvore. Admita que o valor a procurar está com igual probabilidade em qualquer posição da árvore. Note que uma árvore balanceada com N nodos tem aproximadamente $\log_2 N$ níveis.

(b) Calcule o tempo de execução desta função no caso de insucesso (i.e., no caso de o elemento não existir na árvore). O que pode concluir sobre o comportamento assimptótico médio desta função?

26. *(Exame de recurso, 2012/2013)*
Considere a função que calcula a potência inteira de um número.
Assumindo que as operações elementares sobre inteiros (multiplicação, divisão e resto da divisão) de dois números executam em tempo linear no número de bits da representação binária dos números, faça a análise da complexidade (usando uma recorrência) desta função para o pior caso (não se esqueça de caracterizar esse pior caso), em função do número de bits usados na representação dos números inteiros.

```
int pot (int x, int n) {
    int r = 1;

    if (n>0) {
        r = pot (x*x,n/2);
        if (m%2 == 1) r = r * x;
    }
    return r;
}
```

27. *(Exame de recurso, 2012/2013)*
Dado um vector `v` de `N` números inteiros, a mediana do vector define-se como o elemento do vector em que existem no máximo `N/2` elementos (estritamente) menores do que ele, e existem no máximo `N/2` elementos (estritamente) maiores do que ele. Se o vector estiver ordenado, a mediana corresponde ao valor que está na posição `N/2`.

Considere a seguinte definição de uma função que calcula a mediana de um vector.

Assumindo que a função `quantos` executa em tempo linear no comprimento do vector de input, identifique o melhor e pior caso de execução da função `mediana`. Para cada um desses casos determine a complexidade assimptótica da função `mediana`.

```
int mediana (int v[], int N) {
    int i, m, M;
    for (i=0; i<N; i++) {
        quantos (v,N, v[i], &m, &M);
        if (m <= N/2) && (M <= N/2) break;
    }
    return v[i];
}
```

28. *(Exame de recurso, 2012/2013)*
Calcule a complexidade média da função apresentada na alínea anterior. Para isso, assuma que os valores do vector são perfeitamente aleatórios e, por isso, que a probabilidade de o elemento numa qualquer posição do vector ser a mediana é uniforme ($= \frac{1}{N}$).

29. *(Época especial, 2012/2013)*
Considere o seguinte programa anotado para calcular a raíz quadrada inteira de um número.
Assumindo que as operações elementares sobre inteiros (multiplicação, adição e subtracção) executam em tempo constante, faça a análise da complexidade do código acima para o pior caso (não se esqueça de caracterizar esse pior caso), em função do número de bits usados na representação do número `n`.

```
// n == n0 > 0
r = 1;
// n == n0 >= 0 && r == 1
while (r*r < n)
    // n == n0 >= 0 && (r-1)*(r-1) <= n
    r = r+1;
// n == n0 >= 0
// && (r-1)*(r-1) <= n && r * r >= n
if (r * r > n) r = r-1;
// r*r <= n0 && (r+1)*(r+1) > n0
```

30. *(Época especial, 2012/2013)*
Considere a seguinte alternativa para o problema apresentado na alínea anterior.
Faça a análise da complexidade deste código para o pior caso em função do número de bits usados na representação do número `n`.

```
// n == n0 > 0
r = 0 ; s = n;
while (r < s-1) {
    m = (r+s)/2;
    if (m*m > n) s = m;
    else r = m;
}
// r*r <= n0 && (r+1)*(r+1) > n0
```

31. *(Teste, 2013/2014)*
Considere as seguintes definições em que a função `crescente` calcula o comprimento do maior prefixo crescente de um vector de inteiros, e `maxcresc` calcula o tamanho do maior segmento crescente de um vector de inteiros.

```
int crescente (int v[], int N) {
    int i;
    for (i=1; i<N; i++)
        if (v[i] < v[i-1])
           break;
    return i;
}
```

```
int maxcresc (int v[], int N){
    int r = 1, i = 0, m;
    while (i<N-1) {
        m = crescente (v+i, N-i);
        if (m>r) m = r;
        i=i+1;
    }
return r;
}
```

Para cada uma das funções, identifique o melhor e o pior caso de execução e faça a respectiva análise assimptótica do tempo de execução, com base nas comparações entre elementos do array.

32. *(Teste, 2013/2014)*

A seguinte função recursiva calcula o número de elementos diferentes presentes num array de inteiros. Apresente duas recorrências, correspondentes ao melhor e pior casos de execução, e resolva-as, identificando em que situação ocorre cada caso.

```
int diferentes (int v[], int N) {
  int d, i;
  if (N == 0) return 0;
  d = diferentes(v+1, N-1);
  for (i=1; (i<N) && (v[i]!=v[0]); i++);
  if (i==N) return d+1 else return d;
}
```

33. *(Teste, 2013/2014)*
Calcule o número médio de comparações entre os elementos do array que são feitas pela função crescente (definida na questão acima). Para isso considere que os valores do array são perfeitamente aleatórios e por isso, que para qualquer índice `i`, a probabilidade de a posição `i` conter um valor menor do que a posição `i-1` é 0.5.

34. *(Teste, 2013/2014)*
Modifique a definição da função `maxcresc` (definida acima) de forma a obter uma função que execute no pior caso em tempo linear no comprimento do array de entrada.

35. *(Exame de recurso, 2013/2014)*
Considere a função recursiva `contaAux`. Calcule o seu tempo de execução assimptótico no melhor e pior caso, identificando bem esses casos, e apresentando as recorrências e árvores de recursão apropriadas para ambos os casos.

```
int contaAux (int cont[], int n)  {
  if (n == 0) return 0;
  else if (cont[n] >= n) return n;
       else return contaAux(cont, n-1);
}
```

36. *(Teste, 2014/2015)*
Considere o problema de determinar, num array de inteiros o valor da máxima soma de elementos consecutivos. Por exemplo no array `[1,-5,2,-1,4,-3,1,-7,3]` esse valor corresponde a `5` (a soma `2-1+4`.

Efectue a análise do comportamento assimptótico da função `A(N)`, que representa o número de acessos ao array `v` efectuados pela função `maxSoma` apresentada em baixo. Assuma para isso que a invocação da função `soma (v,a,b)` corresponde a fazer `(b-a+1)` acessos o array.

```
int soma (int v[], int a, int b){
  int r = 0, i;
  for (i=a; i<=b; i++) r=r+v[i];
  return r;
}
```

```
int maxSoma (int v[], int N){
  int i, j, r=0, m;
  for (i=0; i<N; i++)
    for (j=0; j<=i; j++){
      m = soma(v,j,i);
      if (m>r) r = m;
  }
  return r;
}
```

37. *(Teste, 2014/2015)*

Considere a definição recursiva da função `maxSoma`.
De forma a analizar o comportamento desta função: (1) apresente uma relação de recorrência que traduza a complexidade desta função em termos do número de acessos ao array `v`; (2) apresente a correspondente árvore de recursão, e (3) apresente uma solução para a referida recorrência.

```
int maxSomaR (int v[], int N){
    int r=0, m1, m2, i;
    if (N>0) {
      m1 = 0; m2=0;
      for (i=0;i<N;i++){
        m2 = m2+v[i];
        if (m2>m1) m1=m2;
      }
      m2 = maxSomaR (v+1,N-1);
      if (m1>m2) r = m1; else r = m2;
    }
    return r;
}
```

38. *(Teste, 2014/2015)*
Na definição iterativa da função `maxSoma` apresentada acima, o ciclo mais interior faz sucessivas invocações da função `soma` que repetem muitos cálculos (e consequentemente acessos ao array). Apresente uma definição alternativa que evita as invocações da função `soma`.

39. *(Teste, 2014/2015)*
Considere a seguinte função que calcula o complemento para dois de um número inteiro armazenado num array de *booleanos*.

```
void complemento (char b[], int N){
  int i = N-1;
  while ((i>0) && !b[i])
    i--;
  i--;
  while (i>=0) {
    b[i] = !b[i]; i--;
  }
}
```

De forma a analizar a complexidade desta função em termos do número de elementos do array que são alterados (i.e., o número de iterações do segundo ciclo):

(a) Identifique o pior caso, apresentando a complexidade (i.e, o número de elementos do array que são alterados) da função nesse caso.

(b) Calcule a complexidade da função no caso médio. Considere para isso que o valor do input é perfeitamente aleatório, i.e., que a probabilidade de cada posição do array ser um `0` ou `1` é 0.5.

40. *(Exame de recurso, 2014/2015)*
Considere as seguintes funções que preenchem um array com os elementos de uma heap por ordem crescente, retornando o número de elementos preenchidos.

```
int toArray1 (BTree b, int v[]){
   int i=0;
   while (b!=NULL) {
      v[i++] = b->value;
      b = removeRoot (b);
   }
   return i;
}
```

```
int toArray2 (Btree b, int v[]) {
  int l=0,r=0;
  if (b!=NULL) {
     v[0] = b->value;
     l = toArray2 (b->left,v+1);
     r = toArray2 (b->right,v+l+1);
     merge (v+1,l,r);
     }
  return (1+l+r);
  }
}
```

Admita que as árvores em causa estão balanceadas.

(a) Determine a complexidade assimptótica da função `toArray1`, assumindo que a função `removeRoot` tem uma complexidade de `log N` sempre que é invocada com uma árvore com `N` elementos.

(b) Apresente uma recorrência que traduza a complexidade da função `toArray2` assumindo que a função `merge` tem uma complexidade `N` quando invocada com um array de tamanho `N = l+r`. Apresente ainda uma solução para essa recorrência.

41. *(Exame de recurso, 2014/2015)*
Relembre o algoritmo de procura numa arvore binária de procura.
Assumindo que se trata de uma arvore balanceada, determine o numero medio de nodos consultados numa arvore (i.e., o numero de itreracoes do ciclo while) com N nodos.

```
Node *search (BTree a, int x){
   while ((a!=NULL) && (a->value != x))
      if (a->value > x) a = a->right;
         else a = a->left;
      return a;
}
```

Assuma que, caso o elemento a procurar exista na arvore, ele pode estar com igual probabilidade em qualquer ponto da arvore.

42. *(Exame de recurso, 2014/2015)*
Considere que se implementa uma tabela de Hash com tratamento de colisões por open addressing e usando arrays dinâmicos.

Considere ainda que as inserções e consultas de uma nova chave executam em tempo constante (1), desde que não haja realocação do array.

Esta assumpção só é válida quando o factor de carga da tabela não é superior a 50%. Por isso, quando esse factor atinge esse valor, i.e., a tabela está 50% ocupada, o array é duplicado e essa duplicação tem um custo adicional igual ao tamanho do array.

Mostre que o custo amortizado da inserção é constante.

43. *(Época especial, 2014/2015)*

Considere que se usa a função `maxInd` para definir a seguinte função para ordenar um array de `N` inteiros.

```
void mSort (int v[], int N){
   int m;
   if (N>1) {
      m = maxInd (v,N);
      swap (v,m,N-1);
      mSort (v,N-1);
   }
}
```

- Apresente uma recorrência que exprima o número de comparações entre elementos do array efectuadas por esta função (assuma que maxInd executa em tempo N para um array de tamanho N.
- Apresente uma solução dessa recorrência começando por desenhar a árvore de recursão que lhe está associada.

44. *(Época especial, 2014/2015)*
Considere que se implementa uma tabela de Hash com tratamento de colisões por open addressing e usando arrays dinâmicos.

Considere ainda que as inserções e consultas de uma nova chave executam em tempo constante (`1`), desde que não haja realocação do array.

Esta assumpção só é válida quando o factor de carga da tabela não é superior a 50%. Por isso, quando esse factor atinge esse valor, i.e., a tabela está 50% ocupada, o array é duplicado e essa duplicação tem um custo adicional igual ao tamanho do array.

Considere ainda que quando uma chave é removida, apenas se marca essa posição como removida (e por isso a remoção de um elemento também executa em tempo constante). No entanto, quando o número de chaves efectivas (i.e., não removidas) corresponde a 1/8 (12.5%) da capacidade da tabela é feita uma realocação da tabela para metade da sua anterior capacidade. Esta realocação executa em tempo linear à capacidade da tabela e remove todas as células marcadas como apagadas.

(a) Mostre que o custo amortizado da remoção de uma chave é constante.

(b) Mostre que se se decidir realocar a tabela para metade da sua capacidade quando 25% das células estiverem apagadas a operação de remoção deixa de ter um custo amortizado linear.

45. *(Teste, 2015/2016)*
Por vezes há mais do que uma dimensão a considerar no *input* de um algoritmo. Considere a seguinte função que, dado um array com `N` *strings*, todas de comprimento `M`, calcula o índice onde se encontra a que é alfabeticamente menor. O tempo de execução de `minInd` será dado por uma função $T(\texttt{N},\texttt{M})$.

```
int minInd (char *nomes [], int N) {
  int i, r = 0;
  for (i=1; i<N; i++)
    if (strcmp (nomes[i], nomes[r]) < 0)
      r = i;
  return r;
}
```

Assumindo que `strcmp` executa no melhor caso em tempo constante (1) e no pior caso em tempo linear no tamanho das strings (M), analise assimptoticamente a função `minInd` no melhor e pior caso.

46. *(Teste, 2015/2016)*

Considere a definição de uma versão do algoritmo *quicksort* que particiona o *array* de input em 3 secções que são depois recursivamente ordenadas. A função `part3` executa em *tempo linear*; utiliza como pivots o primeiro e o último elemento do *array*, que são colocados nas posições finais `q` e `s`.

```
void qSort3 (int A[], int p, int r) {
   if (p < r) {
      int q, s;
      part3(A, p, r, &q, &s);
      qSort3(A, p, q-1);
      qSort3(A, q+1, s-1);
      qSort3(A, s+1, r);
   }
}
```

Sabendo que o melhor caso para o tempo de execução de `qSort3` ocorre quando as 3 secções criadas por `part3` têm aproximadamente o mesmo tamanho, e o pior caso quando a função de partição cria sempre duas secções vazias, escreva e resolva as recorrência correspondentes a cada um destes casos.

47. *(Teste, 2015/2016)*
Considere a seguinte função que compara os valores (inteiros e não negativos) de dois inteiros representados por arrays de `N` bits. Assume-se que esses bits estão armazenados do menos significativo (no índice `0`) ao mais significativo (no índice `N-1`).

```
int intcmp (int n1[], int n2[], int N) {
  int i;
  for (i=N-1; (i>=0) && (n1[i] == n2[i]); i--);
  if (i<0) return 0;
  else return (n1[i] - n2[i]);
}
```

- Analise a complexidade desta função no caso médio. Para isso determine o número médio de comparações (`n1[i] == n2[i]`) feitas, assumindo que os valores do entrada são dois arrays aleatórios (onde em cada posição existe com igual probabilidade o valor `0` ou `1`).
- A função `strcmp` referida na parte A tem um comportamento semelhante a esta função. A única diferença é que a probabilidade de dois caracteres serem iguais é de $\frac{1}{256}$. Diga por isso qual a complexidade média da função `minInd` apresentada acima.

48. *(Exame de recurso, 2015/2016)*
Considere a definição ao lado que calcula quantas vezes ocorre o elemento mais frequente de um array.

- Identifique o melhor e pior casos da execução desta função.
- Para o pior caso identificado acima, determine o número de comparações efectuadas entre elementos do vector.

```
int mf (int v [], int N) {
  int m, r;
  for (r=0, i=0; i<N-r; i++){
    for (m=1, j=i+1; j<N; j++)
      if (v[i]==v[j]) m++;
    if (m>r) r=m;
  }
  return r;
}
```

49. *(Exame de recurso, 2015/2016)*

Considere a seguinte definição de um tipo para representar árvores AVL de inteiros (**balanceadas**). Considere ainda a definição da função `deepest` que determina o nodo mais profundo de uma árvore binária, retornando o nível em que ele se encontra.

(a) Mostre que a função apresentada tem uma complexidade linear no número de elementos da árvore argumento.

(b) Apresente uma definição alternativa, substancialmente mais eficiente, tirando partido da informação presente no factor de balanço de cada nodo. Diga qual a complexidade da função que definiu.

```
typedef struct avlnode {
  int value;
  int bal; // Left/Bal/Right
  struct avlnode *left, *right;
} *AVLTree;

int deepest (AVLTree *a) {
  AVLTree l, r;
  int hl, hr, h;
  if (*a == NULL) h = 0;
  else {
    l = (*a)->left; r = (*a)->right;
    hl = deepest (&l); hr = deepest (&r);
    if ((hl>0) && (hl > hr)) {
      *a = l; h = hl+1;}
    else if (hr>0) {
      *a = r; h = hr+1;}
    else h=1;
  }
  return r;
}
```

50. *(Exame de recurso, 2015/2016)*
Relembre a definição recursiva da função *merge-sort* de ordenação de um vector e que usa uma função `merge` de fusão de dois sub-arrays. Admita por hipótese que a função `merge` executa no melhor caso em tempo constante. Calcule a complexidade da função apresentada no melhor caso.

```
void merge_sort (int N, int v[N]) {
  int m;
  if (N>1) {
    m = N/2;
    merge_sort (m,v);
    merge_sort (N-m,v+m);
    merge (v,N,m);
  }
}
```

51. *(Exame de recurso, 2015/2016)*
Uma forma de implementar uma queue é usando duas stacks: stack A e stack B. As operações de inserção (enqueue) e remoção (dequeue) são realizadas usando as operações de inserção (push) e remoção (pop) de stacks da seguinte forma:

enqueue a inserção faz-se sempre com um pop na stack A

dequeue se existirem elementos na stack B, a remoção faz-se através de um pop na stack B; caso contrário começa-se por passar todos os elementos da stack A para a stack B (fazendo pop em A e push em B). de seguida remove-se (pop) o elemento da stack B.

Assumindo que todas as operações sobre stacks executam em tempo constante, o pior caso da operação de dequeue é linear no tamanho da queue.

Mostre que ambas as operações descritas sobre queues executam em tempo amortizado constante.

Se decidir usar o método do potencial, use como função de potencial $\Phi(Q) = 2 * \text{size}\ (Q_A)$ (em que Q_A corresponde à stack A da queue Q).

52. *(Teste, 2016/2017)*
Considere a seguinte definição da função `maxSomaConseq` que calcula a máxima soma de elementos consecutivos de um array.

```
int maxSomaPref (int v[], int N) {
    int i, r, s;
    for (i=r=s=0; i<N; i++) {
       s+=v[i];
       if (s>r) r=s;
    }
    return r;
    }
}
```

```
int maxSomaConseq(int v[], int N) {
    int x,y;
    if (N==0) return 0;
    x = maxSomaPref (v,N)
    y = maxSomaConseq (v+1,N-1);
    if (x > y) return x;
    else return y;
}
```

(a) Apresente e resolva uma recorrência que traduza o comportamento da função `maxSomaConseq` em termos do **número de acessos ao array**.

(b) Uma forma de tornar esta função mais eficiente (evitando re-calcular muitas somas) consiste em usar um vector `somas` onde na posição `i` se encontra o resultado de `maxSomaPref (v+i,N-i)`. Note ainda que este array pode ser preenchido de uma forma eficiente sem usar a função `maxSomaConseq` (basta preenchê-lo do fim para o início).
Usando as observações acima, apresente uma alternativa para a definição de `maxSomaConseq` que seja linear no tamanho do array.

53. *(Teste, 2016/2017)*
Considere o algoritmo típico de procura de um elemento numa *árvore binária de procura* nos seguintes cenários:

(i) a árvore é equilibrada (i.e. a sua altura é logarítmica no número de elementos da árvore), e *o elemento procurado ocorre na árvore*, com igual probabilidade em qualquer nó;

(ii) a árvore é equilibrada, e a probabilidade de o elemento procurado ocorrer na árvore é dada por p, e quando ocorre ocorre com igual probabilidade em qualquer nó.

(iii) a árvore é totalmente desequilibrada, degenerando numa lista, e *o elemento procurado ocorre na árvore*, com igual probabilidade em qualquer nó;

Para cada um dos cenários acima, efectue a *análise de caso médio* do tempo de execução do algoritmo, contabilizando o número de comparações efectuadas.

54. *(Exame de recurso, 2016/2017)*
Considere a seguinte definição de uma função que calcula o número de bits a `1` na representação de um número inteiro (*hamming weight*).
Identifique o melhor e pior casos do custo da execução (número de iterações do ciclo `while`) desta função em termos do tamanho (número de bits usados) da representação dos números inteiros.

```
int hamming (unsigned int x){
  int r=0;
  while (x!=0) {
    if (x%2 == 1) r++;
    x=x/2;
  }
  return r;
}
```

Note que para um tamanho N fixo do input, a gama de valores possíveis para o input vai de 0 (todos os bits a `0`) até $2^N - 1$ (todos os bits a `1`).

55. *(Exame de recurso, 2016/2017)*
Considere a função `hamming` apresentada acima.

    (a) Assumindo uma amostra aleatória (em cada posição da representação do número pode estar, com igual probabilidade, `0` ou `1`), diga qual a probabilidade de acontecer cada um dos casos identificados na questão anterior.

    (b) Calcule o custo médio da execução (número de iterações do ciclo `while`) dessa função.

56. *(Exame de recurso, 2016/2017)*
Uma forma alternativa de permitir remoções em árvores AVL é marcar os nodos apagados com uma flag (tal como exemplificado no exercício **??**). Esta solução tem complexidade logarítmica no número de nodos da árvore, e é por isso aceitável desde que o número de nodos apagados não seja muito grande (se o número de apagados corresponder a metade dos elementos, a altura da árvore é sensivelmente igual à altura da árvore sem os apagados).

    Considere que existe definida uma função `int inorder (AVL a, LInt *l)` que coloca em `*l` o resultado de uma travessia inorder da árvore `a`, retornando o comprimento da lista produzida. Essa função não inclui na lista os nodos marcados como apagados.

    Assuma que esta função executa em tempo linear no número de elementos da árvore.

    Considere agora que sempre que o número de apagados ultrapassa o número de não apagados, se faz uma limpeza da árvore e que consiste em gerar uma lista dos elementos da árvore para depois gerar uma árvore a partir desta lista (funções `inorder` e `fromList` da alínea **??**). Assuma que essa operação executa em tempo linear no tamanho da árvore.

    Mostre, usando um dos métodos de análise amortizada estudados, que esta operação de remoção (que inclui eventualmente a operação de limpeza) tem um custo amortizado logarítmico.