II. Introdução à Análise de Algoritmos

"Estudo de diferentes características que permitem classificar e comparar os algoritmos"

A **correcção** de um algoritmo é fundamental. A análise da correcção de algoritmos será a primeira que abordaremos.

Os recursos necessários à execução de um algoritmo:

- memória
- largura de banda
- *hardware*
- e particularmente: tempo de computação

permitem comparar os algoritmos quanto à sua **eficiência**.

A **estratégia** que um algoritmo adopta para desempenhar uma determinada tarefa ou resolver um determinado problema é também muito importante.

II. Análise de Algoritmos

Tópicos:

- Análise de correcção: invariantes de ciclo

- Análise do tempo de execução:
  - Modelo computacional
  - Análise do melhor caso, pior caso e caso médio
  - Análise de algoritmos recursivos – equações de recorrência
  - Notação assimptótica para o comportamento de funções
  - Análise de caso médio; algoritmos com aleatoriedade
  - Análise Amortizada

- Estratégias algorítmicas: incremental, divisão e conquista.

II. Análise de Algoritmos

Tópicos (continuação):

- Algoritmos de ordenação:
  - Algoritmos baseados em comparações: árvores de decisão
  - Limite inferior para o tempo de execução no *pior caso* de um algoritmo baseado em comparações
  - Algoritmos de ordenação em tempo linear

- Casos de estudo:
  - Pesquisa linear, pesquisa binária
  - "Insertion sort", "merge sort", "quicksort", "counting sort", "radix sort"
  - Algoritmos de Grafos: pesquisa; algoritmos de Prim e Dijkstra

O que é um algoritmo?

Um **algoritmo** é um procedimento computacional bem definido que aceita um valor (ou conjunto de valores) como **input** e produz um valor (ou conjunto de valores) como **output**.

Definições alternativas:

- uma sequência de passos computacionais que transformam um input num output.

- uma ferramenta para resolver um problema computacional bem definido; problema esse que define a relação I/O pretendida.

Em geral diz-se que um determinado input é uma *instância* do problema resolvido pelo algoritmo.

Importância dos Algoritmos

Áreas típicas onde são utilizados algoritmos:

- Internet: routing, searching, etc
- Criptografia
- Investigação Operacional
- programas com mapas e.g. trajectos
- Matemática
- . . .

Exemplos de problemas complicados:

- dado um mapa em que estão assinaladas as distâncias entre diversos pontos, encontrar o trajecto de menor distância entre cada par de pontos.
- dado um conjunto de matrizes, possivelmente não quadradas e de diferentes dimensões, encontrar a forma mais eficiente de calcular o seu produto.

Importância dos Algoritmos (cont.)

Na procura de um algoritmo que resolva um determinado problema, interessa em geral encontrar um que seja *eficiente*.

Há, no entanto, problemas para os quais não se conhece uma solução eficiente. Esta classe de problemas denomina-se por NP.

Os problemas NP-completos, uma subclasse dos anteriores são especialmente interessantes porque:

- aparentemente são simples
- não se sabe se existe um algoritmo eficiente que os resolva
- aplicam-se a áreas muito importantes
- se um deles for resolúvel de forma eficiente, todos os outros o serão
- por vezes, ao resolver um problema NP-completo, contentamo-nos em encontrar uma solução que *aproxime* a solução ideal, em tempo útil.

Importância dos Algoritmos (cont.)

Por que razão é importante saber conceber e analisar algoritmos?

- Apesar de haver já um grande número de problemas para os quais se conhecem algoritmos eficiente, **nem todos estão ainda resolvidos** e documentados. Importa saber conceber novos algoritmos.

- Se a memória fosse gratuita e a velocidade ilimitada, qualquer solução correcta seria igualmente válida. Mas no mundo real, a eficiência de um algoritmo é determinante: **a utilização de recursos assume grande importância**.

  Algoritmos para um mesmo problema podem variar grandemente em termos de eficiência: podem ser diferenças muito mais importantes do que as devidas ao hardware ou ao sistema operativo.

Os **Algoritmos são uma tecnologia** que importa pois dominar, uma vez que as escolhas efectuadas a este nível na concepção de um sistema podem vir a determinar a sua validade.

Correcção de um Algoritmo

Um algoritmo diz-se **correcto** se para todas as sequências de entrada, ele pára com o output correcto. Neste caso diz-se que ele **resolve** o problema computacional que lhe está associado.

Nem sempre a incorrecção é um motivo para a inutilidade de um algoritmo:

- Em certas aplicações basta que um algoritmo funcione correctamente para *alguns dos seus inputs*.

- Em problemas muito difíceis, poderá ser suficiente obter *soluções aproximadas* para o problema.

A análise da correcção de um algoritmo consiste em determinar se um algoritmo é correcto, e em que condições. Esta análise pode ser efectuada com um elevado grau de formalismo. Aqui vamos recorrer a uma técnica semi-formal baseada em provas indutivas utilizando *invariantes de ciclo*.

Análise de Correcção

A demonstração da correcção de um algoritmo cuja estrutura não apresente ciclos pode ser efectuada por simples inspeção. Exemplo:

```
int soma(int a, int b) {
  int sum;
  sum= a+b;
  return sum;
}
```

No caso de algoritmos recursivos, a prova advém da definição da solução. Exemplo:

```
int factorial(int n) {
  int f;
  if (n<1) { f=1; }
  else { f=n*factorial(n-1); }
  return f;
}
```

Análise de Correcção – Invariantes de Ciclo

No caso de algoritmos que incluam ciclos, uma prova de correcção implica uma análise da evolução de cada ciclo, em todas as suas iterações.

Um **invariante de ciclo** é uma propriedade que se mantém verdadeira durante toda a execução do ciclo e que reflecte todas as transformações do estado do algoritmo que esse ciclo efectua durante a sua execução.

Para utilizarmos um invariante numa prova de correcção temos de demonstrar

Inicialização O invariante verifica-se antes da primeira iteração

Preservação Se o invariante é válido antes de uma iteração, então é-o também depois dessa iteração

Terminação A execução do ciclo *pára*. No fim desta execução, o invariante corresponde a uma propriedade útil do algoritmo

Análise de Correcção – Exemplo 1

Problema Pesquisa da última ocorrência do valor k num vector.

```
int procura(int vector[], int a, int b, int k) {
  int i,f;
  f = -1;
  for (i=a; i<=b; i++) if (vector[i]==k) f = i;
  return f;
}
```

Invariante No início de cada iteração, a variável f terá o valor -1 caso o valor k não ocorra nas posições $[a \ldots (i-1)]$ do vector. Caso contrário, f terá o maior valor no intervalo $[a \ldots i-1]$ tal que $\texttt{vector}[f] = k$.

Inicialização

- Antes da primeira iteração $i = a$ e $f = -1$.
- O valor k não pode existir nas posições $[a \ldots (a-1)]$ do vector: o invariante verifica-se.

Análise de Correcção – Exemplo 1 (cont.)

Preservação

- Assume-se que o invariante é válido no início da iteração i.
- Se o valor k não estiver na posição i do array, o valor de f não será alterado. No início da iteração $i + 1$ a validade do invariante mantém-se também inalterada.
- Se o valor k estiver na posição i do vector, o valor de f será alterado para i. No início da iteração $i + 1$ o invariante verifica-se: f terá o valor da última posição onde k foi encontrado.

Terminação

- No final da execução do ciclo, $i = b + 1$.
- Se o invariante se mantém válido ao longo de toda a execução do ciclo, então:
  - se o valor k existe entre as posições $[a..b]$ do vector, então f terá o valor da última posição onde foi encontrado;
  - caso contrário, f terá o valor -1.
- Uma vez que o valor de f é devolvido no final do algoritmo, podemos dizer que este é correcto.

Análise de Correcção – Exemplo 2

Problema Pesquisa da primeira ocorrência do valor k num vector.

Versão ineficiente:

```
int procura(int vector[], int a, int b, int k) {
  int i,f ;
  i = a; f = -1;
  while (i<=b) {
    if ((f==-1) && (vector[i]==k)) f = i;
    i++;
  }
 return f;
}
```

Invariante No início de cada iteração, se o valor de f for -1 então k não ocorre nas posições $[a \ldots i-1]$ do vector; caso contrário, k ocorre na posição f e não ocorre nas posições $[a \ldots f-1]$.

Exemplo 2 (cont.)

Inicialização trivial . . .

Preservação Se no início da iteração i o valor de f for -1 podemos assumir que k não ocorre nas posições $[a \ldots i-1]$, e:

- se `vector[i]` contiver o valor k, então será executado `f = i`, e no início da próxima iteraçao, com $i' = i+1$, tem-se que k ocorre na posição f, e k não ocorre nas posições $[a \ldots f-1]$, uma vez que $f = i$.
- Cao contrário, no início da próxima iteraçao, com $i' = i+1$, tem-se que f tem ainda valor -1, e k não ocorre nas posições $[a \ldots i'-1]$.

Se no início da iteração i o valor de f for outro, então nesta iteração apenas i será actualizado, e no início da próxima iteraçao, com $i' = i+1$, f terá ainda o mesmo valor, pelo que o invariante é preservado.

Terminação O ciclo termina sempre com $i = b+1$. Se o valor de f for -1 então k não ocorre nas posições $[a \ldots b]$ do vector; caso contrário, k ocorre na posição f e não em posições inferiores.

Análise de Correcção – Exemplo 3

Problema Pesquisa da primeira ocorrência do valor k num vector.

```
int procura(int vector[], int a, int b, int k) {
  int i;
  i = a;
  while ((i<=b) && (vector[i]!=k))
    i++;
  if (i>b) return -1;
  else return i;
}
```

Invariante No início de cada iteração, `k` não se encontra nas posições $[a \ldots i]$ do vector.

Exercício Demonstrar a correcção do algoritmo.

Análise de Correcção – Exemplo 4

Problema Pesquisa *binária* de uma ocorrência de k num vector ordenado.

```
int procura(int vector[], int a, int b, int k) {
  int p;
  if (a>b) return -1
  do {
    p = (a+b)/2;
    if (vector[p]>=k) b = p;
    else a = p;
  } while (a<b);
  if (vector[a]==k) return a;
  else return -1;
}
```

Exercício Escrever o invariante de ciclo e demonstrar a correcção do algoritmo.

Análise do Tempo de Execução – Modelo Computacional

Este tipo de análise implica decisões:

- Qual a tecnologia a utilizar?
- Qual o custo de utilização dos recursos a ela associados?

Assumiremos neste curso:

- Um modelo denominado Random Access Machine (RAM).
- Um computador genérico, uniprocessador, sem concorrência, ou seja, as instruções são executadas sequencialmente.
- Algoritmos implementados como programas de computador.

Modelo Computacional

Não especificaremos detalhadamente o modelo RAM; no entanto seremos *razoáveis* na sua utilização. . .

Um computador real não possui instruções muito poderosas, como *sort*, mas sim operações básicas como:

- aritmética
- manipulação de dados (carregamento, armazenamento, cópia)
- controlo (execução condicional, ciclos, subrotinas)

Todas executam em *tempo constante*.

Modelo Computacional

A propósito de razoabilidade. . .

$\Rightarrow$ Será a instrução *exponenciação* de tempo constante?

O modelo computacional não entra em conta com a hierarquia de memória presente nos computadores modernos (*cache*, *memória virtual*) e que pode ser relevante na análise.

O modelo RAM é no entanto quase sempre bem sucedido.

Análise do Tempo de Execução

Dimensão do input depende do problema:

- ordenação de um vector: *número de posições do vector*
- multiplicação de inteiros: *número total de bits usados na representação dos números*
- pode consistir em mais do que um item: caso de um *grafo* (V, E)

Tempo de execução: número de operações primitivas executadas

Operações definidas de forma *independente* de qualquer máquina

$\Rightarrow$ Será uma *chamada de sub-rotina* (ou função em C) uma operação primitiva?

Exemplo: Pesquisa linear num vector

		Custo	n. Vezes
1	`int procura(int *v, int a, int b, int k) {`		
2	`  int i;`		
3	`  i=a;`	c1	1
4	`  while ((i<=b) && (v[i]!=k))`	c2	m+1
5	`  i++;`	c3	m
6	`  if (i>b)`	c4	1
7	`  return -1;`	c5	1
8	`  else return i; }`	c5	1

Onde m é o número de vezes que a instrução na linha 5 é executada.

Este valor dependerá de quantas vezes a condição de guarda do ciclo é satisfeita: $0 \leq m \leq b - a + 1$.

Tempo Total de Execução

$$T(N) = c_1 + c_2(m+1) + c_3 m + c_4 + c_5$$

Para determinado tamanho fixo $N = b - a + 1$ da sequência a pesquisar – o *input* do algoritmo – o tempo total $T(N)$ pode variar com o conteúdo.

Melhor Caso: valor encontrado na primeira posição do vector

$T(N) = (c_1 + c_2 + c_4 + c_5)$, donde $T(N)$ é constante.

Pior Caso: valor não encontrado

$$T(N) = c_1 + c_2(N+1) + c_3(N) + c_4 + c_5 = (c_2 + c_3)N + (c_1 + c_2 + c_4 + c_5)$$

logo $T(N)$ é função *linear* de N

Outro exemplo: Pesquisa de duplicados num vector

		Custo	n. Vezes
1	`void dup(int *v,int a,int b) {`		
2	`  int i,j;`		
3	`  for (i=a;i<b;i++)`	c1	N
4	`  for (j=i+1;j<=b;j++)`	c2	S1
5	`  if (v[i]==v[j])`	c3	S2
6	`  printf("%d igual a %d\n",i,j);`	c4	S2
7	`}`		

onde

$$N = b - a + 1 \text{ (dimensão do input)}; \quad S_1 = \sum_{i=a}^{b-1}(n_i + 1); \quad S_2 = \sum_{i=a}^{b-1} n_i$$

e $n_i = b - i$ é o número de vezes que o ciclo interior é executado, para cada i.

Tempo Total de Execução

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

Neste algoritmo, o melhor caso e o pior caso são iguais: para qualquer vector de entrada de tamanho N, os ciclos são executados o mesmo número de vezes.

Simplifiquemos, considerando $a = 1, b = N$. Então

$$S_2 = \sum_{i=1}^{N-1} N - i = (N-1)N - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

$$S_1 = \sum_{i=1}^{N-1} (N - i + 1) = (N-1)(N+1) - \frac{(N-1)N}{2} = \frac{1}{2}N^2 + \frac{1}{2}N - 1$$

$T(N)$ é pois uma função *quadrática* do tamanho do input:

$$T(N) = k_2 N^2 + K_1 N + K_0$$

Algumas Considerações

Simplificação da Análise:

- utilizámos *custos abstractos* c_i em vez de tempos concretos

E simplificaremos ainda mais: veremos que para N suficientemente grande o termo quadrático anula os restantes, logo o tempo de execução de *dup* pode ser aproximado por:

$$T(N) = kN^2$$

E de facto nem a constante é relevante face ao termo N^2. Diremos simplesmente que o algoritmo tem um tempo de execução de

$$\Theta(N^2)$$

Análise de Pior Caso e Caso Médio

("worst case" / "average case")

Análise de pior caso é útil:

- limite superior para *qualquer input* – uma garantia!
- pior caso ocorre frequentemente (e.g. pesquisa de informação não existente)
- muitas vezes caso médio é próximo do pior caso! (veremos exemplos)

Análise de caso médio ou esperado: idealmente involveria estudo probabilístico sobre a dimensão e natureza do *input* para um determinado problema (e.g. vectores com elevado grau de ordenação? Com que tamanho médio?)

Em geral assumiremos que para determinada dimensão todos os inputs ocorrem com igual probabilidade (na ordenação de um vector todos os graus de ordenação prévia são igualmente prováveis)

Notação Assimptótica – Comportamento de Funções

Tempo de execução de um algoritmo expresso como função do tamanho do input (em geral um número em $\mathbf{N} = \{0, 1, 2, \ldots\}$):

$$T(N) = k_2 N^2 + k_1 N + k_0$$

Normalmente o tempo *exacto* não é importante: para dados de entrada de elevada dimensão as constantes multiplicativas e os termos de menor grau são anulados (basta uma pequena fracção do termo de maior grau).

Então apenas a *ordem de crescimento* do tempo de execução é relevante e estudamos o *comportamento assimptótico dos algoritmos*.

Se o algoritmo A_1 é assimptoticamente melhor do que A_2, A_1 será melhor escolha do que A_2 excepto para inputs muito pequenos.

Consideraremos apenas funções *assimptoticamente não-negativas*.

Notação Θ

Para uma função $g(n)$ de domínio $\mathbf{N}$ define-se $\Theta(g(n))$ como o seguinte *conjunto de funções*:

$$\Theta(g(n)) = \{f(n) \mid \text{existem } c_1, c_2, n_0 > 0 \text{ tais que } \forall n \geq n_0$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Abuso de linguagem: $f(n) = \Theta(g(n))$ em vez de $f(n) \in \Theta(g(n))$.

Para $n \geq n_0$, $f(n)$ é igual a $g(n)$ a menos de um factor constante

Determinação da classe Θ de funções polinomiais

Basta ignorar os termos de menor grau e o coeficiente do termo de maior grau:

$$7n^2 - 2n = \Theta(n^2)$$

De facto terá de ser para $\forall n \geq n_0$:

$$c_1 n^2 \leq 7n^2 - 2n \leq c_2 n^2$$

$$c_1 \leq 7 - \frac{2}{n} \leq c_2$$

Basta escolher, por exemplo, $n \geq 10$, $c_1 \leq 6$, $c_2 \geq 8$

(constantes dependem da função)

Determinação da classe Θ

Pode-se demonstrar por redução ao absurdo que $6n^3 \neq \Theta(n^2)$.

Se existissem c_2, n_0 tais que $\forall n \geq n_0$,

$$6n^3 \leq c_2 n^2$$

$$n \leq \frac{c_2}{6}$$

o que é impossível sendo c_2 constante e n arbitrariamente grande.

Exercício:

$\Rightarrow$ mostrar para $f(n) = an^2 + bn + c$ que $f(n) = \Theta(n^2)$

Notação O ("big oh")

Para uma função $g(n)$ de domínio $\mathbf{N}$ define-se $O(g(n))$ como o seguinte *conjunto de funções*:

$$O(g(n)) = \{f(n) \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0$$

$$0 \leq f(n) \leq cg(n)\}$$

Para $n \geq n_0$, $g(n)$ é um limite superior de $f(n)$ a menos de um factor constante

Observe-se que $\Theta(g(n)) \subseteq O(g(n))$, logo

$$f(n) = \Theta(g(n)) \text{ implica } f(n) = O(g(n)).$$

Exemplo: $3n^2 + 7n = O(n^2)$, mas também $4n - 5 = O(n^2)$ [$\Rightarrow$ porquê?]

Notação Ω

Para uma função $g(n)$ de domínio $\mathbf{N}$ define-se $\Omega(g(n))$ como o seguinte *conjunto de funções*:

$$\Omega(g(n)) = \{f(n) \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0$$

$$0 \leq cg(n) \leq f(n)\}$$

Para $n \geq n_0$, $g(n)$ é um limite inferior de $f(n)$ a menos de um factor constante

Teorema 1. *Para quaisquer duas funções* $f(n)$, $g(n)$,

$$f(n) = \Theta(g(n)) \textit{ sse } f(n) = O(g(n)) \textit{ e } f(n) = \Omega(g(n))$$

$\Rightarrow$ Qual poderá ser a utilidade deste teorema?

Abusos de Linguagem. . .

$n = O(n^2)$	$n \in O(n^2)$
$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$	$2n^2 + 3n + 1 = 2n^2 + f(n)$ com $f(n) = \Theta(n)$
$2n^2 + \Theta(n) = \Theta(n^2)$	para qualquer $f(n) = \Theta(n)$ existe $g(n) = \Theta(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$

Propriedades das Notações Θ, O, Ω

Transitividade $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ implica $f(n) = \Theta(h(n))$

Reflexividade $f(n) = \Theta(f(n))$

[Ambas válidas também para O e Ω]

Simetria $f(n) = \Theta(g(n))$ sse $g(n) = \Theta(f(n))$

Transposição $f(n) = O(g(n))$ sse $g(n) = \Omega(f(n))$

Funções Úteis em Análise de Algoritmos

Uma função $f(n)$ diz-se limitada:

- *polinomialmente* se $f(n) = O(n^k)$ para alguma constante k.
- *polilogaritmicamente* se $f(n) = O(\log_a^k n)$ para alguma constante k [notação: $\lg = \log_2$].

Algumas relações úteis (a função da direita cresce mais depressa):

$$\begin{array}{rcll}
\log_b n & = & O(\log_a n) & [a, b > 1] \\
n^b & = & O(n^a) & \text{se } b \leq a \\
b^n & = & O(a^n) & \text{se } b \leq a \\
 & & & \\
\log^b n & = & O(n^a) & \\
n^b & = & O(a^n) & [a > 1] \\
n! & = & O(n^n) & \\
n! & = & \Omega(2^n) & \\
\lg(n!) & = & \Theta(n \lg n) &
\end{array}$$

Classificação de Algoritmos

Para além da classe de complexidade e das propriedades de correcção, os algoritmos podem também ser estudados – e classificados – relativamente à categoria de problemas a que dizem respeito:

- Pesquisa
- Ordenação – vários algoritmos serão estudados com detalhe
- Processamento de strings (parsing)
- Problemas de grafos – segundo capítulo do programa
- Problemas combinatoriais
- Problemas geométricos
- Problemas de cálculo numérico.
- ...

Classificação de Algoritmos (cont.)

Uma outra forma de classificar os algoritmos é de acordo com a estratégia que utilizam para alcançar uma solução:

- Incremental (iterativa) – veremos como exemplo o *insertion sort*.
- Divisão e Conquista (*divide-and-conquer*) – veremos como exemplos os algoritmos *mergesort* e *quicksort*.
- Algoritmos Gananciosos (*greedy*) – veremos alguns algoritmos de grafos e.g. Minimum Spanning Tree (Árvore Geradora Mínima).
- Programação Dinâmica – veremos o algoritmo de grafos *All-Pairs-Shortest-Path*.
- Algoritmos com aleatoriedade ou probabilísticos – veremos uma versão modificada do algoritmo *quicksort*.

Caso de Estudo 1: Algoritmo "Insertion Sort"

Problema: ordenação (crescente) de uma sequência de números inteiros. O problema será resolvido com a sequência implementada como um *vector* ("array") que será *reordenado* ($\neq$ construção de uma nova sequência).

Tempo de execução depende dos dados de entrada:

- dimensão
- "grau" prévio de ordenação

Utiliza uma **estratégia incremental** ou iterativa para resolver o problema:

- começa com uma lista ordenada vazia,
- onde são gradualmente inseridos, de forma ordenada os elementos da lista original.

"Insertion Sort"

A sequência a ordenar está disposta entre as posições 1 e N do vector A.

```
void insertion_sort(int A[]) {
  for (j=2 ; j<=N ; j++) {
    key = A[j];
    i = j-1;
    while (i>0 && A[i] > key) {
      A[i+1] = A[i];
      i--;
    }
    A[i+1] = key;
  }
}
```

Análise de Correcção – Invariante de Ciclo

Invariante de ciclo: no início de cada iteração do ciclo `for`, o vector contém entre as posições 1 e $j-1$ os valores iniciais, já ordenados.

$\Rightarrow$ Verificação da *Preservação* obrigaria a estabelecer e demonstrar a validade de um novo invariante para o ciclo *while*:

No início de cada iteração do ciclo interior, a região $A[i+2, ..., j]$ *contém, pela mesma ordem, os valores inicialmente na região* $A[i+1, ..., j-1]$. *O valor da variável* key *é inferior a todos esses valores.*

$\Rightarrow$ *Terminação* [j=n+1] corresponde ao objectivo desejado: vector está ordenado.

Análise do Tempo de Execução

```
                                          Custo        n. Vezes
void insertion_sort(int A[]) {
  for (j=2 ; j<=N ; j++) {                  c1            N
    key = A[j];                             c2          N-1
    i = j-1;                                c3          N-1
    while (i>0 && A[i] > key) {             c4           S1
      A[i+1] = A[i];                        c5           S2
      i--;                                  c6           S2
    }
    A[i+1] = key;                           c7          N-1
  }
}
```

onde $S_1 = \sum_{j=2}^{N} n_j$; $S_2 = \sum_{j=2}^{N} (n_j - 1)$ onde n_j é o número de vezes que o *teste* do ciclo `while` é efectuado, para cada valor de j

Tempo Total de Execução

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4 S_1 + c_5 S_2 + c_6 S_2 + c_7(N-1)$$

Para determinado tamanho N da sequência a ordenar – o *input* do algoritmo – o tempo total $T(n)$ pode variar com o *grau de ordenação prévia* da sequência:

Melhor Caso: sequência está ordenada à partida

$n_j = 1$ para $j = 2, \ldots, N$; logo $S_1 = N - 1$ e $S_2 = 0$;

$$T(N) = (c_1 + c_2 + c_3 + c_4 + c_7)N - (c_2 + c_3 + c_4 + c_7)$$

$T(N)$ é função *linear* de N.

No melhor caso, temos então um tempo de execução em $\Theta(N)$.

Tempo Total de Execução

Pior Caso: sequência previamente ordenada por *ordem inversa* (decrescente)

$n_j = j$ para $j = 2, \ldots, N$; logo

$$S1 = \sum_{j=2}^{N} j = \frac{N(N+1)}{2} - 1 \quad \text{e} \quad S2 = \sum_{j=2}^{N} (j-1) = \frac{N(N-1)}{2}$$

$$T(N) = (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})N^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)N - (c_2 + c_3 + c_4 + c_7)$$

$T(N)$ é função *quadrática* de N

O tempo de execução no pior caso está pois em $\Theta(N^2)$.

$\Rightarrow$ Alternativamente, podemos dizer que o tempo de execução do algoritmo está (em qualquer caso) em $\Omega(N)$ e em $\mathcal{O}(N^2)$.

Caso de Estudo 2: Algoritmo "Merge Sort"

Utiliza uma estratégia do tipo *Divisão e Conquista*:

1. **Divisão** do problema em n sub-problemas

2. **Conquista**: resolução dos sub-problemas:

   - trivial se tamanho muito pequeno.
   - utilizando a mesma estratégia no caso contrário.

3. **Combinação** das soluções dos sub-problemas

   implementação típica é recursiva ($\Rightarrow$ porquê?)

Divisão e Conquista – "Merge Sort"

1. **Divisão** do vector em dois vectores de dimensões similares

2. **Conquista**: ordenação recursiva dos dois vectores usando *merge sort*. Nada a fazer para vectores de dimensão 1!

3. **Combinação**: fusão dos dois vectores ordenados. Será implementado por uma função auxiliar `merge`.

   Função `merge` recebe as duas sequências `A[p..q]` e `A[q+1..r]` já ordenadas.

   No fim da execução da função, a sequência `A[p..r]` está ordenada.

Função Auxiliar de *Fusão*:

```
void merge(int A[], int p, int q, int r) {
  int L[MAX], R[MAX];
  int n1 = q-p+1;
  int n2 = r-q;
  for (i=1 ; i<=n1 ; i++)  L[i] = A[p+i-1];
  for (j=1 ; j<=n2 ; j++)  R[j] = A[q+j];
  L[n1+1] = MAXINT; R[n2+1] = MAXINT;
  i = 1; j = 1;
  for (k=p ; k<=r ; k++)
    if (L[i] <= R[j]) {
      A[k] = L[i]; i++;
    } else {
      A[k] = R[j]; j++;
   }
}
```

Função `merge`: Observações

- Passo básico: comparação dos dois valores contidos nas primeiras posições de ambos os vectores, colocando o menor dos valores no vector final.

- Cada passo básico é executado em *tempo constante* ($\Rightarrow$ porquê?)

- A dimensão do input é $n = r - p + 1$, e nunca poderá haver mais do que n passos básicos.

- Este algoritmo executa então em *tempo linear*: `merge` $= \Theta(n)$.

  $\Rightarrow$ Poderá ser demonstrado com mais rigor?

- $\Rightarrow$ Qual o papel das *sentinelas* de valor `MAXINT`?

Exercício – Correcção de `merge`

O terceiro ciclo `for` implementa os n passos básicos mantendo o seguinte invariante de ciclo:

1. No início de cada iteração, o subvector `A[p..k-1]` contém, *ordenados*, os $k - p$ menores elementos de `L[1..n1+1]` e `R[1..n2+1]`;

2. `L[i]` e `R[j]` são os menores elementos nos respectivos vectores que ainda não foram copiados para `A`.

$\Rightarrow$ Verifique as propriedades de *inicialização*, *preservação*, e *terminação* deste invariante

A última propriedade deve permitir provar a *correcção* do algoritmo, i.e, no fim da execução do ciclo o vector `A` contém a fusão de `L` e `R`.

"Merge Sort"

```
void merge_sort(int A[], int p, int r)
{
  if (p < r) {
    q = (p+r)/2;
    merge_sort(A,p,q);
    merge_sort(A,q+1,r);
    merge(A,p,q,r);
  }
}
```

$\Rightarrow$ Qual a dimensão de cada sub-sequência criada no passo de divisão?

Invocação inicial:

```
merge_sort(A,1,n);
```

em que `A` contém `n` elementos.

Análise de Pior Caso

Simplificação da análise de "merge sort": tamanho do input é uma potência de 2. Em cada **divisão**, as sub-sequências têm tamanho *exactamente* $= n/2$.

Seja $T(n)$ o tempo de execução (no pior caso) sobre um input de tamanho n. Se $n = 1$, esse tempo é constante, que escrevemos $T(n) = \Theta(1)$. Senão:

1. **Divisão**: o cálculo da posição do meio do vector é feita em *tempo constante*: $D(n) = \Theta(1)$

2. **Conquista**: são resolvidos dois problemas, cada um de tamanho $n/2$; o tempo total para isto é $2T(n/2)$

3. **Combinação**: a função `merge` executa em tempo linear: $C(n) = \Theta(n)$

Então:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ \Theta(1) + 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Equações de Recorrência (Fibonacci, 1202!)

A análise de algoritmos recursivos exige a utilização de um instrumento que permita exprimir o tempo de execução sobre um input de tamanho n em função do tempo de execução sobre inputs de tamanhos inferiores.

Em geral num algoritmo de divisão e conquista:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq k \\ D(n) + aT(n/b) + C(n) & \text{se } n > k \end{cases}$$

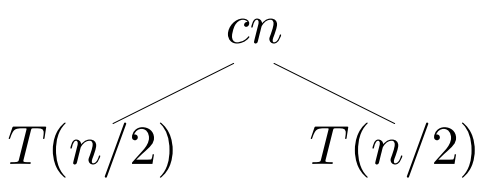
Em que cada **divisão** gera a sub-problemas, sendo o tamanho de cada sub-problema uma fracção $1/b$ do original (pode ser $b \neq a$).

Construção da Árvore de Recursão – 1^o Passo

Reescrevamos a relação de recorrência:

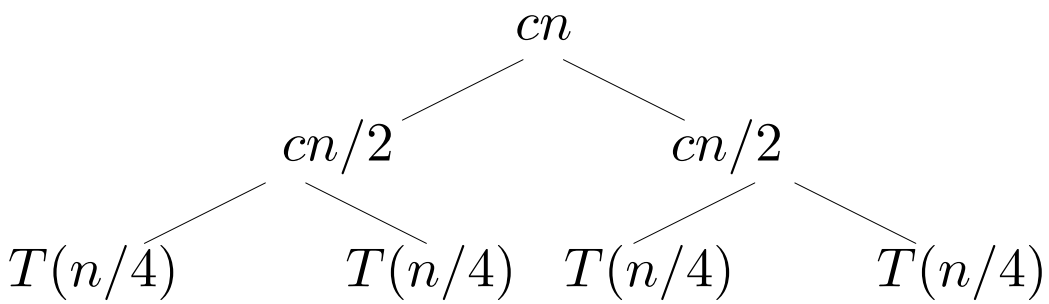
$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

em que c é o maior entre os tempos necessário para resolver problemas de dimensão 1 e o tempo de combinação por elemento dos vectores.

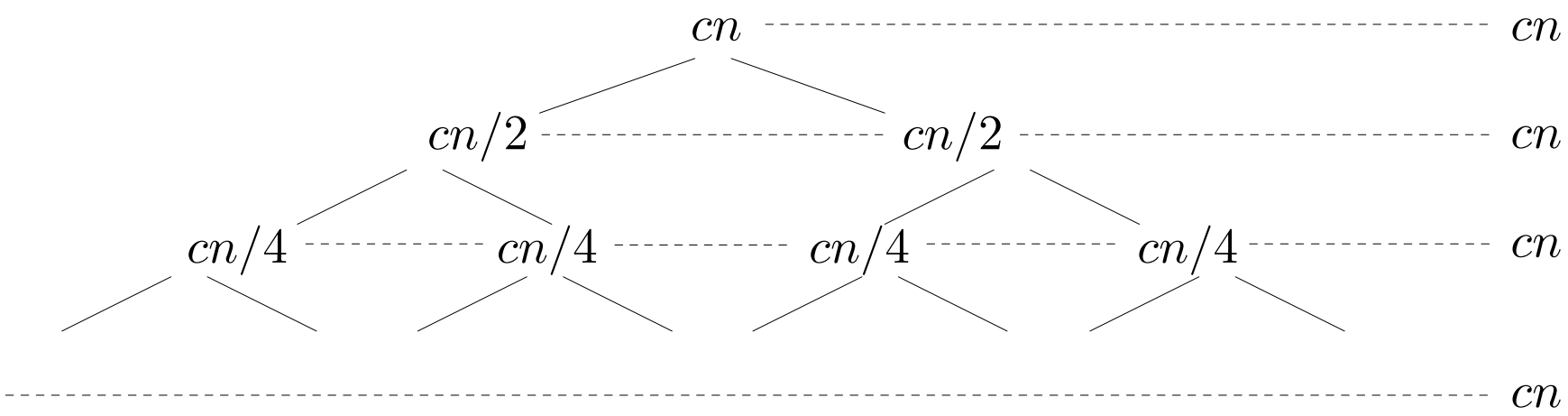


Construção da Árvore de Recursão – 2^o Passo

$T(n/2) = 2T(n/4) + cn/2$:



Árvore de Recursão



- árvore final tem $\lg n + 1$ níveis ($\Rightarrow$ provar por indução)
- custo total de cada nível é constante, $= cn$
- então o custo total é $(\lg n + 1)cn = cn \lg n + cn$

Então o algoritmo "merge sort" executa no pior caso em $T(n) = \Theta(n \lg n)$

Um Pormenor . . .

Admitimos inicialmente que o tamanho da sequência era uma potência de 2.

Para valores arbitrários de n a recorrência correcta é:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T\lceil n/2 \rceil + T\lfloor n/2 \rfloor + \Theta(n) & \text{se } n > 1 \end{cases}$$

A solução de uma recorrência pode ser *verificada* pelo *Método da Substituição*, que permite provar que a recorrência acima tem também como solução

$$T(n) = \Theta(n \lg n)$$

Método da Substituição

- Utiliza *indução* para provar limites *inferiores* ou *superiores* para a solução de recorrências

- Implica a obtenção prévia de uma solução por um método aproximado (tipicamente por observação da árvore de recursão)

  Exemplo: seja a recorrência

$$T(n) = 2T\lfloor n/2 \rfloor + n$$

Pela sua semelhança com a recorrência do "merge sort" podemos adivinhar:

$$T(n) = \Theta(n \lg n)$$

Utilizemos o método para provar o limite superior $T(n) = O(n \lg n)$.

Método da Substituição

Para $T(n) = 2T\lfloor n/2 \rfloor + n$ desejamos provar $T(n) = O(n \lg n)$, i.e,

$$T(n) \leq cn \lg n$$

para um determinado valor de $c > 0$.

1. Assumimos que o limite superior se verifica para $\lfloor n/2 \rfloor$:

$$T\lfloor n/2 \rfloor \leq c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$$

2. Substituindo na recorrência, obtemos um limite superior para $T(n)$:

$$T(n) \leq 2(c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor) + n$$

3. Simplificação: $\lfloor n/2 \rfloor \leq n/2$, e $\lg$ é uma função crescente, logo:

$$\begin{aligned} T(n) &\leq 2c(n/2)\lg(n/2) + n \\ &= cn\lg(n/2) + n \\ &= cn(\lg n - \lg 2) + n \\ &= cn\lg n - cn + n \end{aligned}$$

4. Para $c \geq 1$, terminamos o caso indutivo:

$$T(n) \leq cn\lg n$$

5. Falta provar o caso de base.

Observe-se que a recorrência que estamos a analisar foi definida sem um caso limite. Admitamos a seguinte definição completa:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(n) = 2T\lfloor n/2 \rfloor + n & \text{se } n > 1 \end{cases}$$

Esta definição não parece fornecer um caso de paragem adequado para a nossa prova indutiva de $T(n) \leq cn \lg n$:

$$T(1) \leq c1 \lg 1 = 0 \qquad \Leftarrow \text{Falso!!}$$

No entanto a notação assimptótica [pg. 31] permite contornar o problema. Para provar $T(n) = O(n \lg n)$ basta provar que existem $c, n_0 > 0$ tais que $T(n) \leq cn \lg n$ se verifica para $n \geq n_0$.

Tentemos então substituir o caso-base $n = 1$ por outro, começando por considerar $n_0 = 2$. Temos que $T(2) = 4 \leq c2 \lg 2 = 2c$. Basta escolher $c \geq 2$.

Mas teremos que ser cuidadosos. Calculemos:

$$
\begin{aligned}
T(2) &= 2T\lfloor 2/2 \rfloor + 2 = 2T(1) + 2 = 4 \\
T(3) &= 2T\lfloor 3/2 \rfloor + 3 = 2T(1) + 3 = 5 \\
T(4) &= 2T\lfloor 4/2 \rfloor + 4 = 2T(2) + 4 = 12 \\
T(5) &= 2T\lfloor 5/2 \rfloor + 5 = 2T(2) + 5 = 13 \\
T(6) &= 2T\lfloor 6/2 \rfloor + 6 = 2T(3) + 6 = 16
\end{aligned}
$$

Vemos que $T(2)$ e $T(3)$ dependem ambos de $T(1)$, pelo que $n = 1$ não pode ser simplesmente substituído por $n = 2$ como caso-base do raciocínio indutivo, mas sim pelos dois casos $n = 2, n = 3$. Verifiquemos então para $n = 3$:

$$T(3) = 5 \leq c3 \lg 3 = 4.8c$$

Escolhendo $c \geq 2$ verifica-se também esta condição, pelo que terminamos assim a prova indutiva.

Mudanças de Variável

Permitem transformar recorrências por forma a torná-las "reconhecíveis". Seja:

$$T(n) = 2T\lfloor\sqrt{n}\rfloor + \lg n$$

E efectuemos a mudança de variável $m = \lg n$:

$$T(2^m) = 2T(\sqrt{2^m}) + m = 2T(2^{m/2}) + m$$

E seja $T'(m) = T(2^m)$:

$$T'(m) = 2T'(m/2) + m$$

Reconhecemos uma recorrência com solução $T'(m) = O(m \lg m)$, ou seja:

$$T(n) = O(m \lg m) = O(\lg n . \lg(\lg n))$$

"Adivinhar Soluções"

A recorrência $T(n) = 2T(\lfloor n/2 \rfloor + 100) + n$ é também $O(n \lg n)$

$\Rightarrow$ Porquê? Provar.

$\Rightarrow$ Provar solução da recorrência exacta do "merge sort".

Caso de Estudo 3: Algoritmo "Quicksort"

Usa também uma estratégia de **divisão e conquista**:

1. **Divisão**: *partição* do vector `A[p..r]` em dois sub-vectores `A[p..q-1]` e `A[q+1..r]` tais que todos os elementos do primeiro (resp. segundo) são $\leq$ `A[q]` (resp. $\geq$ `A[q]`)

   - os sub-vectores são possivelmente vazios
   - cálculo de `q` faz parte do processo de partição

   Função `partition` recebe a sequência `A[p..r]`, executa a sua partição "in place" usando o último elemento do vector como **pivot** e devolve o índice `q`.

2. **Conquista**: ordenação recursiva dos dois vectores usando *quicksort*. Nada a fazer para vectores de dimensão 1.

3. **Combinação**: nada a fazer!

Função de Partição

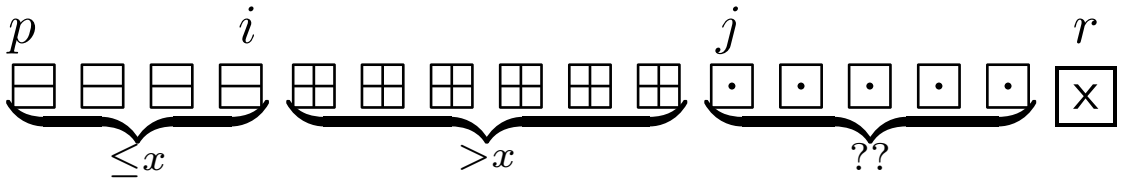
```
int partition (int A[], int p, int r)
{
  x = A[r];
  i = p-1;
  for (j=p ; j<r ; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j);
    }
  swap(A, i+1, r);
  return i+1;
}

void swap(int X[], int a, int b)
{ aux = X[a]; X[a] = X[b]; X[b] = aux; }
```

Função de partição executa em tempo linear $D(n) = \Theta(n)$.

Análise de Correcção – Invariante

No início de cada iteração do ciclo `for` tem-se para qualquer posição k do vector:

1. Se $p \leq k \leq i$ então $A[k] \leq x$;
2. Se $i+1 \leq k \leq j-1$ então $A[k] > x$;
3. Se $k = r$ então $A[k] = x$.



$\Rightarrow$ Verificar as propriedades de *inicialização* ($j = p$, $i = p-1$), *preservação*, e *terminação* ($j = r$)

$\Rightarrow$ o que fazem as duas últimas instruções?

Algoritmo "Quicksort"

```
void quicksort(int A[], int p, int r)
{
  if (p < r) {
    q = partition(A,p,r)
    quicksort(A,p,q-1);
    quicksort(A,q+1,r);
  }
}
```

A recorrência correspondente a este algoritmo é:

$$T(n) = D(n) + T(k) + T(k') + C(n)$$

sendo $D(n) = \Theta(n)$ e $C(n) = 0$; $k' = n - k - 1$

$$T(n) = \Theta(n) + T(k) + T(n - k - 1)$$

Análise de Pior Caso

$$T(n) = \Theta(n) + \max_{k=0}^{n-1} \left(T(k) + T(n-k-1)\right), \quad \text{com } k \text{ entre } 0 \text{ e } n-1$$

Admitamos $T(n) \leq cn^2$; temos por substituição:

$$\begin{aligned} T(n) &\leq \Theta(n) + \max\left(ck^2 + c(n-k-1)^2\right) \\ &= \Theta(n) + c\max\left(k^2 + (n-k-1)^2\right) \\ &= \Theta(n) + c\max(\underbrace{2k^2 + (2-2n)k + (n-1)^2}_{P(k)}) \end{aligned}$$

por análise de $P(k)$ conclui-se que os máximos no intervalo $0 \leq k \leq n-1$ se encontram nas extremidades, com valor $P(0) = P(n-1) = (n-1)^2$.

O pior caso ocorre então quando a partição produz um vector com 0 elementos e outro com $n-1$ elementos.

Continuando o raciocínio:

$$T(n) \leq \Theta(n) + c(n^2 - 2n + 1) = \Theta(n) + cn^2$$

logo temos uma prova indutiva de $T(n) = O(n^2)$. Mas será isto apenas um limite superior para o pior caso ou será também neste caso $T(n) = \Theta(n^2)$?

Basta considerar o caso em que *em todas as invocações recursivas* a partição produz vectores de dimensões 0 e $n - 1$ para se ver que este tempo de pior caso ocorre mesmo na prática:

$$T_p(n) = \Theta(n) + T_p(n-1) + T_p(0) = \sum_{i=0}^{n} \Theta(i) = \Theta(n^2)$$

Temos então $T_p(n) = \Theta(n^2)$.

Análise de Tempo de Execução de "quicksort"

- No pior caso "quicksort" executa em $\Theta(n^2)$, tal como "insertion sort", mas este pior caso ocorre quando a sequência de entrada se encontra já ordenada $\Rightarrow$ (Porquê?), caso em que "insertion sort" executa em tempo $\Theta(n)$!

- Análise de *melhor caso*: partição produz vectores de dimensão $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil - 1$

$$T(n) \leq \Theta(n) + 2T(n/2)$$

  com solução $T_m(n) = \Theta(n \lg n)$.

- Contrariamente à situação mais comum, o caso médio de execução de "quicksort" aproxima-se do melhor caso, e não do pior.

- Basta construir a árvore de recursão admitindo por exemplo que a função de partição produz *sempre* vectores de dimensão $1/10$ e $9/10$ do original.

- Apesar de aparentemente má, esta situação produz $T_m(n) = \Theta(n \lg n)$.

Análise de Caso Médio de "quicksort"

- Numa execução de "quick sort" são efectuadas n invocações da função de partição. $\Rightarrow$ (porquê?)

- Em geral o tempo total de execução é $T(n) = O(n + X)$, onde X é o número *total* de *comparações* efectuadas.

- É necessária uma análise detalhada, probabilística, do caso médio, para determinar o *valor esperado* de X.

- Numa situação "real" a função de partição não produzirá sempre vectores com as mesmas dimensões relativas . . .

Análise de Caso Médio

- *Análise Probabilística* implica a utilização de uma *distribuição* sobre o input.
- Por exemplo no caso da ordenação, deveríamos conhecer a probabilidade de ocorrência de cada permutação possível dos elementos do vector de entrada.
- Quando é irrealista ou impossível assumir algo sobre os inputs, pode-se *impor* uma distribuição uniforme, por exemplo permutando-se previamente de forma aleatória o vector de entrada.
- Desta forma asseguramo-nos de que todas as permutações são igualmente prováveis.
- Num tal algoritmo com aleatoriedade, nenhum input particular corresponde ao pior ou ao melhor casos; apenas o processamento prévio (aleatório) pode gerar um input pior ou melhor.

Algoritmo "quicksort" com aleatoriedade

Em vez de introduzir no algoritmo uma rotina de permutação aleatória do vector de entrada, usamos a técnica de *amostragem aleatória*.

O *pivot* é (em cada invocação) escolhido de forma aleatória de entre os elementos do vector. Basta usar a seguinte versão da função de partição:

```
int randomized-partition (int A[], int p, int r)
{
  i = generate_random(p,r)      /* número aleatório entre p e r */
  swap(A,r,i);
  return partition(A,p,r);
}
```

Este algoritmo pode depois ser analisado com ferramentas probabilísticas (fora do âmbito deste curso).

Algoritmos de Ordenação Baseados em Comparações

Todos os algoritmos estudados até aqui são baseados em comparações: dados dois elementos `A[i]` e `A[j]`, é efectuado um teste (e.g. `A[i]<=A[j]`) que determina a ordem relativa desses elementos. Assumiremos agora que

- um tal algoritmo não usa qualquer outro método para obter informação sobre o valor dos elementos a ordenar;

- a sequência não contém elementos repetidos.

A execução de um algoritmo baseado em comparações sobre sequências de uma determinada dimensão pode ser vista de forma abstracta como uma *Árvore de Decisão*.

Nesta árvore, cada nó:

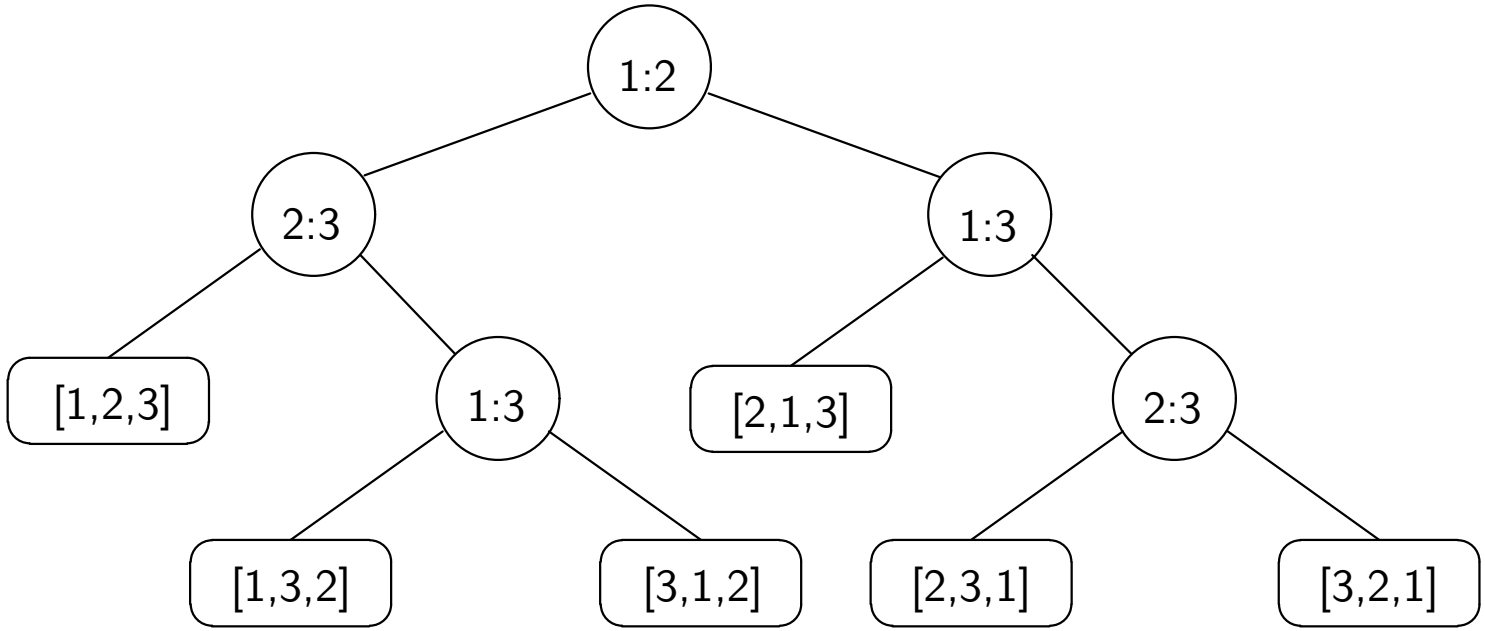
- corresponde a um teste de comparação entre dois elementos da sequência;
- tem como sub-árvore esquerda (resp. direita) a árvore correspondente à continuação da execução do algoritmo caso o teste resulte verdadeiro (resp. falso).

Cada folha corresponde a uma ordenação possível do input; *todas* as permutações da sequência devem aparecer como folhas. ($\Rightarrow$ Porquê?)

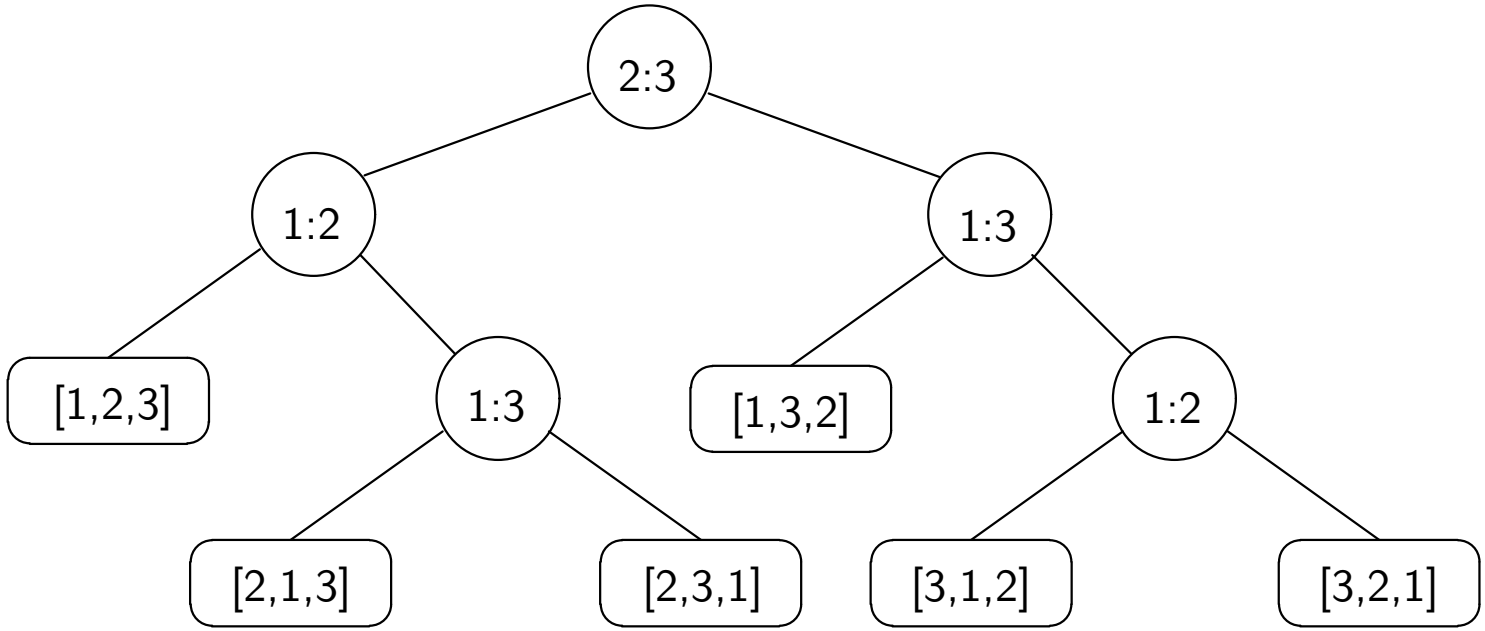
A execução concreta do algoritmo para um determinado vector de input corresponde a um *caminho da raíz para uma folha*, ou seja, uma sequência de comparações.

O pior caso de execução de um algoritmo de ordenação baseado em comparações corresponde ao caminho mais longo da raíz para uma folha. O número de comparações efectuadas é dado neste caso pela *altura* da árvore.

Exemplo – Árv. de Decisão para "insertion sort", $N = 3$



Exemplo – Árv. de Decisão para "merge sort", $N = 3$



Um Limite Inferior para o Pior Caso . . .

Teorema. *A altura h de uma árvore de decisão tem o seguinte limite mínimo:*

$$h \geq \lg(N!) \qquad \textit{com } N \textit{ a dimensão do input}$$

Prova. *Em geral uma árvore binária de altura h tem no máximo 2^h folhas. As árvores que aqui consideramos têm $N!$ folhas correspondentes a todas as permutações do input, pelo que (cfr. pg. 35)*

$$N! \leq 2^h \quad \Rightarrow \quad \lg(N!) \leq h$$

Corolário. *Seja $T(N)$ o tempo de execução no pior caso de qualquer algoritmo de ordenação baseado em comparações. Então*

$$T(N) = \Omega(N \lg N)$$

"Merge sort" é assimptoticamente óptimo uma vez que é $O(N \lg N)$.

Algoritmo "Counting Sort"

- Assume-se que *os elementos de* `A[]` *estão contidos no intervalo entre 0 e* k (com k conhecido).
- O algoritmo baseia-se numa *contagem*, para cada elemento x da sequência a ordenar, do número de elementos inferiores ou iguais a x.
- Esta contagem permite colocar cada elemento directamente na sua posição final. $\Rightarrow$ Porquê?
- Algoritmo produz novo vector (*output*) e utiliza vector auxiliar temporário.

Algoritmo não efectua comparações, o que permite quebrar o limite $\Omega(N \lg N)$.

Algoritmo "Counting Sort"

```
void counting_sort(int A[], int B[], int k) {
  int C[k+1];
  for (i=0 ; i<=k ; i++)          /* inicialização de C[] */
    C[i] = 0;
  for (j=1 ; j<=N ; j++)          /* contagem ocorr. A[j] */
    C[A[j]] = C[A[j]]+1;
  for (i=1 ; i<=k ; i++)          /* contagem dos <= i    */
    C[i] = C[i]+C[i-1];
  for (j=N ; j>=1 ; j--) {        /* construção do        */
    B[C[A[j]]] = A[j];            /* vector ordenado      */
    C[A[j]] = C[A[j]]-1;
  }
}
```

$\Rightarrow$ Qual o papel da segunda instrução do último ciclo?

Análise de "Counting Sort"

Tempo

```
void counting_sort(int A[], int B[], int k) {
  int C[k+1];
  for (i=0 ; i<=k ; i++)                          Θ(k)
    C[i] = 0;
  for (j=1 ; j<=N ; j++)                          Θ(N)
    C[A[j]] = C[A[j]]+1;
  for (i=1 ; i<=k ; i++)                          Θ(k)
    C[i] = C[i]+C[i-1];
  for (j=N ; j>=1 ; j--) {                        Θ(N)
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
  }
}
```

Se $k = O(N)$ então $T(N) = \Theta(N+k) = \Theta(N)$ $\Rightarrow$ Alg. Tempo Linear!

Propriedade de *Estabilidade*

Elementos iguais aparecem no sequência ordenada pela mesma ordem em que estão na sequência inicial

Esta propriedade torna-se útil apenas quando há dados associados às chaves de ordenação.

"Counting Sort" é estável. $\Rightarrow$ Porquê?

Algoritmo "Radix Sort"

Algoritmo útil para a ordenação de sequências de estruturas com múltiplas chaves. Imagine-se um vector de *datas* com campos *dia*, *mês*, e *ano*. Para efectuar a sua ordenação podemos:

1. Escrever uma função de comparação (entre duas datas) que compara primeiro os anos; em caso de igualdade compara então os meses; e em caso de igualdade destes compara finalmente os dias:

   $21/3/2002 < 21/3/2003$ compara apenas anos
   $21/3/2002 < 21/4/2002$ compara anos e meses
   $21/3/2002 < 22/3/2002$

   Qualquer algoritmo de ordenação tradicional pode ser então usado.

2. Uma alternativa consiste em ordenar a sequência três vezes, uma para cada chave das estruturas. Para isto é necessário que o algoritmo utilizado para cada ordenação parcial seja *estável*.

Exemplo de Utilização de "Radix Sort"

O mesmo princípio pode ser utilizado para ordenar sequências de inteiros com o mesmo número de dígitos, considerando-se sucessivamente cada dígito, partindo do *menos significativo*.

Exemplo:

$$\begin{array}{c} 475 \\ 985 \\ 123 \\ 598 \\ 246 \\ 812 \\ 444 \end{array} \Longrightarrow \begin{array}{c} 81\color{red}{2} \\ 12\color{red}{3} \\ 44\color{red}{4} \\ 47\color{red}{5} \\ 98\color{red}{5} \\ 24\color{red}{6} \\ 59\color{red}{8} \end{array} \Longrightarrow \begin{array}{c} 8\color{red}{12} \\ 1\color{red}{23} \\ 4\color{red}{44} \\ 2\color{red}{46} \\ 4\color{red}{75} \\ 9\color{red}{85} \\ 5\color{red}{98} \end{array} \Longrightarrow \begin{array}{c} \color{red}{123} \\ \color{red}{246} \\ \color{red}{444} \\ \color{red}{475} \\ \color{red}{598} \\ \color{red}{812} \\ \color{red}{985} \end{array}$$

Como proceder se os números não tiverem todos o mesmo número de dígitos?

Algoritmo "Radix Sort"

```
void radix_sort(int A[], int p, int r, int d)
{
  for (i=1; i<=d; i++)
    stable_sort_by_index(i, A, p, r);
}
```

- o dígito menos significativo é 1; o mais significativo é d.

- o algoritmo `stable_sort_by_index(i, A, p, r)`:
  - ordena o vector A entre as posições p e r, tomando em cada posição por chave o dígito i;
  - tem de ser estável (por exemplo, "counting sort").

Prova de Correcção: indutiva. $\Rightarrow$ Qual a propriedade fundamental?

Análise de Tempo de Execução de "Radix Sort"

- Se `stable_sort_by_index` for implementado pelo algoritmo "counting sort", temos que dados N números com d dígitos, e sendo k o valor máximo para os números, o algoritmo "radix sort" ordena-os em tempo $\Theta(d(N+k))$.

- Se $k = O(N)$ e d for pequeno, tem-se $T(N) = \Theta(N)$.

- "Radix Sort" executa então (em certas condições) em tempo linear.

Análise Amortizada de Algoritmos

Uma nova ferramenta de análise, que permite estudar o tempo necessário para se efectuar uma *sequência de operações sobre uma estrutura de dados*.

- A ideia chave é considerar a média em relação à sequência de operações: uma operação singular pode ser pesada; no entanto, quando considerada globalmente como parte de uma sequência, o seu tempo médio de execução pode ser baixo.

- Trata-se do estudo do custo médio de cada operação no pior caso, e não de uma análise de caso médio!

3 técnicas: Análise agregada, Método contabilístico, Método do potencial

Análise agregada

Princípios:

- Mostrar que para qualquer n, uma sequência de n operações executa no pior caso em tempo $T(n)$.
- Então o *custo amortizado* por operação é $T(n)/n$.
- Considera-se que todas as diferentes operações têm o mesmo custo amortizado (as outras técnicas de análise amortizada diferem neste aspecto).

Análise agregada – Exemplo de Aplicação

Consideremos uma estrutura de dados do tipo **Pilha** ("Stack") com as operações `Push(S,x)`, `Pop(S)`, e `IsEmpty(S)` habituais. Cada uma executa em tempo constante (arbitraremos tempo 1).

Considere-se agora uma extensão deste tipo de dados com uma operação `MultiPop(S,k)`, que remove os `k` primeiros elementos da stack, deixando-a vazia caso contenha menos de `k` elementos.

```
MultiPop(S,k) {
  while (!IsEmpty(S) && k != 0) {
    Pop(S);
    k = k-1;
  }
```

Qual o tempo de execução de `MultiPop(S,k)`?

Análise de `MultiPop(S,k)`

- número de iterações do ciclo `while` é $\min(s, k)$ com s o tamanho da stack.
- em cada iteração é executado um `Pop` em tempo 1, logo o custo de `MultiPop(S,k)` é $\min(s, k)$.

Consideremos agora uma sequência de n operações `Push`, `Pop`, e `MultiPop` sobre a stack.

- Como o tamanho da stack é *no máximo* n, uma operação `MultiPop` na sequência executa no *pior caso* em tempo $O(n)$.
- Logo *qualquer operação* na sequência executa no pior caso em $O(n)$, e a sequência é executada em tempo $O(n^2)$.

Esta estimativa considera isoladamente o tempo no pior caso de cada operação. Apesar de correcta dá um limite superior demasiado "largo".

Análise agregada de `MultiPop(S,k)`

A análise agregada permite obter um limite mais apertado para o tempo de execução de uma sequência de n operações `Push`, `Pop`, e `MultiPop`

- Cada elemento é `popped` no máximo uma vez por cada vez que é `pushed`.
- Na sequência, `Pop` pode ser invocado (incluindo a partir de `MultiPop`) no máximo tantas vezes quantas as invocações de `Push` – no máximo n.
- O custo de execução da sequência é então $\leq 2n$ (1 por cada `Push` e `Pop`), ou seja, $O(n)$.
- O custo médio, ou custo amortizado, de cada operação, é então $O(n)/n = O(1)$.

Apesar de uma operação `MultiPop` isolada poder ser custosa ($O(n)$), o seu custo amortizado é $O(1)$.

Análise do Tempo de Execução de `BFS` e `DFS`

Utilizamos **Análise Agregada**. No caso da pesquisa em largura:

- Cada vértice é `enqueued` e `dequeued` exactamente uma vez. Isto é garantido pelo facto de os vértices nunca serem pintados de branco depois da inicialização.
- `enqueue` e `dequeue` executam em tempo $O(1)$, logo o tempo total gasto em operações sobre a Queue é $O(|V|)$.
- A lista de adjacência de cada vértice é percorrida no máximo uma vez (quando o vértice é `dequeued`), e o comprimento *total* das listas é $\Theta(|E|)$. Logo, o tempo total tomado pela travessia das listas de adjacências é $O(|E|)$.
- A *inicialização* do algoritmo é feita em tempo $O(|V|)$.
- Assim, o tempo de execução de `BFS` é $O(|V| + |E|)$
  – linear no tamanho da representação por listas de adjacências de G.

Análise do Tempo de Execução de `BFS` e `DFS`

E para a pesquisa em profundidade:

- Em `CDFS`, os ciclos `for` executam em tempo $O(|V|)$, excluindo o tempo tomado pelas invocações de `DFS`.

- `DFS` é invocada *exactamente uma vez* para cada vértice do grafo (a partir de `CDFS` ou da própria `DFS`) – garantido porque é invocada apenas com vértices brancos e a primeira coisa que faz é pintá-los de cinzento (e nenhum vértice volta a ser pintado de branco).

- Em `DFS(`G`,s)`, o ciclo `for` é executado $Adj(s)$ vezes. No total das invocações de `DFS`, este ciclo é então executado $\sum_{v \in V} |Adj(v)| = \Theta(|E|)$.

- O tempo de execução de `CDFS` é então $\Theta(|V| + |E|)$ – também linear no tamanho da representação.

Tempo de Execução do Algoritmo de Prim

Análise agregada (pior caso) sobre $G = (V, E)$:

- Número de operações de inicialização é linear em $|V|$.
- Os dois ciclos `for` podem ser fundidos num único que atravessa vértices adjacentes a x. Este ciclo atravessa completamente todas as listas de adjacências, e como o seu corpo é executado em tempo constante, demora $\Theta(|E|)$.
- Teste `if` $(\text{não há arcos candidatos})$ é executado no máximo $|V| - 1$ vezes.
- Número total de comparações está em $O(|V|^2)$. $(\Rightarrow$ porquê?)

Então, o algoritmo executa em tempo $O(|V|^2 + |E|)$.