II. Análise de Tempo de Execução de Algoritmos

Tópicos:

- Modelo computacional
- Análise de melhor caso e pior caso
- Notação assimptótica para o comportamento de funções
- Análise de algoritmos recursivos – equações de recorrência
- (*) Análise de caso médio; algoritmos com aleatoriedade
- (*) Análise Amortizada

II. Análise de Tempo de Execução de Algoritmos

Tópicos (continuação):

- Estratégias algorítmicas: incremental, divisão e conquista.

- 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"

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.

O modelo, tal como 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*.

O modelo computacional não entra em conta com aspectos sofisticados das arquitecturas modernas, como a hierarquia de memória (*cache, memória virtual*) e que podem ser relevantes na análise.

A utilização do modelo RAM é no entanto quase sempre bem sucedida.

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 concreta

$\Rightarrow$ 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)
9	`}`		

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: $0 \leq m \leq b - a + 1 = N$.

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, $m = 0$

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

Pior Caso: valor não encontrado, $m = N$

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

logo $T(N)$ é função *linear* de N

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);`		
7	`}`		

Note-se que

- c_1 é o custo de executar `i=a` e avaliar `i<b` **ou** de executar `i++` e avaliar `i<b`, não distinguindo entre esses dois casos
- c_3 é o custo das linhas 5 e 6, não distinguindo entre os dois casos do condicional

Estas simplificações não alteram muito a análise (porquê?)

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]) printf(...)                c3            S2
6          printf("%d igual a %d\n", i, j);
7  }
```

Seja $N = b-a+1$ (tamanho do input). O ciclo exterior executa $(b-1)-a+1 = b-a$ iterações, sendo a condição `i<b` avaliada $b-a+1 = N$ vezes.

Para um dado valor de $i \in \{a, \ldots, b-1\}$, o ciclo interior executa $b-(i+1)+1 = b-i$ iterações, sendo a condição `j<=b` avaliada $b-i+1$ vezes. Logo

$$S_1 = \sum_{i=a}^{b-1} b-i+1 \qquad \text{e} \qquad S_2 = \sum_{i=a}^{b-1} b-i$$

Tempo Total de Execução

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

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:

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

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*: assume-se (fixado o tamanho) que todos os inputs ocorrem com igual probabilidade (e.g. na ordenação de um vector todas as permutações são igualmente prováveis). Utilizam-se técnicas probabilísticas.

Esta análise é muitas vezes substituída por um estudo empírico: geram-se inputs de forma aleatória e regista-se o comportamento do algoritmo.

Notação Assimptótica – Comportamento de Funções

Tempo de execução de um algoritmo expresso como função do tamanho do input:

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

Normalmente o tempo *exacto* não é importante: para inputs de tamanho elevado o efeito das constantes multiplicativas e dos termos de menor grau é anulado.

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

$$T(N) = \Theta(N^2)$$

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.

Notação Θ

Consideraremos apenas funções *assimptoticamente não-negativas*.

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.

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* $f(n)$, $g(n)$,

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

Utilização da Notação Assimptótica

Relembremos o primeiro algoritmo analisado, caracterizado

- no melhor caso por $T(N) = c_1 + c_2 + c_4 + c_5$,
- no pior caso por $T(N) = (c_2 + c_3)N + (c_1 + c_2 + c_4 + c_5)$

Podemos escrever: $T(N) = \Omega(1)$ e $T(N) = O(N)$.

Já para o segundo algoritmo, tínhamos

- $T(N) = k_2N^2 + k_1N + k_0$

sem distinção de diferentes casos. Podemos então escrever: $T(N) = \Theta(N^2)$.

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, mas não a seguinte!]

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

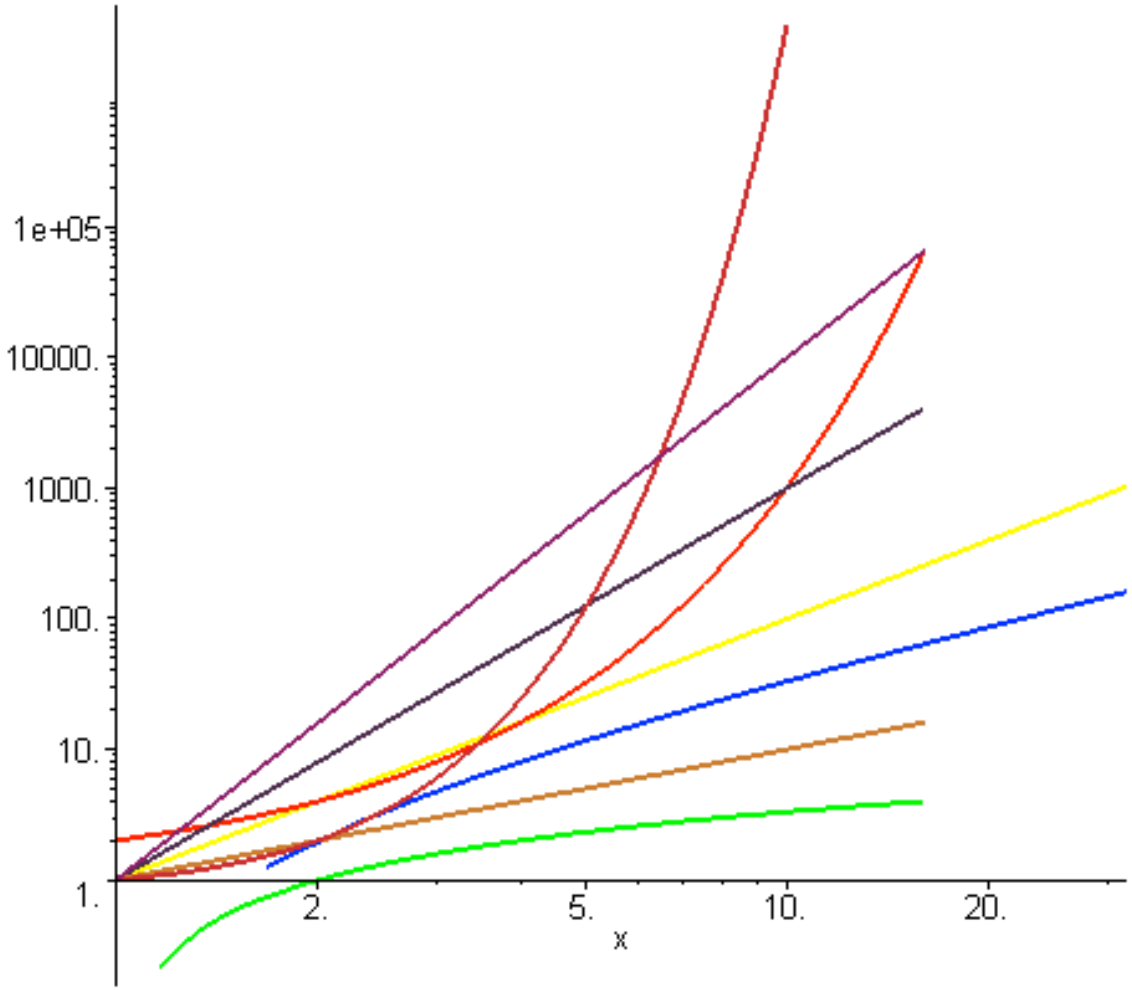
Algumas relações úteis:

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

Crescimento de Funções Típicas

Tempo (μs)		$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	
Tempo assimp.		n	$n \lg n$	n^2	n^3	2^n
Tempo de	n=10	$.00033s$	$.0015s$	$.0013s$	$.0034s$	$.001s$
execução por	n=100	$.003s$	$.03s$	$.13s$	$3.4s$	$4.10^{14}s$
tamanho do	n=1000	$.033s$	$.45s$	$13s$	$.94h$	séculos
input	n=10000	$.33s$	$6.1s$	$22m$	39 dias	$\cdots$
	n=100000	$3.3s$	$1.3m$	1.5 dias	108 anos	$\cdots$
Tamanho máx.	$1s$	3.10^4	2000	280	67	20
do input para	$1m$	18.10^5	82000	2200	260	26

Crescimento de Funções Típicas



$\log x$, funções polinomiais de diversos graus (rectas, incluindo $n \lg n$), 2^x, e $x!$.

Análise assimptótica sumária: pesquisa linear num vector

Identificamos uma ou mais operações elementares (tempo constante) que sejam representativas do tempo de execução, e cuja variação corresponda ao melhor e pior caso do algoritmo. Basta considerar a execução destas operações.

No caso da pesquisa linear temos a operação de comparação, `(v[i]!=k)`

```
                                                          M.C.          P.C.
1  int procura (int *v, int a, int b, int k) {
2    int i;
3    i=a;
4    while ((i<=b) && (v[i]!=k))                           1             N
5      i++;
6    if (i>b) return -1;
8    else return i;
9 }
```

$$T(N) = \Omega(1), O(N)$$

Análise assimptótica sumária: pesquisa de duplicados

Contamos as operações de comparação, `(v[i]==v[j])`

		M.C.	P.C.
1	`void dup (int *v, int a, int b) {`		
2	`  int i, j;`		
3	`  for (i=a ; i<b ; i++)`		
4	`  for (j=i+1 ; j<=b ; j++)`		
5	`  if (v[i]==v[j])`	S	S
6	`  printf("%d igual a %d\n", i, j);`		
7	`}`		

$$S = \sum_{i=a}^{b-1} b - i, \quad \text{logo} \quad T(N) = \Theta(N^2)$$

O Tamanho de um Problema – Questão Subtil!

Problema: Dado um inteiro positivo n, haverá dois inteiros $j, k > 1$ tais que $n = jk$? (i.e, será n não-primo?)

Considere-se o seguinte algoritmo do tipo "força bruta":

```
found = 0;
j = 2;
while ((!found) && j < n) {
  if (n mod j == 0) found = 1;
  else j++;
}
```

Este algoritmo executa em tempo $O(n)$, no entanto trata-se de um problema famoso pela sua dificuldade (e é por isso utilizado em muitos algoritmos criptográficos).

De facto é importante identificar correctamente o tamanho de n, uma vez que disso depende a classificação do algoritmo como polinomial ou exponencial.

Tamanho e Representação

A noção apropriada de tamanho de um número corresponde ao número de caracteres necessários para o escrever: o tamanho de 3500 é 4 em notação decimal.

Um inteiro n em notação decimal ocupa aproximadamente $\log_{10} n$ dígitos; em notação binária (representação em máquina) ocupa $\log_2 n$ dígitos.

Observe-se: se um algoritmo executa no pior caso em tempo linear em n, e n tem tamanho $s = \log_k n$, então o algoritmo executa em tempo $n = k^s$, ou seja $T(s) = O(k^s)$. Um algoritmo de tempo aparentemente linear é de facto de tempo exponencial no tamanho da representação do número!

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*.

[Ordenação *in-place*, apenas utiliza espaço adicional de tamanho constante]

Tempo de execução depende dos dados de entrada:

- tamanho
- "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 0 e $N-1$ do vector A.

```
void insertionSort(int A[], int N) {
    int i, j, key;
    for (j=1 ; 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 0 e $j-1$ os mesmos valores que lá estavam inicialmente, já ordenados.

$\Rightarrow$ **Utilidade**: com j=N, invariante garante que o vector está ordenado.

$\Rightarrow$ Verificação da *Preservação* obriga a estabelecer e demonstrar a validade de um 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 que estavam inicialmente na região* $A[i+1, ..., j-1]$. *O valor da variável* key *é inferior a todos esses valores.*

Análise do Tempo de Execução

```
                                                  Custo        n. Vezes
void insertion_sort(int A[], int N) {
  for (j=1 ; 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 $S_1 = \sum_{j=1}^{N-1} n_j$; $S_2 = \sum_{j=1}^{N-1}(n_j - 1)$ onde n_j é o número de vezes que a *condição* do ciclo `while` é avaliada, 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 um 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.

Temos então um tempo de execução $T(N) = \Omega(N)$.

Tempo Total de Execução

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

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

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

Logo $T(N) = k_2 N^2 + k_1 N + k_0$, função *quadrática* de N

$T(N) = O(N^2)$.

Para a análise assimptótica não é relevante a determinação exacta dos coeficientes k_2, k_1, k_0.

Análise assimptótica sumária: "insertion sort"

```
                                                    M.C.          P.C.
void insertion_sort(int A[], int N) {
  for (j=1 ; j<N ; j++) {
    key = A[j];
    i = j-1;
    while (i>=0 && A[i] > key) {                    N-1           S
      A[i+1] = A[i];
      i--;
    }
    A[i+1] = key;
  }
}
```

$$S = \sum_{j=1}^{N-1} j, \quad \text{logo} \quad T(N) = \Omega(N), O(N^2)$$

INTERLÚDIO: 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
- 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) – vimos 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** – e.g. algoritmo de grafos *All-Pairs-Shortest-Paths*.
- Algoritmos **Aleatorizados** ou Probabilísticos – veremos uma versão modificada do algoritmo *quicksort*.

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 (caso de paragem);
   - utilizando recursivamente a mesma estratégia em 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 tamanho similar

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 ordenada*:

```
void merge(int A[], int p, int q, int r) {
  int n1 = q-p+1, n2 = r-q;
  int L[n1+1], R[n2+1];

  for (i=0 ; i<n1 ; i++)  L[i] = A[p+i];
  for (j=0 ; j<n2 ; j++)  R[j] = A[q+j+1];
  L[n1] = INT_MAX; R[n2] = INT_MAX;

  i = 0; j = 0;
  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?)
- O tamanho do input é $n = r - p + 1$, e são executados n passos básicos.
- Este algoritmo executa então em *tempo linear*: $T(N) = \Theta(n)$.
- $\Rightarrow$ 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[0..n1]` e `R[0..n2]`;

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

O invariante permite provar a correcção do algoritmo, i.e, no fim da execução do ciclo o vector `A` contém a fusão ordenada 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 o tamanho de cada sub-sequência criada no passo de divisão?

Invocação inicial:

`merge_sort(A,0,n-1);`

em que `A` contém `n` elementos.

Análise do Tempo de Execução

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 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 correspondente ao 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:

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

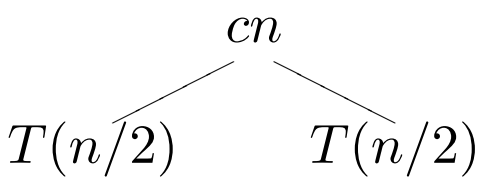
Em que cada **divisão** gera a sub-problemas, sendo o tamanho de cada sub-problema uma fracção $1/a$ do original

Construção da Árvore de Recursão – 1^o Passo

Reescrevamos a relação de recorrência:

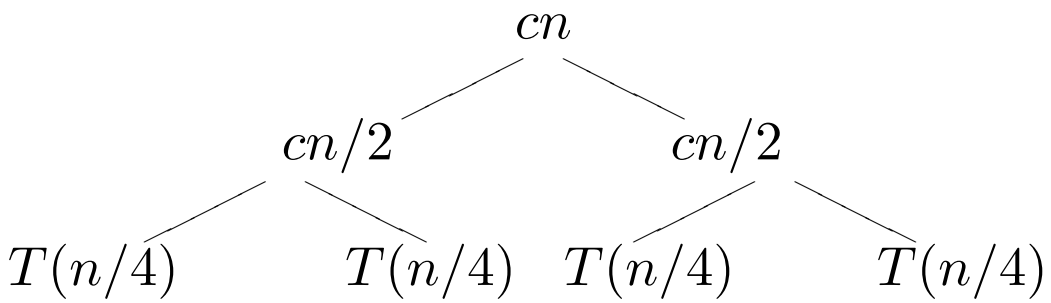
$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

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

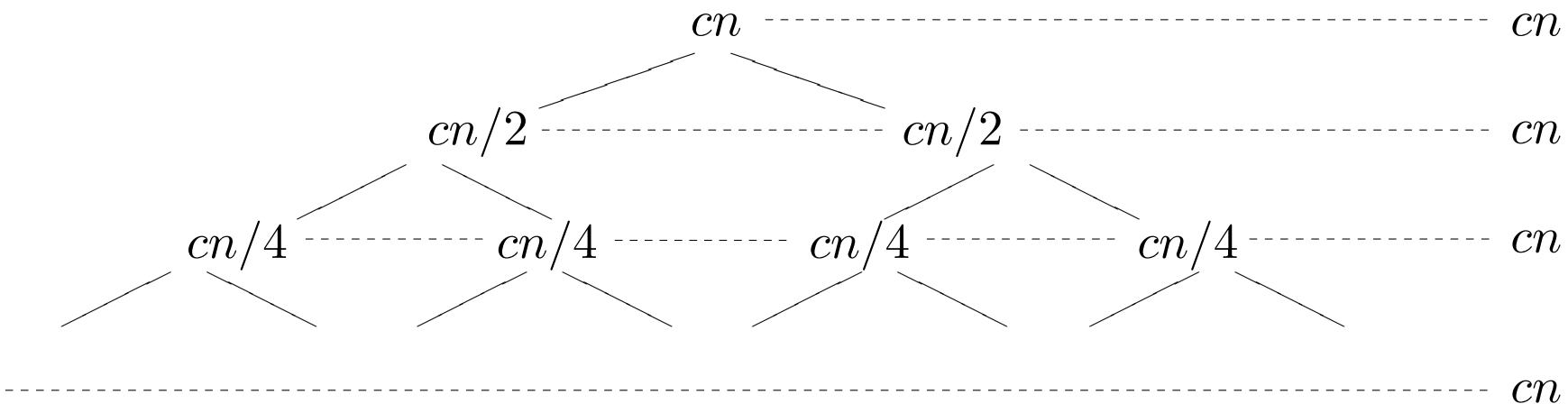


Construção da Árvore de Recursão – 2^o Passo

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



Árvore de Recursão



- último nível da árvore tem n folhas, cada uma com peso c (caso de paragem)
- árvore final tem $\lg n + 1$ níveis
  custo total de cada nível é sempre o mesmo, $= cn$
- então o custo total é $(\lg n + 1)cn = cn \lg n + cn$

"Merge sort" executa em $T(n) = \Theta(n \lg n)$, não havendo distinção de casos.

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 $c > 0$ e valores de n suficientemente grandes.

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. Admitamos $T(1) = 1$ como caso de base da recorrência. Então $T(1) \not\leq c1\lg 1 = 0$, mas a notação assimptótica permite-nos escolher outros, e de facto $T(2) \leq c2\lg 2$ e $T(3) \leq c3\lg 3$ para $c \geq 2$.

Exercícios

$\Rightarrow$ Provar que a recorrência $T(n) = 2T(\lfloor n/2 \rfloor + 100) + n$ é também $O(n \lg n)$

$\Rightarrow$ Provar também 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

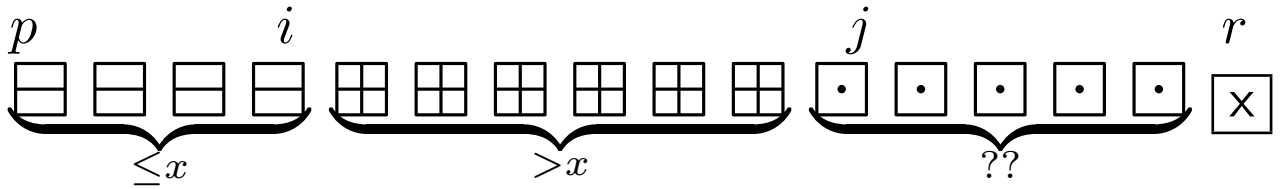
```
int partition (int A[], int p, int r)
{
  x = A[r];
  i = p-1;
  for (j=p ; j<r ; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j);
    }
  swap(A, i+1, r);
  return i+1;
}

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 $A[k] = x$.



⇒ Verificar as propriedades de *inicialização* ($j = p$, $i = p - 1$), *preservação*, e *utilidade* ($j = r$)

⇒ 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);
  }
}
```

Ao contrário de "merge sort", todo o esforço está agora no passo de *divisão*!

Execução de "Quicksort"

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	3	8	2	1	5	9	4	10	17	11	16	14	12	15	13
3	2	1	4	6	7	5	9	8	10	11	12	13	14	17	15	16
1	2	3	4	6	7	5	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	7	6	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

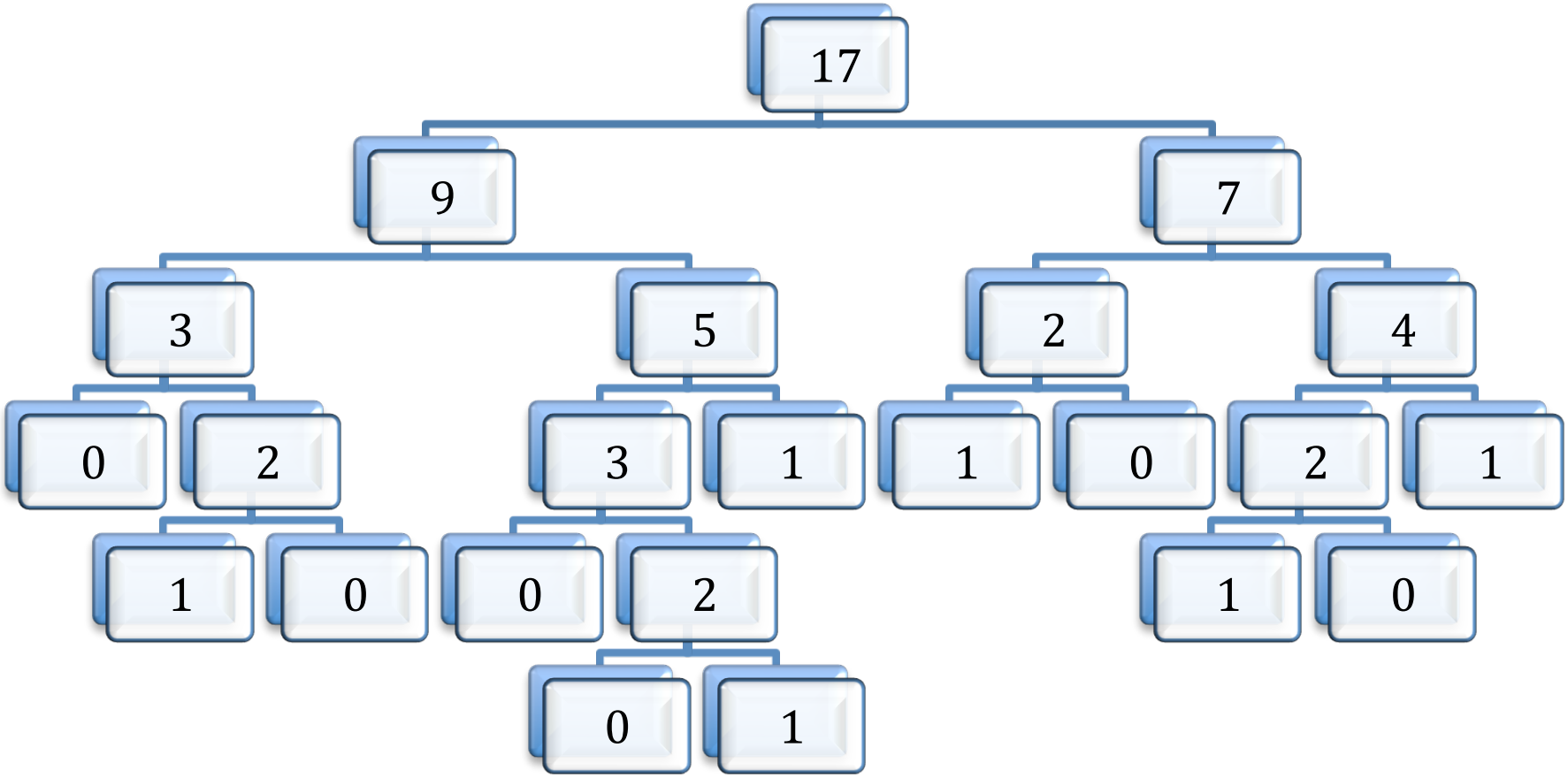
Pivot

Elemento já colocado na posição final

Array de comprimento 1 (paragem)

Árvore de Execução de "Quicksort"

(nós etiquetados com tamanho do *array* em cada chamada)



Análise de "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);
  }
}
```

Tentativa de escrever uma recorrência correspondente a este algoritmo:

$$T(n) \;=\; D(n) + T(k) + T(k') + C(n) \;=\; \Theta(n) + T(k) + T(n-k-1)$$

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

Mas note-se que isto *não é de facto uma recorrência*, uma vez que k não é constante! O seu valor varia com as chamadas recursivas.

Análise de Pior e Melhor Casos

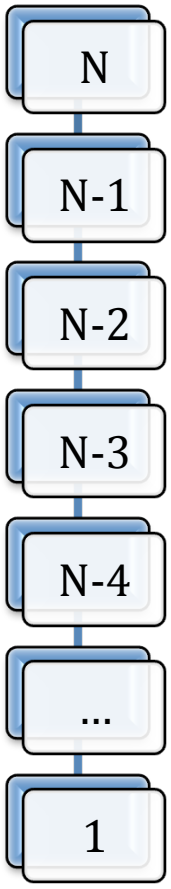
Podemos no entanto escrever recorrências para os casos extremos, usando operadores de mínimo / máximo:

Pior caso: $T(n) = \Theta(n) + \max_{k=0}^{n-1} (T(k) + T(n-k-1))$

Melhor caso: $T(n) = \Theta(n) + \min_{k=0}^{n-1} (T(k) + T(n-k-1))$

- utilizaremos a intuição para reescrever estas recorrências de forma simplificada
- as recorrências simplificadas têm soluções fáceis
- mostraremos que o uso das recorrências simplificadas é de facto correcto

Árvore de Execução de "Quicksort" no Pior Caso



⇒ Qual é o tempo de execução?

Análise de Pior Caso

A árvore anterior corresponde à situação em que *em todas as invocações recursivas* a partição produz vectores de dimensões 0 e $n-1$, ou seja, $k = 0$ ou $k = n-1$ em todas as chamadas recursivas, o que corresponde à seguinte recorrência simplificada:

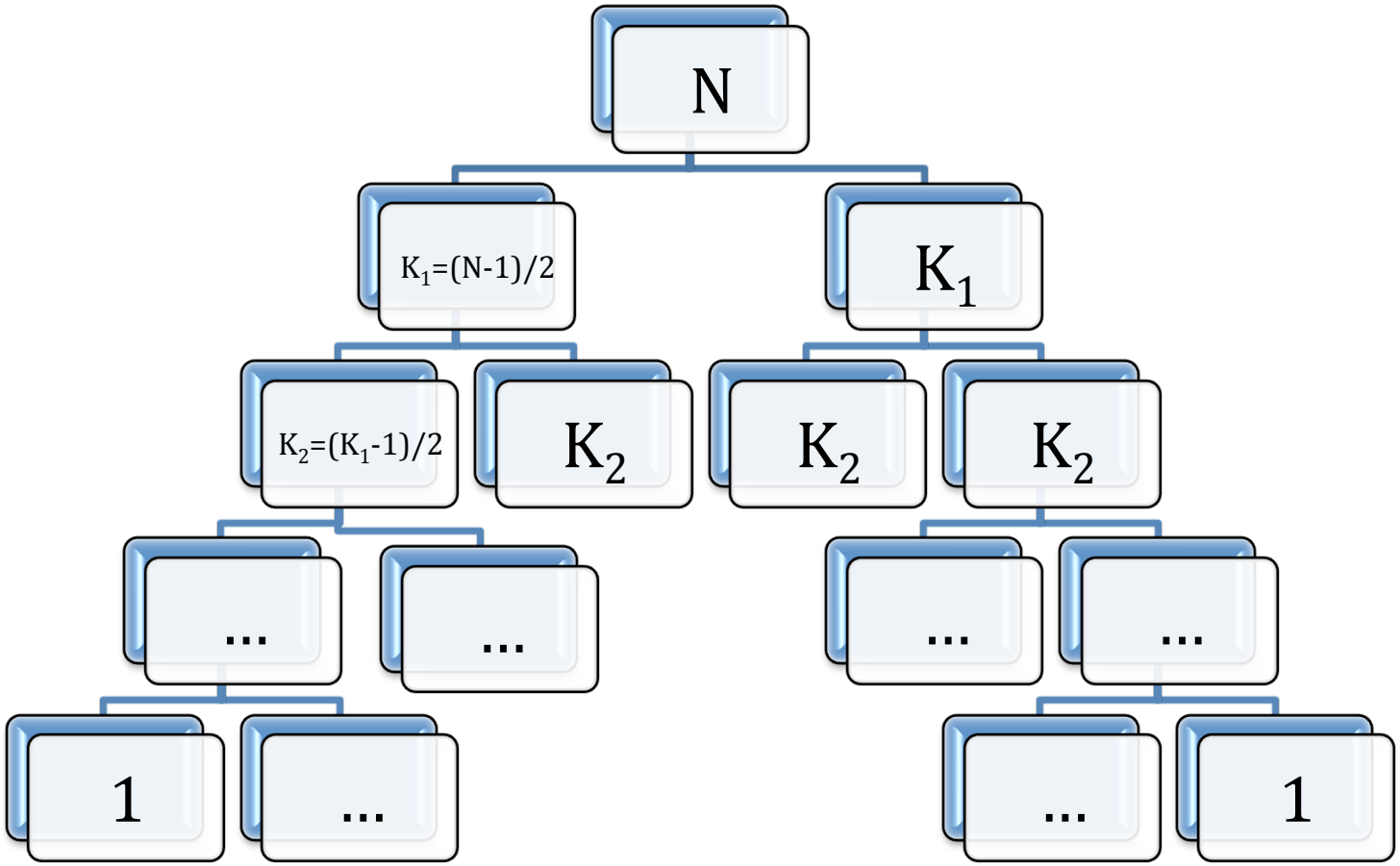
$$T(n) = \Theta(n) + T(n-1) + T(0) = \sum_{i=0}^{n} \Theta(i) = \Theta(n^2)$$

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

$\Rightarrow$ Que tipo de sequências provocam este comportamento?

$\Rightarrow$ E como mostrar que esta árvore e respectiva recorrência simplificada correspondem de facto ao pior caso? Até agora usámos apenas a intuição . . .

Árvore de Execução de "Quicksort" no Melhor Caso



Análise de Melhor Caso

- Se $n+1$ for uma potência de 2 (i.e. $n = 2^m - 1$ para algum m), temos uma árvore completa com $\lg n$ níveis, com custo $n, n-1, n-3, n-7, \ldots$.

- Para um valor de N arbitrário, partição produz vectores de dimensão $\lfloor (n-1)/2 \rfloor$ e $\lceil (n-1)/2 \rceil$:

$$T(n) = \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil - 1)$$

com solução (cfr. *merge sort*) $T(n) = \Theta(n \lg n)$.

Análise de Pior Caso - Justificação

O pior caso é descrito de forma exacta pela seguinte recorrência:

$$T(n) = \Theta(n) + \max_{k=0}^{n-1} (T(k) + T(n-k-1))$$

Provemos que $T(n) = O(n^2)$ pelo *método de substituição*.

Para isso admitimos que $T(n) \leq cn^2$; temos:

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

Análise de Pior Caso - Justificação

$$T(n) \leq \Theta(n) + c \max(\underbrace{2k^2 + (2-2n)k + (n-1)^2}_{P(k)})$$

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

Isto justifica a recorrência simplificada:

$$T(n) = \Theta(n) + T(n-1) + T(0)$$

Continuando o raciocínio, com $k = 0$ ou $k = n-1$ tem-se:

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

Análise de Tempo de Execução de "quicksort"

- No pior caso "quicksort" executa em tempo quadrático, tal como "insertion sort", mas este pior caso ocorre (por exemplo) quando a sequência de entrada se encontra já ordenada, caso em que "insertion sort" executa em tempo $\Theta(n)$.

- 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 para constatar isto.

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*.

Árvore de Decisão

- Cada nó corresponde a um teste de comparação entre dois elementos da sequência;

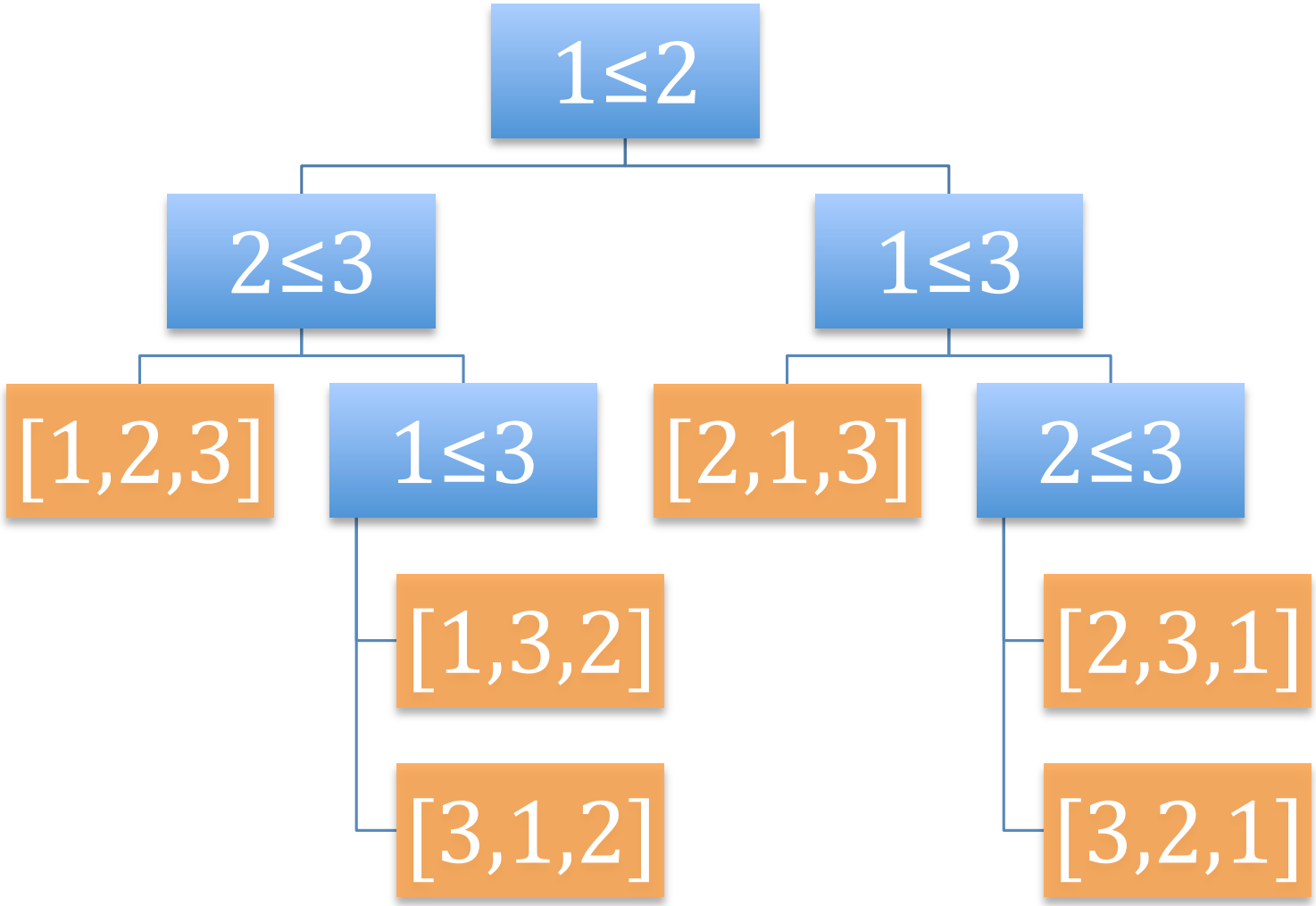
- tem como sub-árvore esquerda (resp. direita) a árvore correspondente à continuação da execução do algoritmo caso o teste tenha resposta 'verdadeiro' (resp. 'falso').

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

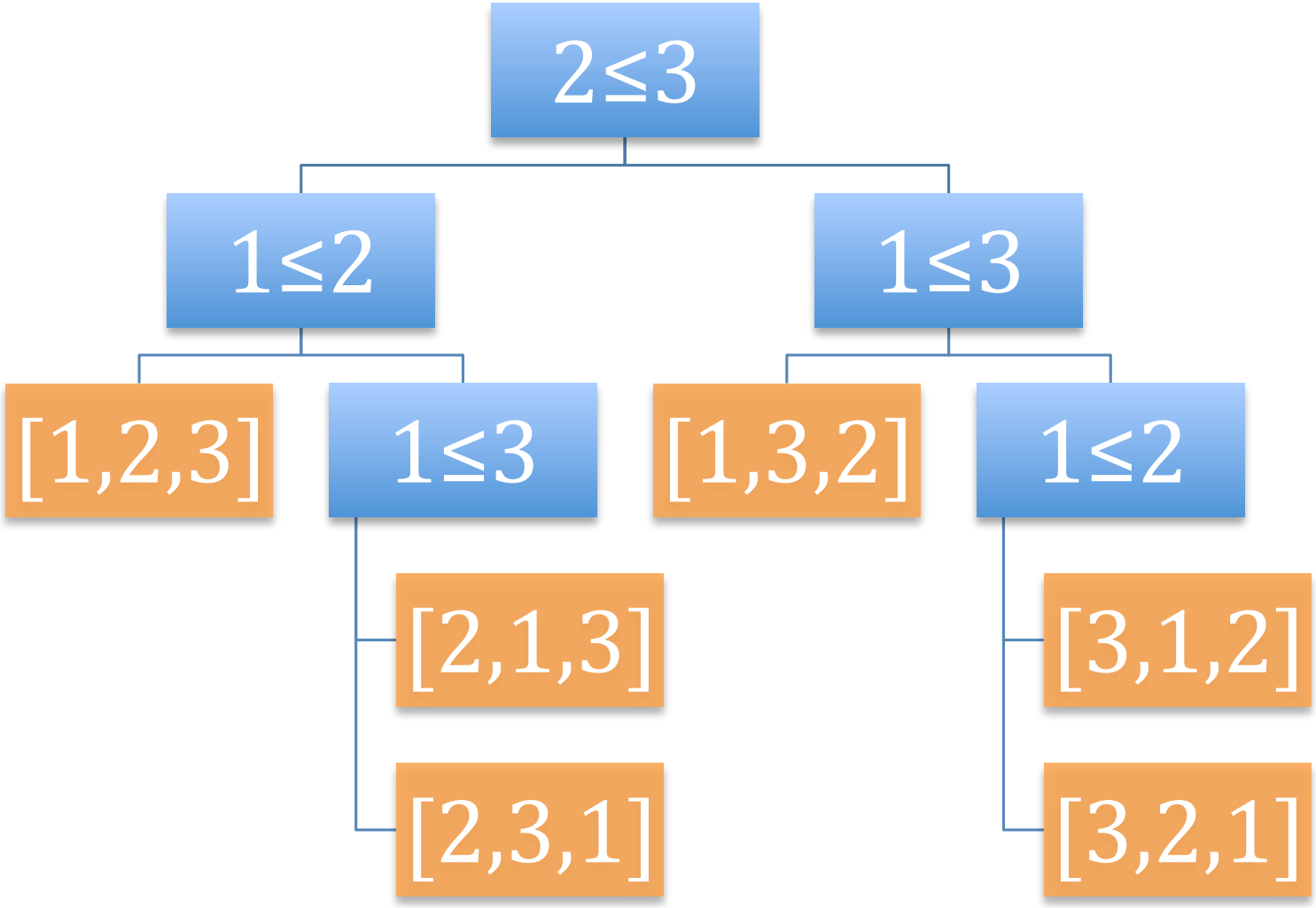
A execução concreta do algoritmo para um determinado vector de input corresponde a um *caminho da raíz para uma folha*, ou seja, uma sequência de comparações.

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. 20)*

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

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"

Será possível conceber algoritmos de comparação sem recorrer a comparações?

- **Restrição:** 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?

- **Funcionamento não é *in-place***: "counting sort" produz novo vector (*output*) e utiliza vector auxiliar temporário.

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

Algoritmo "Counting Sort"

Ordena vector entre as posições 1 e N.

```
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;
  }
}
```

⇒ 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 na 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:

$$\begin{array}{c} 475 \\ 985 \\ 123 \\ 598 \\ 246 \\ 812 \\ 444 \end{array} \Longrightarrow \begin{array}{c} 812 \\ 123 \\ 444 \\ 475 \\ 985 \\ 246 \\ 598 \end{array} \Longrightarrow \begin{array}{c} 812 \\ 123 \\ 444 \\ 246 \\ 475 \\ 985 \\ 598 \end{array} \Longrightarrow \begin{array}{c} 123 \\ 246 \\ 444 \\ 475 \\ 598 \\ 812 \\ 985 \end{array}$$

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 de Caso Médio

- *Análise Probabilística*: implica a consideração de uma *distribuição* sobre os inputs, que torne possível o cálculo do *valor esperado* do número de operações elementares executadas.

- Por exemplo no caso da ordenação, esta distribuição descreve a probabilidade de ocorrência de cada permutação possível dos elementos do vector de entrada.

- Com base nesta distribuição, calcula-se o valor esperado do número de operações de comparação efectuadas.

- Na ausência dessa distribuição, é necessário *assumir-se* algo sobre os inputs.

- Por exemplo no caso da ordenação, assume-se que todas as permutações podem ocorrer com igual probabilidade $(1/N!)$.

Exemplo 1: "Quicksort"

- Questão prévia: num algoritmo recursivo pode ser menos evidente a escolha de operações elementares representativas e sua contagem.

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

- Em geral o tempo total de execução é então $T(n) = O(n + X)$, onde X é o número *total* de *comparações* efectuadas. Salvaguarda-se assim a possibilidade de a contagem de operações representativas poder ser inferior ao número de chamadas recursivas, porque estas só por si resultam em tempo linear.

- Para a determinação do caso médio estudaremos o *valor esperado* de X.

- Assumimos que *todos os elementos são diferentes* (outros casos só podem executar mais rapidamente), e dispostos aleatoriamente no vector de entrada.

Análise de Caso Médio de "quicksort"

- O número esperado de comparações pode ser calculado como

$$\sum_{1 \leq i < j \leq n} Pr\left(A[i] \text{ e } A[j] \text{ serem comparados}\right)$$

- Seja $A'[i]$ o i-ésimo menor elemento do vector, i.e. o elemento contido na i-ésima posição do vector já ordenado. A soma anterior pode ser escrita como

$$\sum_{1 \leq i < j \leq n} Pr\left(A'[i] \text{ e } A'[j] \text{ terem sido comparados}\right)$$

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} Pr\left(A'[i] \text{ e } A'[j] \text{ terem sido comparados}\right)$$

Análise de Caso Médio de "quicksort"

- Dados i, j, Seja n_{ij} o número de elementos $x \in A$ tais que $A'[i] < x < A'[j]$.

- É possível que ambos estes elementos façam parte, conjuntamente, do argumento de diversas invocações recursivas do algoritmo, sempre que for feita uma partição usando como pivot um elemento inferior a $A'[i]$ ou superior a $A'[j]$. Mas a certa altura terá que acontecer uma de duas coisas:
  - Eles serão *separados* por uma partição com um pivot contido entre os seus valores: neste caso os dois farão parte do argumento de duas invocações "paralelas" de `quicksort`, e **não serão comparados**. [n_{ij} casos]
  - Um deles será utilizado como pivot numa chamada de partição que inclua o outro elemento, e naturalmente **serão comparados**. [2 casos]

- Estas são as duas únicas possibilidades! Logo,

$$Pr\left(A'[i] \text{ e } A'[j] \text{ serem comparados}\right) = \frac{2}{2 + n_{ij}} = \frac{2}{j - i + 1}$$

Execução de "Quicksort"

Por exemplo, (2,7) não são comparados, são separados pelo pivot 4 para posteriores invocações diferentes. Mas (2,4) e (4,7) são comparados, pela mesma razão.

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	3	8	2	1	5	9	4	10	17	11	16	14	12	15	13
3	2	1	4	6	7	5	9	8	10	11	12	13	14	17	15	16
1	2	3	4	6	7	5	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	7	6	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Olhando para o vector ordenado, e desconhecendo-se o vector original, a probabilidade de 2 e 7 terem sido comparados é de $2/6$.

Análise de Caso Médio de "quicksort"

Então $T_{avg}(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} Pr\,(A'[i] \text{ e } A'[j] \text{ serem comparados})$

$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$

$= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k}$

$\leq \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k}$

$= 2n \sum_{k=2}^{n} \frac{1}{k} = 2n(\sum_{k=1}^{n} \frac{1}{k} - 1)$

$\leq 2n(1 + \ln n - 1) = 2n \ln n$

$T_{avg}(n) = O(n \lg n)$ = Melhor caso!

Algoritmos aleatorizados

- 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 aleatorizado, nenhum input particular corresponde ao pior ou ao melhor casos; apenas o processamento prévio (aleatório) pode gerar um input pior ou melhor.

Algoritmo "quicksort" aleatorizado

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 como o algoritmo original, mas sem necessidade de se assumir o que quer que seja sobre os inputs
– a uniformidade é imposta, em vez de assumida.

Exemplo 2: Incremento de um vector de bits

Operação de incremento de um número inteiro (não negativo) representado como vector de bits (com o menos significativo na posição 0)

```
void inc (int b[], int N) {
  int i = 0;
  while ((i < N) && (b[i] == 1)) {
    b[i] = 0;
    i++;
  }
  if (i<N) b[i] = 1;
}
```

Claramente temos $T(N) = \Omega(1), O(N)$, quando o vector representa respectivamente 0 e $2^N - 1$ (1 em todas as posições).

Caso Médio do Incremento de um vector de bits

```
void inc (int b[], int N) {
  int i = 0;
  while ((i < N) && (b[i] == 1)) {
    b[i] = 0;
    i++;
  }
  if (i<N) b[i] = 1;
}
```

Para o caso médio assumimos que todos os números ocorrem com igual probabilidade, e calculamos o número esperado de *bit flips* (atribuições efectuadas):

$$
\begin{array}{rcll}
T(n) & = & 1 \text{ flip } * \mathbf{1/2} & [\text{metade dos números termina com 0}] \\
 & + & 2 \text{ flips } * \mathbf{1/4} & [\text{dos restantes, metade termina com 01}] \\
 & + & 3 \text{ flips } * \mathbf{1/8} & [\text{dos restantes, metade termina com 011}] \\
 & + & \cdots & \\
 & = & \sum_{k=1}^{n} k/2^k = O(1) & = \text{Melhor Caso!} \qquad \left[\sum_{k=1}^{\infty} k/2^k = 2\right]
\end{array}
$$

Exemplo 3: Pesquisa Linear

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

$T(n) = \Omega(1), O(n)$ Calculamos o valor esperado do número de comparações efectuadas, respectivamente 0 e $2^n - 1$ estudando agora dois casos diferentes:

Caso 1: `k` não ocorre no array, $T(n) = n$

Caso 2: `k` ocorre no array, com igual probabilidade em todas as posições.

Caso Médio da Pesquisa Linear

Caso 2: `k` ocorre no array. A probabilidade de a (primeira) ocorrência estar numa determinada posição é a mesma $(1/n)$ para todas as posições, logo:

$$\begin{array}{rcll} T(n) & = & 1 \text{ compar. } * \mathbf{1/n} & [k = v[a]] \\ & + & 2 \text{ compar/s. } * \mathbf{1/n} & [k = v[a+1]] \\ & + & 3 \text{ compar/s. } * \mathbf{1/n} & [k = v[a+2]] \\ & + & \cdots & \\ & = & \frac{1}{n} \sum_{k=1}^{n} k \;=\; \frac{n(n+1)}{2n} \;=\; \frac{n+1}{2} & \end{array}$$

Caso Médio: Seja p a probabilidade de `k` ocorrer no array.

$$T(n) \;=\; p\frac{n+1}{2} + (1-p)n \;=\; O(n) = \text{Pior Caso!}$$

Assumindo-se aleatoriedade temos $p = \frac{n}{2^m}$ em que m é o número de bits usado na representação dos números. Se for muito baixo teremos $T(n) \approx n$.

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 o *pior caso da sequência* de N operações, em situações em que este é claramente mais baixo do que a soma dos tempos de pior caso das N operações singulares.

- Trata-se do estudo do custo médio (relativamente à sequência) de cada operação no pior caso, e não de uma análise de caso médio! Não envolve ferramentas probabilísticas nem assunções sobre os inputs.

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 Agregada 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 $\min(s, k)$: pior caso *linear*, $O(s)$.

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

- Como o tamanho da stack é *no máximo* n, uma operação `MultiPop` na sequência executa no pior caso em tempo $O(n)$.
- Esta é a operação mais custosa, logo *qualquer operação* na sequência executa 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 pessimista.

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.
- As operações primitivas (`Push` ou `Pop`) são executadas no máximo $\leq 2n$ vezes, logo a sequência executa em tempo $O(n)$.
- O custo médio, ou custo amortizado, de cada operação no pior caso, é então *calculado* como $2n/n$, ou em termos assimptóticos $O(n)/n = O(1)$.

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

Método Contabilístico

Princípios:

- *Atribuír* custos amortizados c_i, possivelmente diferentes, às diversas operações, que podem não corresponder aos reais – superiores para algumas operações (acumulando *crédito)* e inferiores para outras (gastando crédito acumulado).

- Se o custo amortizado total de uma sequência (*qualquer sequência!*) de n operações for um limite superior para o custo real, $\sum_i t_i \leq \sum_i c_i$, então o custo total amortizado fornece um limite para o tempo de pior caso da sequência, o que significa que os custos amortizados individuais de cada operação podem ser considerados válidos para efeitos de análise de pior caso.

- Cálculo do *saldo* entre custo amortizado acumulado e custo real acumulado:

$$\begin{aligned} bal_0 &= K \qquad (\geq 0) \\ bal_{i+1} &= bal_i + c_{i+1} - t_{i+1} \end{aligned}$$

  Basta provar que para qualquer $i > 0$ se tem $bal_i \geq 0 \Longrightarrow bal_{i+1} \geq 0$.

Análise Contabilística de `MultiPop(S,k)`

Escolha de custos amortizados:

Custo	real t_i	amortizado c_i
`Push`	1	$2 = \mathrm{O}(1)$
`Pop`	1	$0 = \mathrm{O}(1)$
`MultiPop`	$\min(s, k)$	$0 = \mathrm{O}(1)$

Note-se que o custo real de `MultiPop` não é constante! Mas o custo amortizado constante é adequado para análise.

Basta observar que cada elemento contido na stack em qualquer momento tem crédito '1' associado, excedentário em relação ao custo real de 1, que resulta do custo amortizado de 2 cobrado a cada operação `Push`.

Este crédito de cada elemento é o suficiente para cobrir o custo real do seu `Pop` subsequente, quer individual, quer como parte de um `MultiPop`. Sendo assim estas operações podem ter custo amortizado 0, estando garantida a condição do slide anterior.

Método do Potencial

Princípios:

- O método contabilístico gera crédito individualmente, por cada operação efectuada.

- O método do potencial considera este crédito de forma global, para toda a estrutura de dados, com base numa *função de potencial* sobre os estados sucessivos da estrutura de dados.

- Seja Φ_i o potencial da estrutura de dados no estado i, ou seja depois de i operações da sequência. O custo amortizado da operação i é então dado por

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

- A ideia é que o potencial da estrutura deve aumentar com operações de baixo custo, e diminuir com operações de alto custo.

Método do Potencial

- O cálculo do custo amortizado total é *telescópico*:

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} t_i - \Phi_{i-1} + \Phi_i \\
&= (t_1 - \Phi_0 + \Phi_1) + (t_2 - \Phi_1 + \Phi_2) + (t_3 - \Phi_2 + \Phi_3) \\
&+ \ldots + (t_n - \Phi_{n-1} + \Phi_n) \\
&= \Phi_n - \Phi_0 + \sum_{i=1}^{n} t_i
\end{aligned}$$

- As seguintes condições são suficientes para garantir que $\sum_i c_i \geq \sum_i t_i$:
  $\Phi_0 = 0$ e $\Phi_i \geq 0$ para $i > 0$

- Nestas condições o custo total amortizado fornece um limite superior para o tempo de pior caso da sequência, como desejado.

Análise de Potencial de `MultiPop(S,k)`

Escolha de função de potencial:

$\Phi_i =$ *número total de elementos armazenados na stack depois de* i *operações*.

- Claramente tem-se $\Phi_0 = 0$ e $\Phi_i \geq 0$ para $i > 0$, logo a função é adequada.
- Calculemos então os custos amortizados.

Custo	real t_i	amortizado c_i
`Push`	1	$1 - \Phi_{i-1} + \Phi_i = 2$
`Pop`	1	$1 - \Phi_{i-1} + \Phi_i = 0$
`MultiPop`	$\min(s,k)$	$\min(s,k) - \Phi_{i-1} + \Phi_i =$ $\min(s,k) - \min(s,k) = 0$

- Assim, novamente se constata que o custo amortizado constante é adequado para análise.

Exemplo: Vectores Dinâmicos

Os vectores (*arrays*) são estruturas de dados com capacidade fixa e por isso limitada. Podem no entanto ser alocadas quer estaticamente quer dinamicamente. Por exemplo em C:

```
int u[1000];                                  // estático
int *v = calloc(1000, sizeof(int));           // dinâmico
```

Uma solução comum para o problema do crescimento de um vector para além da sua capacidade é a utilização de vectores dinâmicos com *realocação*: quando se pretende inserir um elemento que já não cabe no vector, cria-se um novo vector com o *dobro* da capacidade do primeiro, copiando-se todos os elementos do primeiro para o segundo, antes de se inserir o novo elemento.

```
int* myrealloc(int* v, int n) {
  int* new = calloc(2*n, sizeof(int));
  for (i=0 ; i<n ; i++) new[i] = v[i];
  return new;
}
```

Exemplo: Vectores Dinâmicos

Considere-se agora uma *stack* assim implementada e a sua operação *push*:

```
typedef struct stack {
  int *vec;
  int n;        // n. de elementos inseridos
  int cap;      // capacidade máxima
} Stack;

Stack push (Stack s, int x) {
  if (s.n == s.cap) {
    s.vec = myrealloc(s.vec, s.cap);
    s.cap *= 2;
  }
  (s.vec)[s.n] = x;
  (s.n)++;
  return s;
}
```

Análise Agregada: Vectores Dinâmicos

A análise de pior caso clássica da função *push* leva-nos a $T(n) = O(n)$, com n o tamanho actual da stack.

No entanto, claramente numa sequência de operações *push* a realocação será um evento raro! Aplicando análise agregada a uma sequência de n operações *push* a partir de uma *stack* de capacidade c inicialmente vazia, teremos

- c inserções de tempo (1),
- seguidas de uma inserção de tempo $(c + 1)$,
- novamente seguidas de $c - 1$ inserções de tempo (1),
- seguidas de uma inserção de tempo $(2 * c + 1)$,
- seguidas de de $2 * c - 1$ inserções de tempo (1),
- seguidas de uma inserção de tempo $(4 * c + 1)$,
- seguidas de de $4 * c - 1$ inserções de tempo (1),
- . . .

Análise Agregada: Vectores Dinâmicos

Considere-se $c = 1$:

i	1	2	3	4	5	6	7	8	9	10	...
cap_i	1	2	4	4	8	8	8	8	16	16	...
t_i	1	**1**+1	**2**+1	1	**4**+1	1	1	1	**8**+1	1	...
$\log(i-1)$		0	1		2				3		...

Note-se que $1 + 2 + 4 + 8 + \ldots = \sum_{k=0}^{\log(n-1)} 2^k$

$$\sum_{i=1}^{n} t_i = n + \sum_{k=0}^{\log(n-1)} 2^k = n + (2^{\log(n-1)+1} - 1) = n + 2(n-1) - 1 = O(n)$$

O custo amortizado de cada *push* é então $T(n) = \frac{O(n)}{n} = O(1)$

Método Contabilístico: Vectores Dinâmicos

Existe agora uma única operação, embora com dois tipos diferentes de execução. Seja '2' o seu custo amortizado. Vejamos a evolução do saldo ao longo da sequência, com $bal_0 = 0$ e $bal_{i+1} = bal_i - t_i + c_i$

i	1	2	3	4	5	6	7	8	9	10	. . .
cap_i	1	2	4	4	8	8	8	8	16	16	. . .
t_i	1	2	3	1	5	1	1	1	9	1	. . .
c_i	2	2	2	2	2	2	2	2	2	2	. . .
bal_i	1	1	0	1	-2						. . .

O custo amortizado escolhido não garante a condição fundamental de saldo acumulado não-negativo ao longo de toda a execução.

Método Contabilístico: Vectores Dinâmicos

Consideremos alternativamente o valor '3' para o custo amortizado.

i	1	2	3	4	5	6	7	8	9	10	...
cap_i	1	2	4	4	8	8	8	8	16	16	...
t_i	1	2	3	1	5	1	1	1	9	1	...
c_i	3	3	3	3	3	3	3	3	3	3	...
bal_i	2	3	3	5	3	5	7	9	3	4	...

Está assegurada a condição de saldo não-negativo, o que parece justificar que o custo amortizado de $3 = O(1)$ é adequado: a operação `push` é, em termos amortizados, de tempo constante no pior caso.

$\Rightarrow$ Intuitivamente, a que corresponde este custo amortizado de 3 unidades?

Note-se no entanto que não foi efectuado nenhum tipo de raciocínio formal para provar que o saldo permanece efectivamente positivo . . .

Método do Potencial: Vectores Dinâmicos

Seja $\Phi_i = 2 * n_i - cap_i$, i.e. a diferença entre o dobro do número de elementos contidos na stack e a sua capacidade.

- Consideraremos agora $cap_0 = 0$, primeiro passo expande capacidade para 1.
- Tem-se então $\Phi_0 = 0$ e $\Phi_i \geq 0$ para $i > 0$, logo a função é adequada.

$i = n_i$	0	1	2	3	4	5	6	7	8	9	10	. . .
cap_i	0	1	2	4	4	8	8	8	8	16	16	. . .
Φ_i	0	1	2	2	4	2	4	6	8	2	4	. . .

- Ao contrário do que acontecia com o saldo no método contabilístico, existe agora a certeza de que o potencial não é negativo, uma vez que mais de metade da capacidade está seguramente preenchida, logo $\Phi_i > 0$, e vai-se aproximando do valor da capacidade. Quando é necessário copiar esse número de elementos em tempo linear, existe um potencial com esse valor.

Método do Potencial: Vectores Dinâmicos

- O custo amortizado de `push` é então:
  - Se $n_{i-1} < cap_{i-1}$

    $$t_i - \Phi_{i-1} + \Phi_i = 1 - (2 * n_{i-1} - cap_{i-1}) + (2 * (n_{i-1} + 1) - cap_{i-1}) = 3$$

  - Se $n_{i-1} = cap_{i-1}$

    $$t_i - \Phi_{i-1} + \Phi_i = (n_{i-1} + 1) - (2 * n_{i-1} - n_{i-1}) + (2 * (n_{i-1} + 1) - 2 * n_{i-1}) = 3$$

- Novamente o custo de $3 = O(1)$ garante que a operação `push` é, em termos amortizados, de tempo constante no pior caso.

O método do potencial pode ser usado depois do método contabilístico, no sentido de *justificar* os custos amortizados arbitrados naquele método.

Queue Implementada com Duas Stacks

Uma implementação possível de uma fila de espera (*Queue*) utiliza duas pilhas A e B, por exemplo:

```
typedef struct queue {
 Stack a;
 Stack b;
} Queue;
```

- A inserção (`enqueue`) de elementos é sempre realizada na pilha A;

- para a saída de elementos (`dequeue`), se a pilha B não estiver vazia, é efectuado um *pop* nessa pilha; caso contrário, para todos os elementos de A excepto o último, faz-se sucessivamente *pop* e *push* na pilha B. Faz-se depois *pop* do último, que é devolvido como resultado.

Queue Implementada com Duas Stacks

- Análise de pior caso tradicional: `dequeue` executa em tempo $O(n)$.

- Análise agregada: uma operação `dequeue` de tempo linear no tamanho n da pilha só pode ser executada na sequência de n operações `dequeue` de tempo constante, que esvazie a pilha B. Globalmente esta sequência executa em tempo $O(n)$, logo cada `dequeue` executa em tempo amortizado $O(n)/n = O(1)$. A presença de operações `enqueue` pelo meio não altera esta análise.

- Método contabilístico: basta atribuír custo amortizado de 3 à operação `enqueue`, cobrindo o custo das operações `Push`, `Pop`, `Push` necessárias para o elemento passar para a Stack B. `dequeue` tem custo amortizado 1.

- Método do potencial: $\Phi = 2 *$ tamanho da Stack A. `dequeue` não altera o potencial (custo amortizado = custo real = 1); `enqueue` aumenta em duas unidades o potencial (custo amortizado = 2 + custo real).