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.

⇒ 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.

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

Análise do Tempo de Execução

Código	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);
  }
}
```

⇒ 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:

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

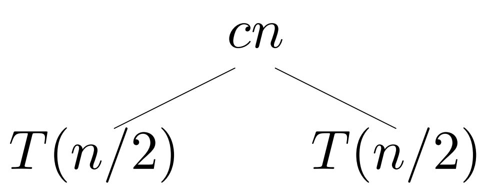
Em que cada **divisão** gera a sub-problemas, sendo o tamanho de cada sub-problema uma fracção $1/b$ do original (pode ser $b \neq a$).

Construção da Árvore de Recursão – 1^o Passo

Reescrevamos a relação de recorrência:

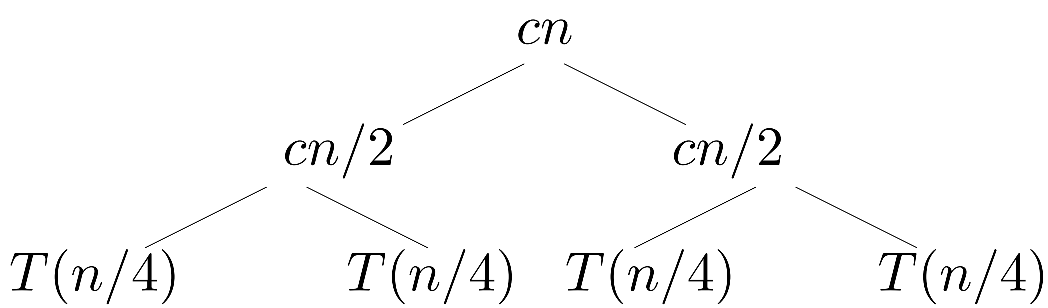
$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

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.

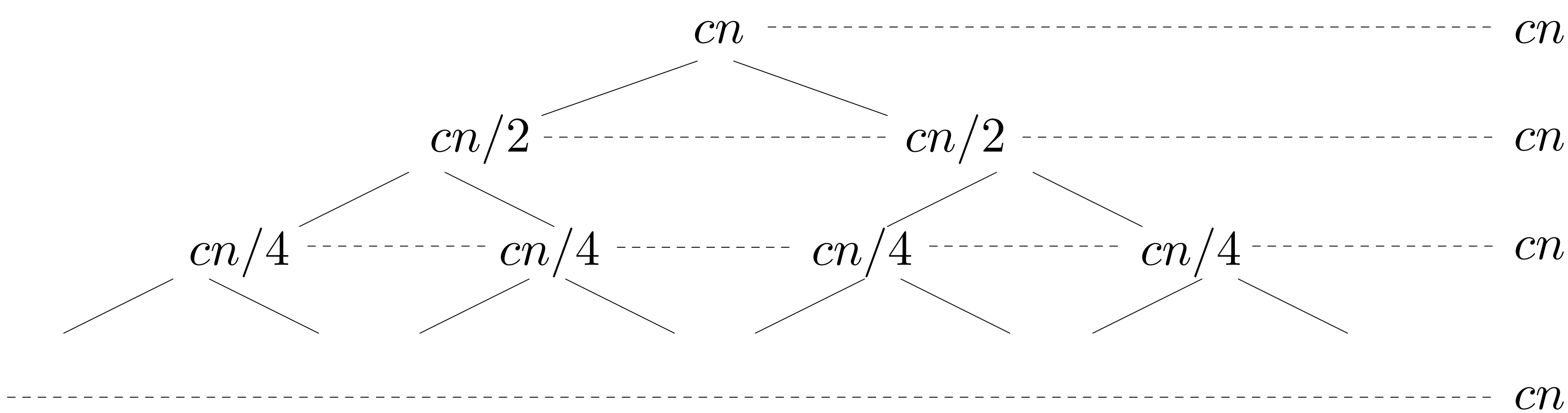


Construção da Árvore de Recursão – 2^o Passo

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



Árvore de Recursão



- árvore final tem $\lg n + 1$ níveis (⇒ 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

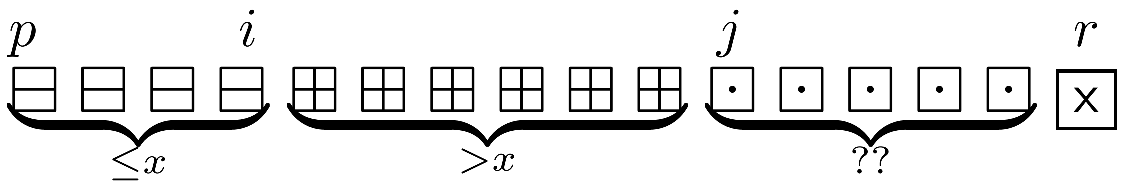
```
int partition (int A[], int p, int r)
{
  x = A[r];
  i = p-1;
  for (j=p ; j<r ; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j);
    }
  swap(A, i+1, r);
  return i+1;
}

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:

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

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).