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

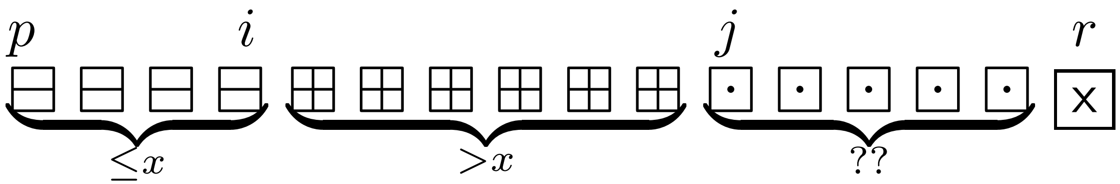
```
int partition (int A[], int p, int r)
{
  x = A[r];
  i = p-1;
  for (j=p ; j<r ; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j);
    }
  swap(A, i+1, r);
  return i+1;
}

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 *terminação* $(j = r)$

⇒ 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);
  }
}
```

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 "quick-sort" 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:

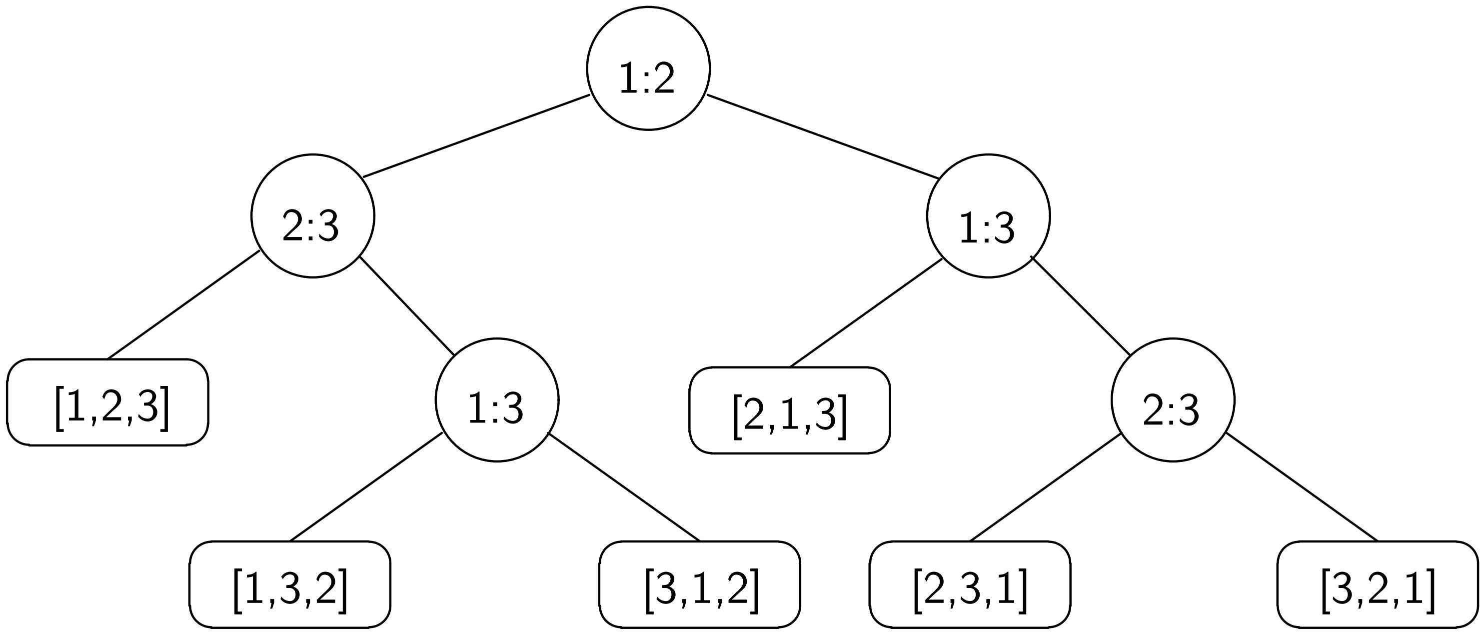
- corresponde a um teste de comparação entre dois elementos da sequência;
- tem como sub-árvore esquerda (resp. direita) a árvore correspondente à continuação da execução do algoritmo caso o teste resulte verdadeiro (resp. falso).

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

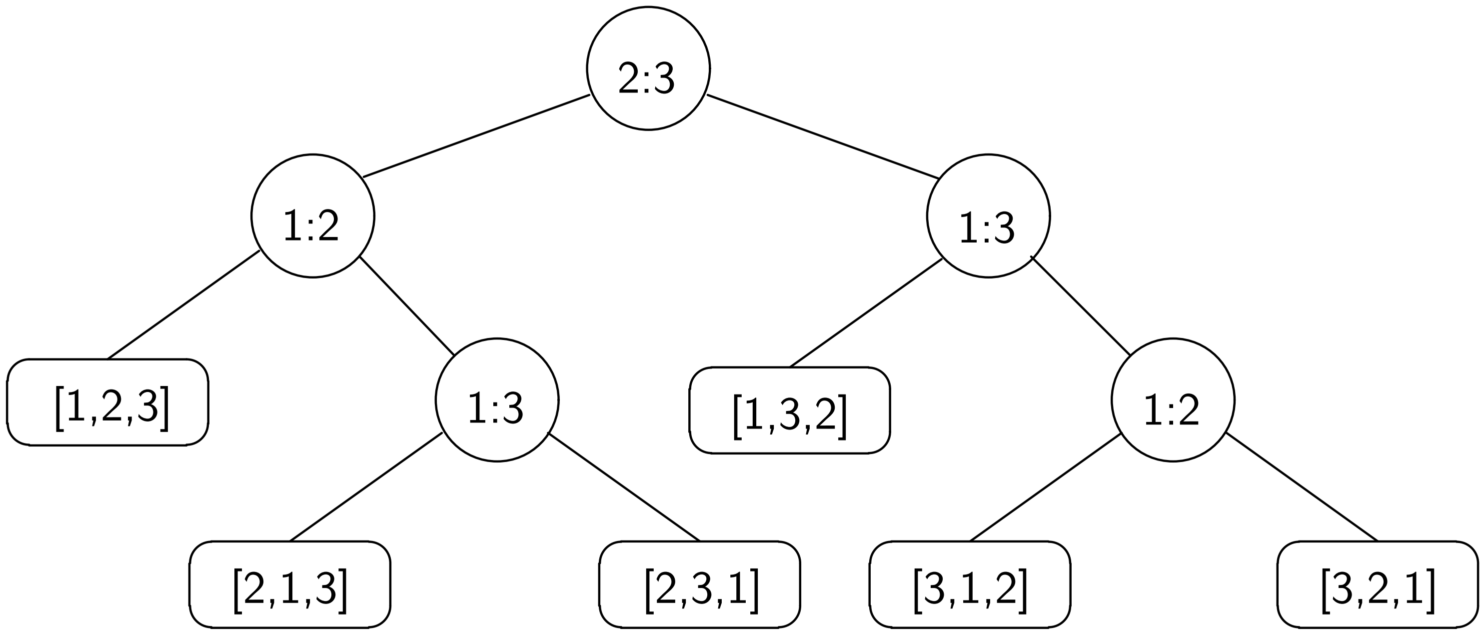
A execução concreta do algoritmo para um determinado vector de input corresponde a um *caminho da raíz para uma folha*, ou seja, uma sequência de comparações.

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;
  }
}
```

⇒ 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++)`	$\Theta(k)$
`    C[i] = 0;`	
`  for (j=1 ; j<=N ; j++)`	$\Theta(N)$
`    C[A[j]] = C[A[j]]+1;`	
`  for (i=1 ; i<=k ; i++)`	$\Theta(k)$
`    C[i] = C[i]+C[i-1];`	
`  for (j=N ; j>=1 ; j--) {`	$\Theta(N)$
`    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:

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

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.