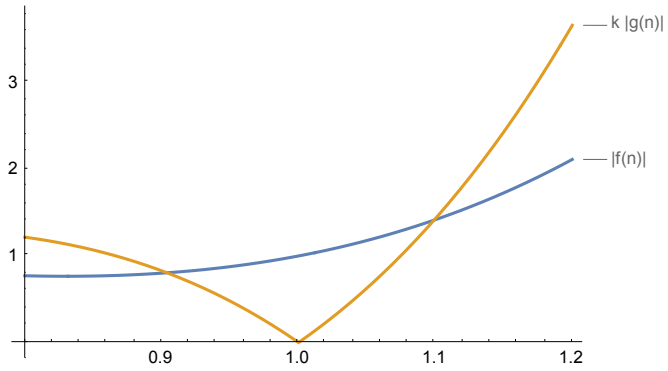
Aula 4 e 5
Análise de Complexidade

Algoritmos de ordenação

Def. Uma função f(n)∈O(g(n)) se $\exists_{k>0}\ \exists_{n_0>0}\ \forall_{n>n_0}\ |f(n)| < k\,|g(n)|$



Nota: Recorde que me Java todas operações aritméticas e booleanas são de tempo O(1), pois os tipos são limitados. O mesmo não acontece em Mathematica onde os inteiros são arbitrariamente longos.

Exercício

a) Mostre que se f(n)∈O(g(n)) sse $\limsup_{n\to\infty} \frac{|f(n)|}{|g(n)|} < \infty$.

R:

⇒) $\limsup_{n\to\infty} \frac{|f(n)|}{|g(n)|} = \lim_{n\to\infty} \left(\sup_{m>n} \left\{ \frac{|f(m)|}{|g(m)|} \right\}\right) \leq$

$\sup_{m>n_0} \left\{ \frac{|f(m)|}{|g(m)|} \right\} \leq \sup_{m>n_0} \left\{ \frac{k\,|g(m)|}{|g(m)|} \right\} = k < \infty$

⇐) $\lim_{n\to\infty} \left(\sup_{m>n} \left\{ \frac{|f(m)|}{|g(m)|} \right\}\right) = k'$ logo qq $\varepsilon>0$, existe n_0 tal que qq $n > n_0$ temos $\frac{|f(m)|}{|g(m)|} < k'+\varepsilon$

escolha-se um ε ao nosso gosto, existe $k=k'+\varepsilon$ e

existe n_0 tal que qq $n > n_0$ $|f(m)| < (k'+\varepsilon)\,|g(m)|$!

b) Mostre que se p(n) é um polinómio (positivo) de grau k então p(n) ∈ $O(n^k)$.

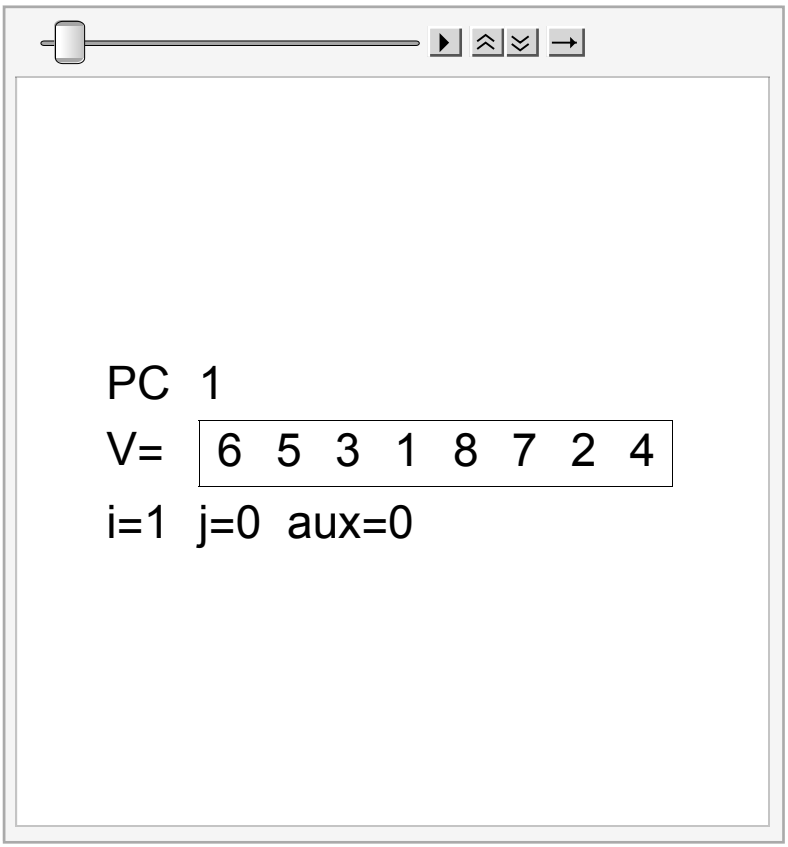
R:

$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

Usando o exercício anterior basta verificar
$\lim_{n\to\infty} \frac{p(n)}{n^k} = a_k < \infty$!

Insertion Sort

O InsertionSort é um dos algoritmos de ordenação mais simples.



O código do Insertionsort em Java é o seguinte:

```
public static void insertionSort (int V[]) {
1 int i = 1, j, aux, n = V.length;
2 while (i < n) {
3   aux = V[i];
4   j = i - 1;
5   while (j >= 0 && V[j] > aux) {
6       V[j + 1] = V[j];
7       j--;}
8   V[j + 1] = aux;
9   i++;}
10 }
```

Análise da Complexidade do InsertionSort

Exercício: Analise a complexidade do Insertionsort no caso em que apenas o último elemento está desordenado, e que pode ser colocado equiprovavelmente em qualquer posição.

i = n - 1 ->, neste caso pode tomar valores j=-1 até i-2= n-3

j∈ -1... n-3 ou seja j pode tomar n-3+1+1=n-1 valores! $P(J = j) = \frac{1}{n-1}$

m(j) =(n-2)-j

Complexidade do 2º ciclo quando i=n-1

$\sum_{j=-1}^{n-3} P(J = j)$(k+m(j)*2k)

$\sum_{j=-1}^{n-3} \frac{1}{n-1}$(k+((n-2)-j)*2k)

Complexidade do corpo do 1º ciclo

3k+$\sum_{j=-1}^{n-3} \frac{1}{n-1}$(k+((n-2)-j)*2k)

i <= n - 2 -> o corpo do 1 º ciclo (passo 3 ao 9) demora 3 k (e com a guarda 4k)

$$2\,k + \left(\sum_{i=1}^{n-2} 4\,k\right) + 3\,k + \frac{1}{n-1}\sum_{j=-1}^{n-3} (k + ((n-2) - j) * 2\,k)$$

$$5\,k + 4\,k\,(-2+n) + \frac{k + 2\,k\,(-1+n) + k\,(-2+n)\,n}{-1+n}$$

```
Expand[%]
```

$$-3\,k - \frac{k}{-1+n} + 4\,k\,n + \frac{k\,n^2}{-1+n}$$

Logo é o tempo é O(n)!

Caso médio:
Assuma que no segundo ciclo poderá terminar com j=-1,0,….,i-1 equiprovavelmente.
O objetivo é calcular a complexidade média.

P(J=j)= $\frac{1}{(i+1)}$

O valor esperado de tempo do 2 º ciclo é :

$$\sum_{j=-1}^{i-1} P(J = j) * O(\text{SegCiclo} \mid J = j)$$

$$\frac{1}{(i+1)}\left(\sum_{j=-1}^{i-1} O(\text{SegCiclo} \mid J = j)\right)$$

note - se que j é o valor que a variável j termina

Falta relacionar o valor com que j termina e o número de iteradas m do 2º ciclo.
m(j) =(i-1)-j

(k+m(j)*2k) -> corresponde à complexidade quando são feitas m(j) iteradas

$$\frac{1}{(i+1)}\left(\sum_{j=-1}^{i-1} (k + m(j) * 2\,k)\right)$$

$$\frac{1}{(i+1)}\left(\sum_{j=-1}^{i-1} (k + ((i-1) - j) * 2\,k)\right)$$

Falta introduzir o 1º ciclo

$$k + k + \sum_{i=1}^{n-1} \left(3\,k + \left(\frac{1}{(i+1)} \left(\sum_{j=-1}^{i-1} (k + ((i-1) - j) * 2\,k)\right)\right)\right)$$

$$2\,k + \frac{1}{2}\left(-8\,k + 7\,k\,n + k\,n^2\right)$$

```
Expand[%]
```

$$-2\,k + \frac{7\,k\,n}{2} + \frac{k\,n^2}{2}$$

Logo , o caso médio é $O(n^2)$

Melhor caso:

$$2\,k + \sum_{i=1}^{n-1} 4\,k$$

```
Expand[2 k + 4 k (-1 + n)]
```

$$-2\,k + 4\,k\,n$$

Logo é de tempo linear, ou seja O(n)

Pior caso:

$$2\,k + \sum_{i=1}^{n-1} \left(4\,k + \sum_{j=0}^{i-1} 2\,k\right) = 2\,k + \sum_{i=1}^{n-1} (4\,k + 2\,k\,i) = 2\,k + 4\,k\,(n-1) + 2\,k \sum_{i=1}^{n-1} i$$

$$-2\,k + 3\,k\,n + k\,n^2$$

Logo O(n^2)

$$\sum_{j=0}^{n} j$$

$$\frac{1}{2}\,n\,(1+n)$$

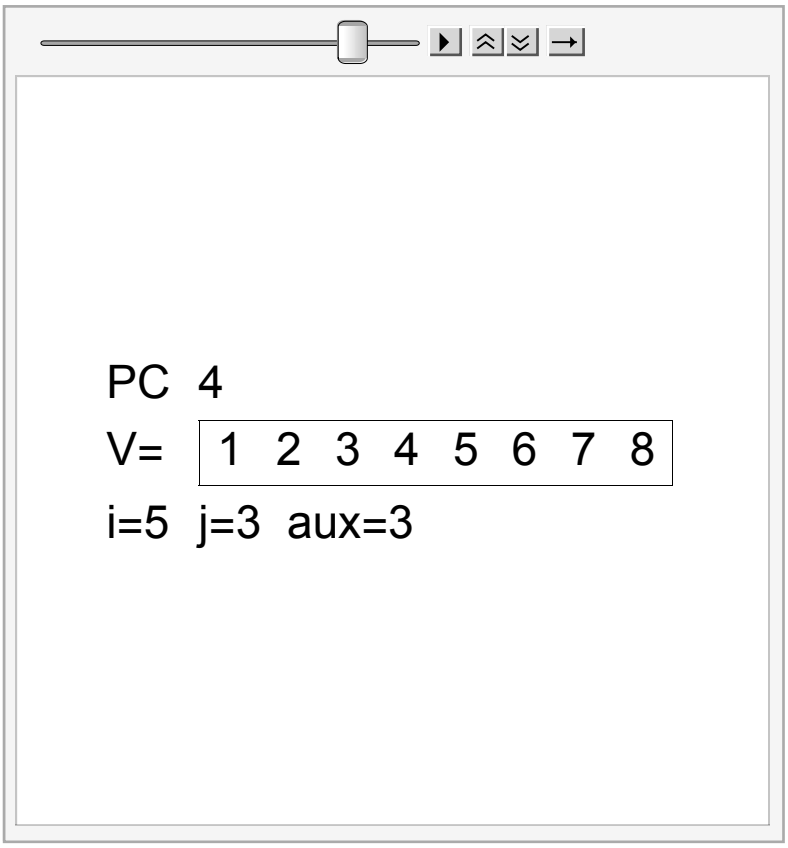
Onde k é um limite superior para a execução de 6;7, 8;9, 3;4, e 1; mais, é ainda um limite superior à avaliação das guardas dos while's

Caso médio:
Assuma que no segundo ciclo poderá terminar com j=-1,0,….,i-1 equiprovavelmente.
O objectivo é calcular a complexidade média.

Exercício: Analise a complexidade do Insertionsort no caso em que apenas o último elemento está desordenado, e que pode ser colocado equiprovalvemente em qualquer posição.

Bubble Sort

Outro algoritmo muito simples de ordenação é o BubbleSort



Exercício: Encontre os invariantes relevantes para o Bubblesort.

```
public static void BubbleSort (int V[])){
1  int i=1, j, aux, n = V.length;
2  while (i < n) {// V[n-i+1]..V[n-1]
                  //     temos os i-1
               //maiores valores de V ordenados
3  j = 0;
4  while (j < n-i) {  //I1 & V[j] tem a maior
                      //V[0] até V[j]
5   if (V[j+1] < V[j]) {
6       aux = V[j];
7       V[j] = V[j+1];
8       V[j+1] = aux;}
9     j++;}
10 i++;}}
```

Análise da Complexidade do BubbleSort

```
public static void BubbleSortM (int V[])){
  int i=1, j, aux, n = V.length;
  while (i < n) {
  j = 0;
  while (j < n-i) {
    if (V[j+1] < V[j]) {
        aux = V[j];
        V[j] = V[j+1];
        V[j+1] = aux;}
     j++;}
```

```
  i++;}}
```

Pior caso:

$O(1)+\sum_{i=1}^{n-1}(O(1)+\sum_{j=0}^{n-i-1}O(1))=O(\sum_{i=1}^{n-1}\sum_{j=0}^{n-i-1}1)=$

$O(\sum_{i=1}^{n-1}n-i)=$

$O(n^2)$

Alguns factos óbvios:
O(1)+O(f(n))=O(f(n)) com f(n)>k

$\sum_{k=1}^{n}O(f(k))=O(\sum_{k=1}^{n}f(k))$

Melhor caso:
$O(n^2)$

Caso médio:
$O(n^2)$

Exercício: Modifique o BubbleSort para que seja linear no melhor caso.

```
public static void BubbleSortM (int V[])){
  int i=1, j, aux, n = V.length;
  boolean t=true;
  while (i < n && t) {
  j = 0;
  t=false;
  while (j < n-i) {
    if (V[j+1] < V[j]) {
        t=true;
        aux = V[j];
        V[j] = V[j+1];
        V[j+1] = aux;}
     j++;}
  i++;}}
```

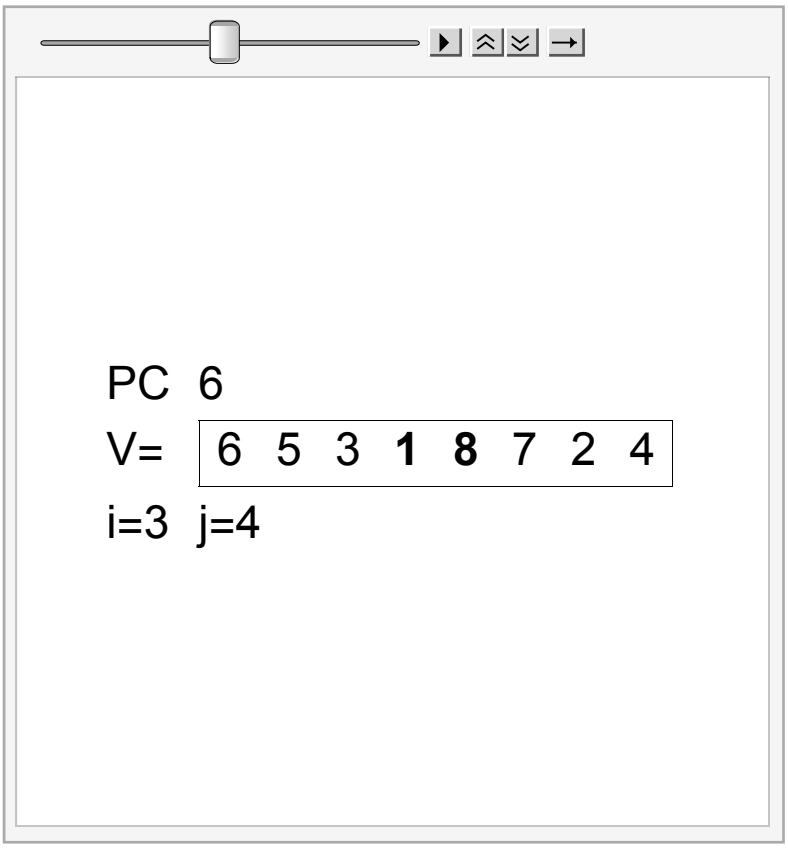
Quick Sort

O QuickSort é um algoritmo muito eficiente que se deve a Hoare. O algoritmo é recursivo e baseia-se num método de partição.

```
public static void swap(int vec[],int a,int b) {//troca elem da posição a com posição
b - O(1)
  int tmp = vec[a];
  vec[a] = vec[b];
  vec[b] = tmp;
}

public static int partition(int vec[], int left, int right) {
1  int i=left, j=left + 1;
2  while(j<=right) {
3    if (vec[j]<=vec[left]){
4      i++;
5      swap(vec,i,j);}
```

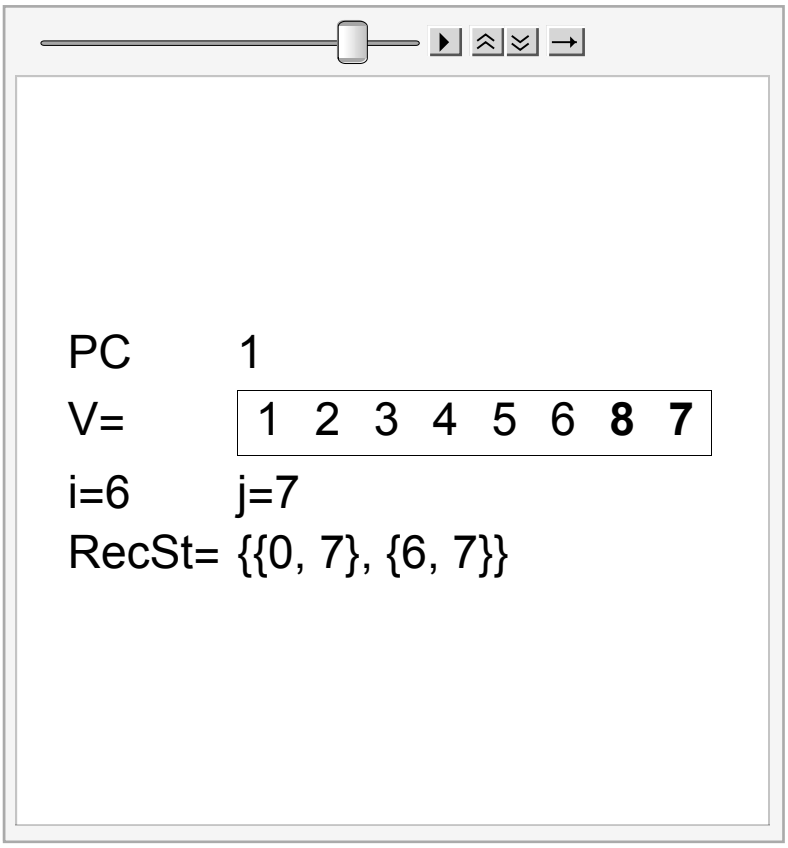
```
6    j++;}
7  swap(vec,left, i);
8  return i;
}

public static void quickSort(int vec[], int left, int right) { //sobreposição em Java
  int r;
  if (right > left) {
    r = partition(vec,left,right);
    quickSort(vec, left, r - 1);
    quickSort(vec, r + 1, right);
  }
}

public static void quickSort(int vec[]) {
    quickSort(vec, 0, vec.length-1);
}
```

Comecemos por examinar o método de partição.



O método particiona o vetor em duas partes, a parte com elementos menores ou iguais ao pivot (vec[left]) e a parte como elementos superiores. A nova posição do pivot é o valor retornado pela função. Esta posição é depois usada para continuar a ordenar o vector.

Assim facilmente se obtém o Quicksort:



A demonstração da correção do QuickSort é feita por indução, assumindo a correção do partition.

Teo: O Algoritmo QuickSort ordena o vetor por ordem crescente.

Prova: Prova por indução completa

Base Vec tem comprimento 0 ou 1, done!
HI Quicksort ordena qq vec de tamanho <n
Tese QuickSort ordena corretamente um vetor de dimensão n?

Assumindo que partition está correto, logo após o if o vec tem todos elementos menores (ou iguais) que vec[i] de vec[0] a vec[r-1] e todos os maiores que vec[i] de vec[r+1] até vec[n-1], por HI, o Quicksort ordena bem todos os tamanho <n, logo vai ordenar bem de 0 a r-1 e de r+1 até n-1, pois estes vetores têm tamanho < que n.
Done!

Vamos agora estudar as ferramentas para analisar a complexidade de algoritmos recursivos.

Def. Uma função f(n)∈Θ(g(n)) se
f(n)∈O(g(n)) e g(n)∈O(f(n))

Exemplo: seja p(n) um polinómio positivo de grau k, então
$p(n) \in \Theta(n^k)$.

As árvores de recursão permitem visualizar o número de chamadas recursivas e o custo em cada chamada.

Considere a por exemplo a recursão $T(n) = 2\,T(n/2) + n^2$

Da análise da árvore concluímos que

$T(n)=\sum_{i=0}^{\log(n)} n^2/2^i$

```
Sum[r^i, {i, 1, t}]
```

$$\frac{r\left(-1+r^t\right)}{-1+r}$$

```
Sum[n^2 / 2^i, {i, 0, Log[2, n]}]
```

```
Expand[%]
```

$$-n+2\,n^2$$

Considere agora $T(n) = T(n/3) + T(2\,n/3) + n$

$T(n)=n\sum_{i=1}^{\log_{3/2}(n)} O(1)$