DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Algoritmos Elementares de Ordenação

Sedgewick: Capítulo 6

IAED

Algoritmos Elementares de Ordenação

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shell Sort

Motivação

- Porquê estudar algoritmos elementares ?

- Razões de ordem prática
  - Fáceis de codificar e por vezes suficientes
  - Rápidos/eficientes para problemas de dimensão média e por vezes os melhores em certas situações

- Razões pedagógicas
  - Bom exemplo para aprender terminologia e contexto dos problemas a codificar e por vezes suficiente
  - Alguns são fáceis de generalizar para algoritmos mais eficientes ou para melhorar o desempenho de outros algoritmos

Análise de Desempenho

- Parâmetro de interesse - tempo de execução
  - regra: tempo $O(N^2)$ para ordenar N items
  - mas se N for pequeno podem ser os melhores

- Mas desempenho em memória também interessa
  - ordenação *in-place*
  - utilizando memória adicional

Algoritmo Estável

- Um algoritmo de ordenação é dito **estável** se preserva a ordem relativa dos items com chaves repetidas
  - ex: ordenar lista de alunos, já previamente ordenada por nome, por ano de graduação

- Algoritmos elementares são normalmente estáveis, mas poucos algoritmos avançados o são

Algoritmo Interno

- Um algoritmo de ordenação é dito **interno** se o conjunto de todos os dados a ordenar couber em memória RAM; caso contrário é dito externo

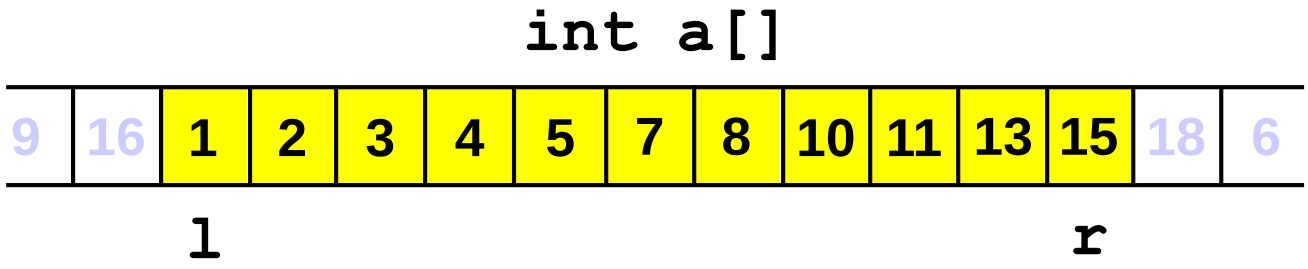
- Distinção muito importante:
  - ordenação interna pode aceder a qualquer dado com um custo muito pequeno
  - ordenação externa tem de aceder aos dados de forma sequencial (ou em blocos)

- Vamos estudar apenas algoritmos de ordenação interna

Algoritmos de Ordenação - Implementação

- Função **`sort()`** implementa o algoritmo de ordenação
- Devolve tabela de **`int`** ordenada, entre as posições `l` e `r`

```
void sort(int a[], int left, int right);
```



Algoritmos de Ordenação - Implementação

```
#include <stdio.h>

int main() {
  int i, a[]={10,9,8,7,6,5,4,3,2,1};

  sort(a, 0, 9);

  for (i = 0; i < 10; i++)
    printf("%3d ", a[i]);
  printf("\n");

  return 0;
}
```

Algoritmos de Ordenação – outro exemplo

```
#include <stdio.h>
#include <stdlib.h>
#define N 10000

int main() {
  int i, a[N];

  for (i = 0; i < N; i++)
    a[i] = 1000*(1.0*rand()/RAND_MAX);

  sort(a, 0, N-1);

  for (i = 0; i < N; i++)
    printf("%3d ", a[i]);
  printf("\n");
  return 0;
}
```

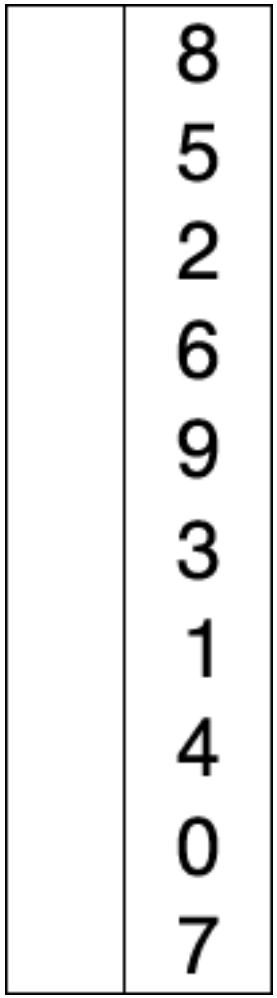
Número aleatório (int) entre 0 e 999

Número aleatório (float) entre 0 e 1

Resta-nos definir a função sort ☺

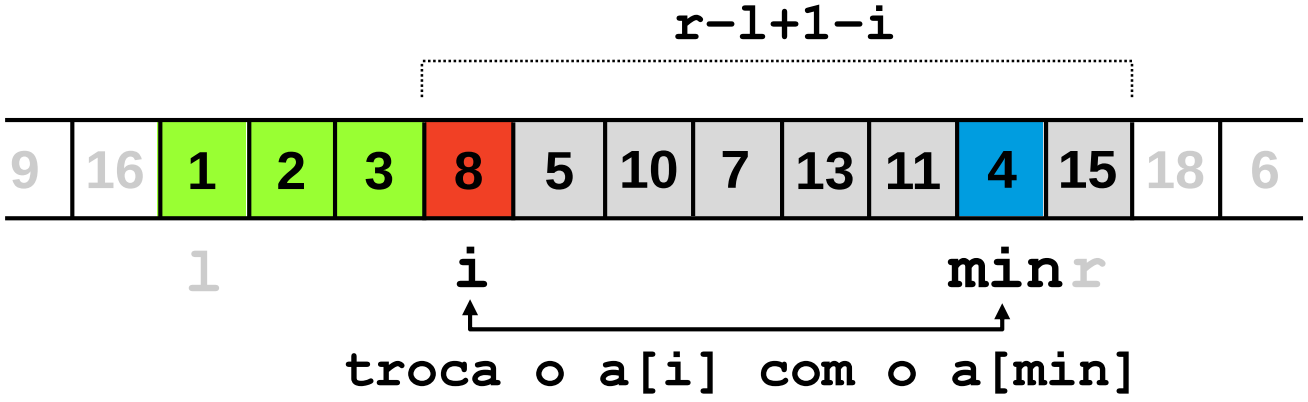
Selection Sort

- Para cada elemento `i` entre as posições `l` e `r`
  - Procura o menor elemento entre `i` e `r`
  - Se o menor valor — guardado na posição `min` — for menor que o valor guardado na posição `i`,
    - troca o `a[i]` com o `a[min]`



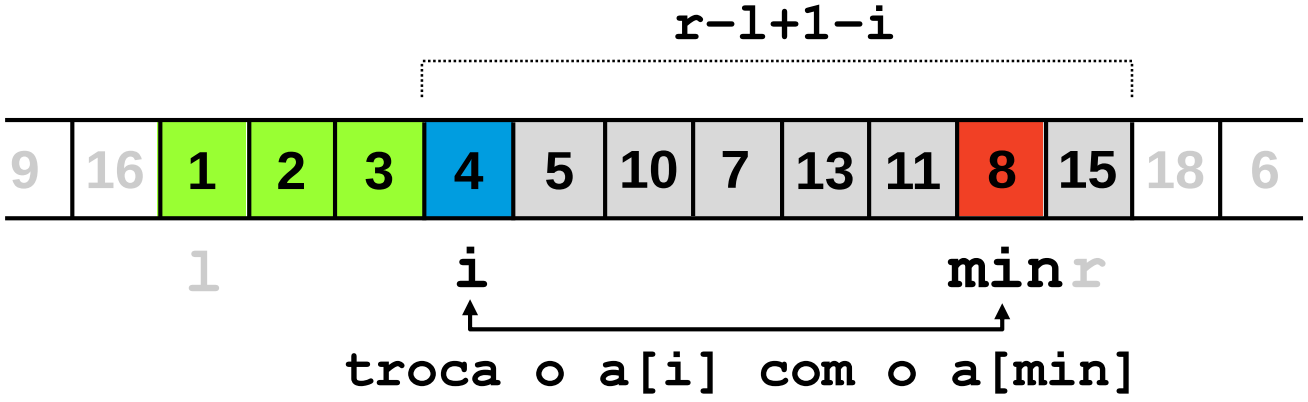
Selection Sort

- A cada passo, escolher o menor entre os `r-l+1-i` maiores elementos
- Para cada `i`, os primeiros `i` elementos ficam já na sua posição final
- Para cada valor de `i` do primeiro ciclo, segundo ciclo é executado `r-i` vezes



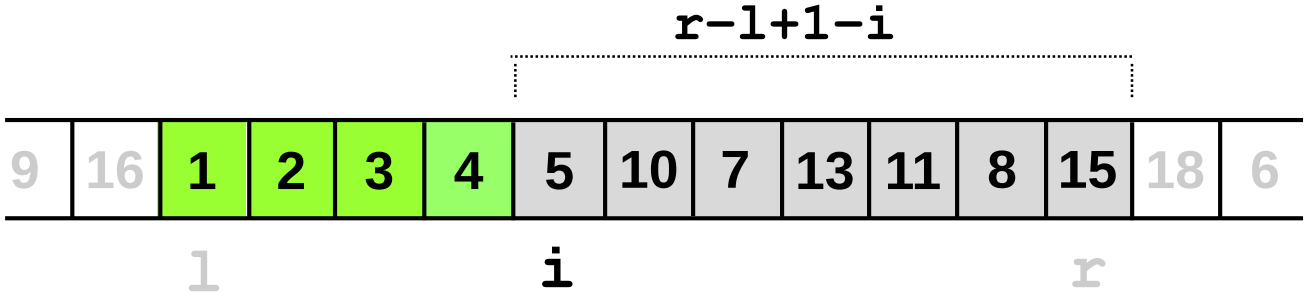
Selection Sort

- A cada passo, escolher o menor entre os `r-l+1-i` maiores elementos
- Para cada `i`, os primeiros `i` elementos ficam já na sua posição final
- Para cada valor de `i` do primeiro ciclo, segundo ciclo é executado `r-i` vezes



Selection Sort

- A cada passo, escolher o menor entre os `r-l+1-i` maiores elementos
- Para cada `i`, os primeiros `i` elementos ficam já na sua posição final
- Para cada valor de `i` do primeiro ciclo, segundo ciclo é executado `r-i` vezes



Exercício

- Considere o seguinte vector

`v[ ] = {72, 29, 38, 22, 60, 2}`

Indique o conteúdo de v no final de cada passo do algoritmo ***selection sort***.

Inicio:	72	29	38	22	60	2
Passo 1:	2	29	38	22	60	72
Passo 2:	2	22	38	29	60	72
Passo 3:	2	22	29	38	60	72
Passo 4:	2	22	29	38	60	72
Final:	2	22	29	38	60	72

Selection Sort

```
void SelectionSort(int a[], int left, int right) {
    int i, j;
    for (i = left; i < right; i++) {
        int aux, min = i;
        for (j = i+1; j <= right; j++)
            if (a[j] < a[min])
                min = j;

        aux = a[i];
        a[i] = a[min];
        a[min] = aux;
    }
}
```

Procura o mínimo

Troca elementos

Selection Sort

```
void SelectionSort(int a[], int left,
    int i, j;
    for (i = left; i < right; i++) {
        int aux, min = i;
        for (j = i+1; j <= right; j++)
            if (a[j] < a[min])
                min = j;

        aux = a[i];
        a[i] = a[min];
        a[min] = aux;
    }
}
```

Estas variáveis apenas existem dentro deste bloco

Algoritmos de Ordenação – Abstrações Úteis

- Items ordenados por uma "chave"
- Características específicas de cada item ou chave
- No entanto cada algoritmo tem comportamento igual :
  Logo, vamos usar abstrações!!

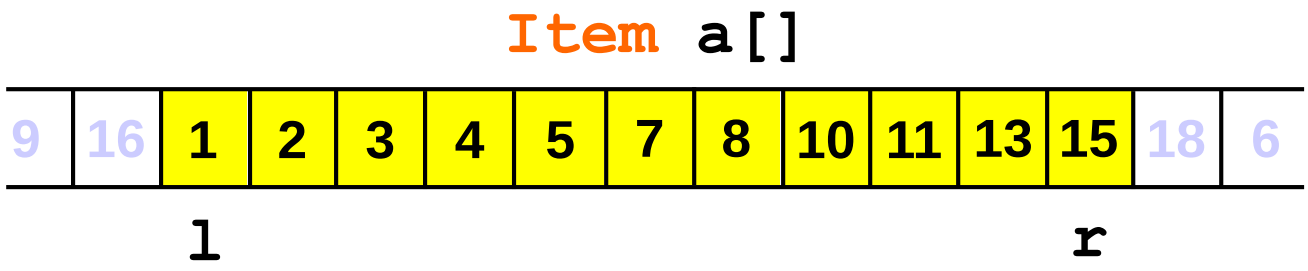
```
typedef int Item;
#define key(A) (A)
#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)
```

A característica do "A" que será utilizada para a ordenação será, neste caso, o próprio A

Algoritmos de Ordenação – Abstrações Úteis

- Função **`sort()`** implementa o algoritmo de ordenação
- Devolve tabela de **`Item`** ordenada, entre as posições `l` e `r`

```
void sort(Item a[], int left, int right);
```



Selection Sort

```
void selection(Item a[], int left, int right) {
  int i, j;

  for (i = left; i < right; i++) {
    int min = i;
    for (j = i+1; j <= right; j++)
      if (less(a[j], a[min]))
        min = j;
    exch(a[i], a[min]);
  }
}
```

Algoritmos de Ordenação – Abstrações Úteis

```
#define N 10000
typedef int Item;
#define key(A) (A)
#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)

void selection(Item a[], int left, int right);

int main() {
   int i, a[N];

   for (i = 0; i < N; i++)
        a[i] = 1000*(1.0*rand()/RAND_MAX);

   selection(a, 0, N-1);
   (...)
}
```

Exercício

- Considere o seguinte vector

```
a[ ] = {22, 33, 1, 10, 11, 2}
```

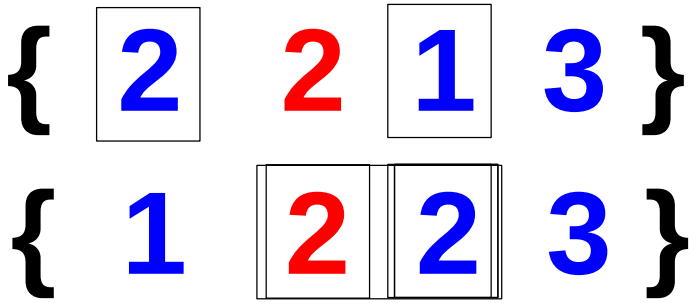
Indique o conteúdo de **`a`** depois de 3 iterações do algoritmo ***selection sort***.

Inicio:	22	33	1	10	11	2
	1	33	22	10	11	2
	1	2	22	10	11	33
R:	**1**	**2**	**10**	**22**	**11**	**33**

Selection Sort

- Tempo de execução:
  - Comparações: $N^2 / 2$ Trocas: N
  - No pior caso é $O(N^2)$, com $N = \texttt{r-l}$
  - No melhor caso, é $O(N^2)$, com $N = \texttt{r-l}$

- O algoritmo é estável?
  - Será que a ordem relativa de chaves duplicadas é mantida?



Alterámos a ordem relativa de chaves duplicadas!

Selection Sort

- Tempo de execução:
  - Comparações: $N^2 / 2$, trocas: N
  - No pior caso é $O(N^2)$, com $N = \texttt{r-l}$
  - No melhor caso, é $O(N^2)$, com $N = \texttt{r-l}$

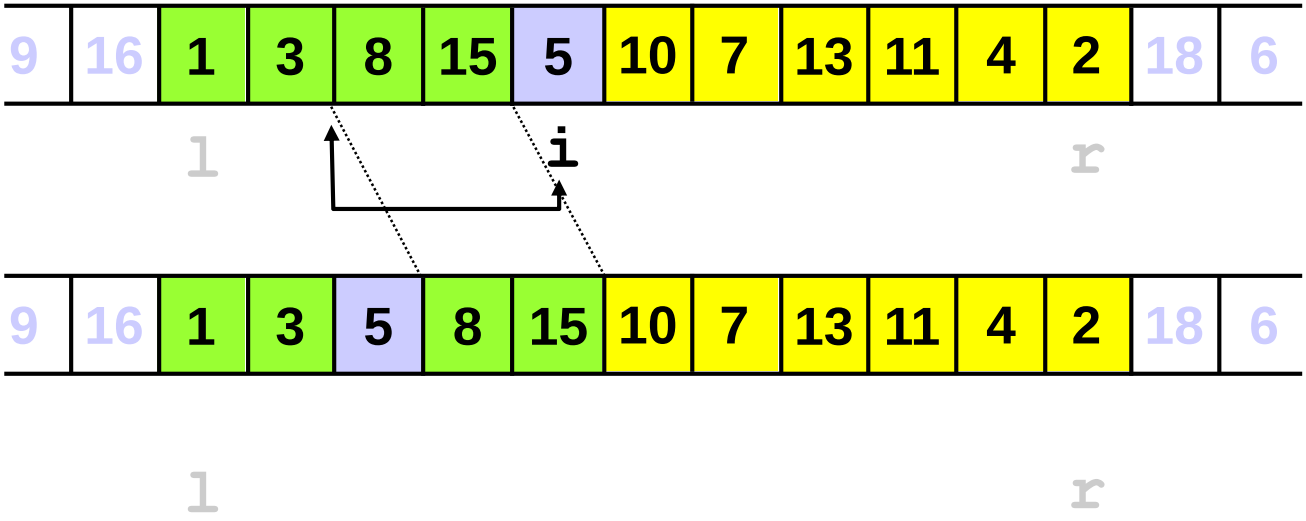
- O algoritmo ***não é estável***
  - i.e. ordem relativa de chaves duplicadas não é mantida
  - Podemos alterar o algoritmo para que este fique estável.

Insertion Sort (ideia geral)

6	5	3	1	8	7	2	4
l							r

Insertion Sort (algoritmo)

- Para cada `i`, os primeiros `i` elementos ficam ordenados, embora possam ainda não ficar na sua posição final
- Isso significa que, se o vector já estiver ordenado, fazemos apenas *N-1* comparações!



Exercício

- Considere o seguinte vector

```
v[ ] = {72, 29, 38, 22, 60, 2}
```

Indique o conteúdo de **v** no final de cada passo do algoritmo ***insertion sort***.

Início:	* 72	29	38	22	60	2
Passo 1:	29 *	72	38	22	60	2
Passo 2:	* 29	38	72	22	60	2
Passo 3:	22	29	38 *	72	60	2
Passo 4:	* 22	29	38	60	72	2
Final:	**2**	**22**	**29**	**38**	**60**	**72**

Insertion Sort

```
void insertion(Item a[], int left, int right) {
  int i,j;

  for (i = left+1; i <= right; i++) {
       Item v = a[i];
       j = i-1;
       while ( j >= left && less(v, a[j])) {
              a[j+1] = a[j];
              j--;
       }
       a[j+1] = v;
   }
}
```

Uso uma variável auxiliar v onde guardo o valor de a[i]

O j vai percorrer o vector a partir da posição i-1 até encontrar um elemento menor que v

Enquanto v<a[j] vou puxando os valores para a direita

Quando deixar de ser, significa que encontrei o "slot" para o v

Insertion Sort

- Tempo de execução:
  - No pior caso é $O(N^2)$, com $N = \texttt{r-l}$, i.e. vector já ordenado por ordem inversa
  - No ***melhor caso*** é $O(N)$, com $N = \texttt{r-l}$, i.e. vector já ordenado

- Algoritmo é **estável**
  - i.e. ordem relativa de chaves duplicadas é mantida

Exercício (Verdadeiro ou Falso)

- O tempo de execução do InsertionSort é $\Omega(N)$.
- O tempo de execução do InsertionSort é $\Theta(N^2)$.
- O número de trocas no algoritmo InsertionSort é $O(N)$.
- O número de trocas no algoritmo SelectionSort é $O(N)$.

Exercício (Verdadeiro ou Falso)

- O tempo de execução do InsertionSort é $\Omega(N)$.

  Verdadeiro.

  O tempo de execução no melhor caso é $\Omega(N)$, logo o Insertion Sort é $\Omega(N)$.

- O tempo de execução do InsertionSort é $\Theta(N^2)$.

  Falso.

  $\Theta(N^2)$ sse $\Omega(N^2)$ e $O(N^2)$. Como o insertionSort não é $\Omega(N^2)$ não pode ser $\Theta(N^2)$.

- O número de trocas no algoritmo InsertionSort é $O(N)$.

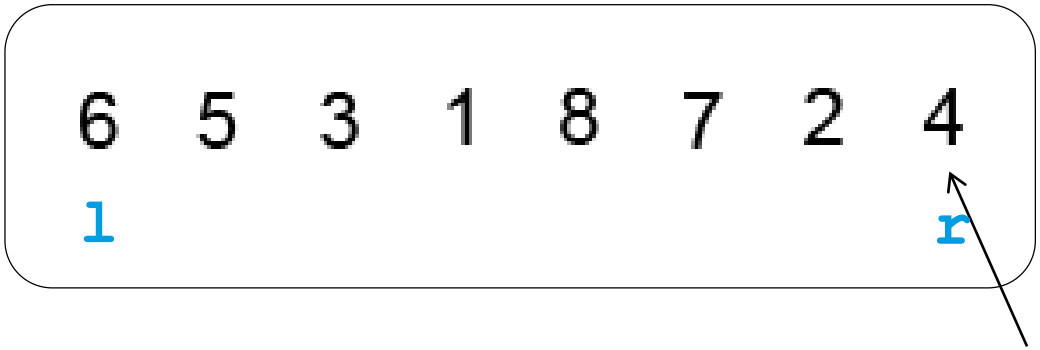
  Falso.

  O número de trocas é $O(N^2)$.

- O número de trocas no algoritmo SelectionSort é $O(N)$.

  Verdadeiro.

Bubble Sort



Em cada iteração `i` há sempre um elemento
(o `r-i`)
que fica na sua posição final

Bubble Sort

- Para cada valor de `i` no primeiro ciclo, o segundo ciclo é executado `r-i` vezes
- Para cada `i`, os **últimos** `i` elementos ficam já na sua posição final
- Só são efectuadas trocas entre elementos adjacentes

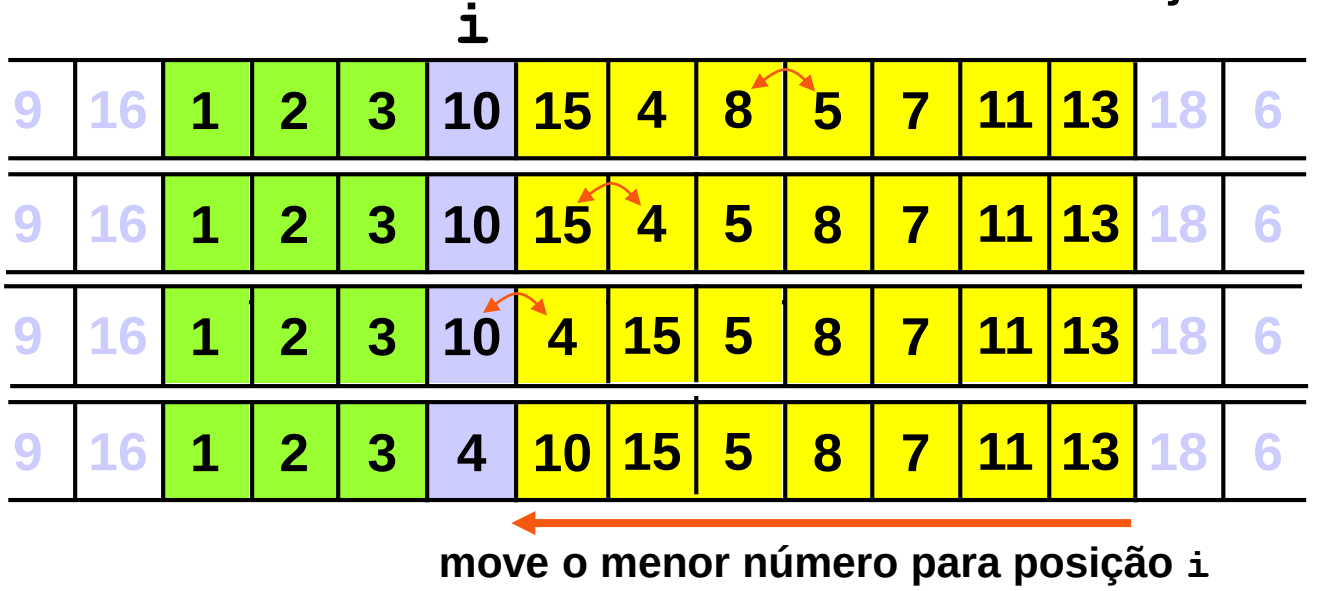
Bubble Sort

```
void bubble(Item a[], int left, int right) {
   int i, j;

   for (i = left; i < right; i++)
      for (j = left; j < right + (left-i); j++)
         compexch(a[j], a[j+1]);
}
```

Bubble Sort (direita para a esquerda)

- Para cada valor de `i` no primeiro ciclo, o segundo ciclo é executado `r-i` vezes
- Para cada `i`, os **primeiros** `i` elementos ficam já na sua posição final
- Só são efectuadas trocas entre elementos adjacentes



Bubble Sort (direita para a esquerda)

```
void bubble(Item a[], int left, int right) {
  int i, j;

  for (i = left; i < right; i++)
    for (j = right; j > i; j--)
      compexch(a[j-1], a[j]);
}
```

(esquerda para a direita)

```
void bubble(Item a[], int left, int right) {
 int i, j;

 for (i = left; i < right; i++)
   for (j = left; j < right+(left-i); j++)
        compexch(a[j], a[j+1]);
}
```

Exercício: Será que não haverá maneira de reduzir o número de iterações, em particular quando o vector já está parcialmente ordenado?

Bubble Sort (direita para a esquerda + condição de paragem)

```
void bubble(Item a[], int left, int right) {
    int i, j, done;

    for (i = left; i < right; i++){
        done=1;
        for (j = right; j > i; j--)
            if (less(a[j], a[j-1])){
                exch(a[j-1], a[j]);
                XXXXX ;
            }
        YYYYY;
    }
}
```

Bubble Sort (direita para a esquerda + condição de paragem)

```
void bubble(Item a[], int left, int right) {
    int i, j, done;

    for (i = left; i < right; i++){
        done=1;
        for (j = right; j > i; j--)
            if (less(a[j], a[j-1])){
                exch(a[j-1], a[j]);
                done=0;
            }
        YYYYY;
    }
}
```

Bubble Sort (direita para a esquerda + condição de paragem)

```
void bubble(Item a[], int left, int right) {
    int i, j, done;

    for (i = left; i < right; i++){
        done = 1;
        for (j = right; j > i; j--)
            if (less(a[j], a[j-1])){
                exch(a[j-1], a[j]);
                done = 0;
            }
        if (done) break;
    }
}
```

Usamos uma variável auxiliar done a 1

Se fizermos uma troca, alteramos done para 0

Se não fizermos nenhuma troca, então o vector está ordenado E done continua a 1

Exercício

- Considere o seguinte vector

`v[ ] = {72, 29, 38, 22, 60, 2}`

Indique o conteúdo de **v** no final 2 passagens do algoritmo ***bubble sort (esquerda para a direita)***.

Primeira passagem:

72	**29**	38	22	60	2
29	**72**	**38**	22	60	2
29	38	**72**	**22**	60	2
29	38	22	**72**	**60**	2
29	38	22	60	**72**	**2**
29	**38**	**22**	**60**	**2**	**72**

Exercício

- Considere o seguinte vector

`v[ ] = {72, 29, 38, 22, 60, 2}`

Indique o conteúdo de **`v`** no final 2 passagens do algoritmo ***bubble sort (esquerda para a direita)***.

Segunda passagem: **29 38** 22 60 2 72

29 **38 22** 60 2 72

29 22 **38 60** 2 72

29 22 38 **60 2** 72

29 22 38 2 60 72

Comparação

- Pior Caso

	Selection	Insertion	Bubble
Comparações	$N^2 / 2$	$N^2 / 2$	$N^2 / 2$
Trocas Chaves	N	$N^2 / 2$	$N^2 / 2$

- Caso Médio

	Selection	Insertion	Bubble
Comparações	$N^2 / 2$	$N^2 / 4$	$N^2 / 2$
Trocas Chaves	N	$N^2 / 4$	$N^2 / 2$

Comparação

- Tabelas com poucos elementos fora de ordem
  - Insertion e Bubble Sort são quase lineares
  - os melhores algoritmos de ordenação podem ser quadráticos
- Contexto onde elementos são grandes e chaves pequenas
  - Selection Sort é linear no número de dados
  - N dados com tamanho M palavras
  - custo comparação: 1; custo troca: M
  - $N^2 / 2$ comparações e NM custo de trocas
  - termo NM domina: custo proporcional ao tempo necessário para mover os dados
- Alternativa: uso de ponteiros (que tb vamos aprender!)

Avaliação Experimental

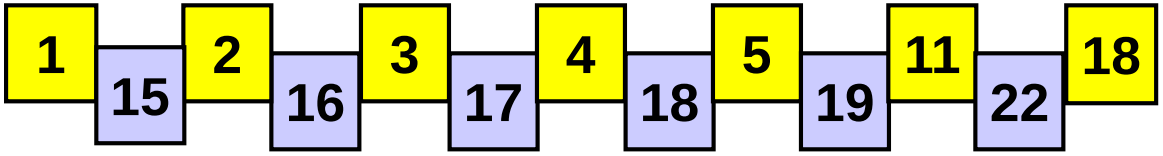
N	Selection	Insertion	Bubble
1000	5	4	11
2000	21	15	45
4000	85	62	182

- Bubble/Insertion Sort são lentos: trocas ocorrem apenas entre items adjacentes
  - se o menor item está no final da tabela, serão precisos N passos para o colocar na posição correcta
- **Shellsort**:
  - acelerar o algoritmo permitindo trocas entre elementos que estão afastados
  - bubble/insertion sort, mas com elementos distanciados de h

Shell Sort - Definições

- Vector diz-se h-ordenado se qualquer sequência de números separados por h posições está ordenada

- Vector é equivalente a h sequências ordenadas entrelaçadas



- O resultado de h-ordenar um vector que está k-ordenado, é um vector que está h-ordenado e k-ordenado

Shell Sort - Ideia

- Rearranjar os dados de forma a que estejam h-ordenados
- Usando valores de h grandes é possível mover elementos na tabela distâncias grandes

- Ordenar primeiro para valores de h grandes e depois para valores de h pequenos
  - facilita as últimas ordenações, para valores de h pequenos

- Utilizando este procedimento para qualquer sequência de h's que termine em 1 produz no final uma tabela ordenada

- Cada passo torna o próximo mais simples

Shell Sort

```
void shellsort(Item a[], int left, int right)
{
  int i, j, h;
  for (h = 1; h <= (right-left)/9; h = 3*h+1)
    ;
  for ( ; h > 0; h /= 3)
    for (i = left+h; i <= right; i++) {
      int j = i;
      Item v = a[i];
      while (j >= left+h && less(v, a[j-h])) {
        a[j] = a[j-h];
        j -= h;
      }
      a[j] = v;
    }
}
```

gera sequência *h*: 1 4 13 40 121 364 1093 3280 ...

executado para valores de *h*, por ordem inversa

InsertionSort com saltos de tamanho *h* *(para h=1, obtemos o método original)*

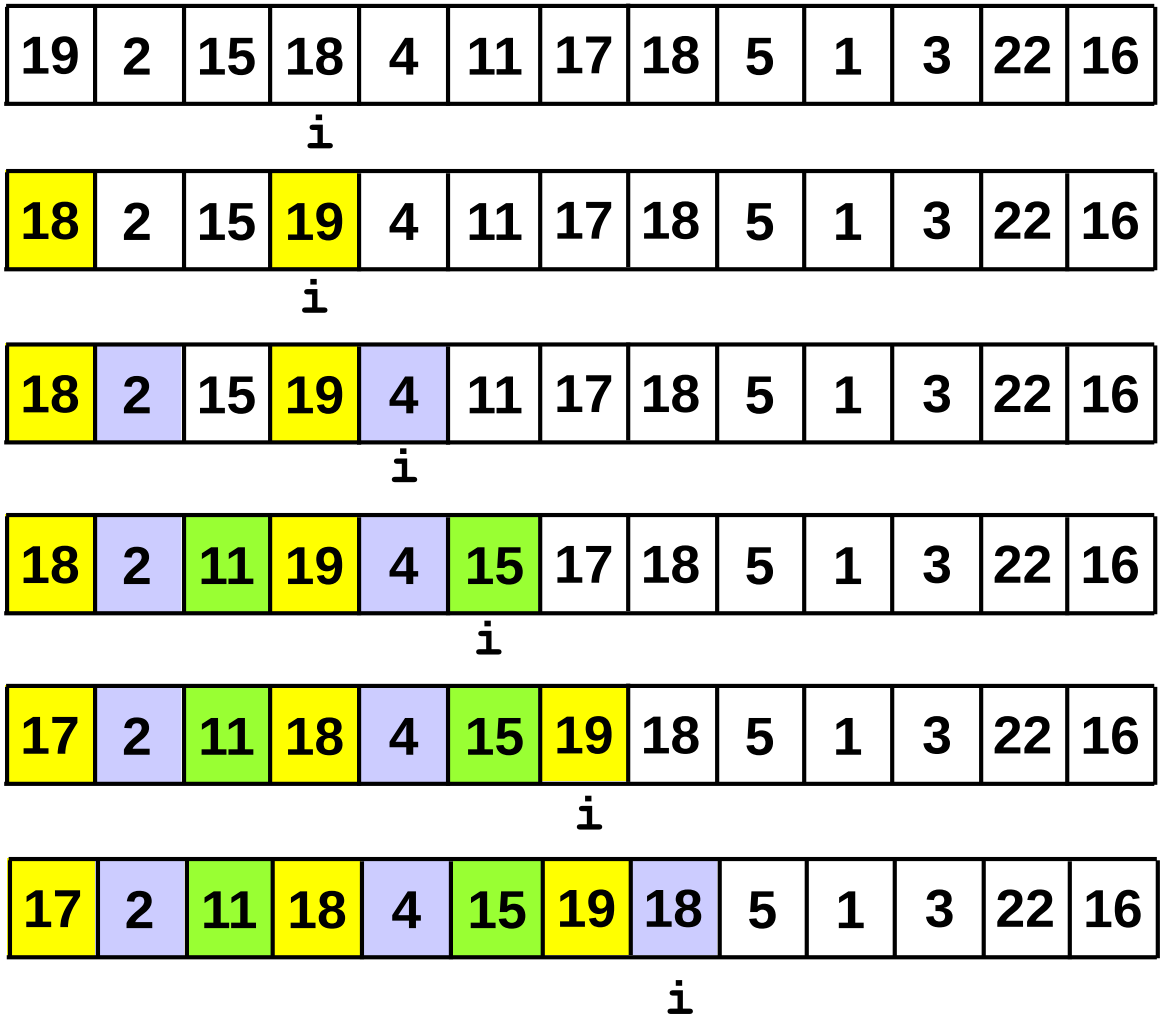
Shell Sort

```
void shellsort(Item a[], int left
{
  int i, j, h;
  for (h = 1; h <= (right-left)/9
    ;
  for ( ; h > 0; h /= 3)
    for (i = left+h; i <= right; i++) {
      Item v = a[i];
      int j = i;
      while (j >= left+h && less(v, a[j-h])) {
        a[j] = a[j-h];
        j -= h;
      }
      a[j] = v;
    }
}
```

```
void insertion(Item a[], int left, int right)
{ int i,j;
  for (i = left+1; i <= right; i++) {
          Item v = a[i];
          j = i-1;
          while ( j >= left && less(v, a[j]))
          {
                    a[j+1] = a[j];
                    j--;
          }
          a[j+1] = v;
    }
}
```

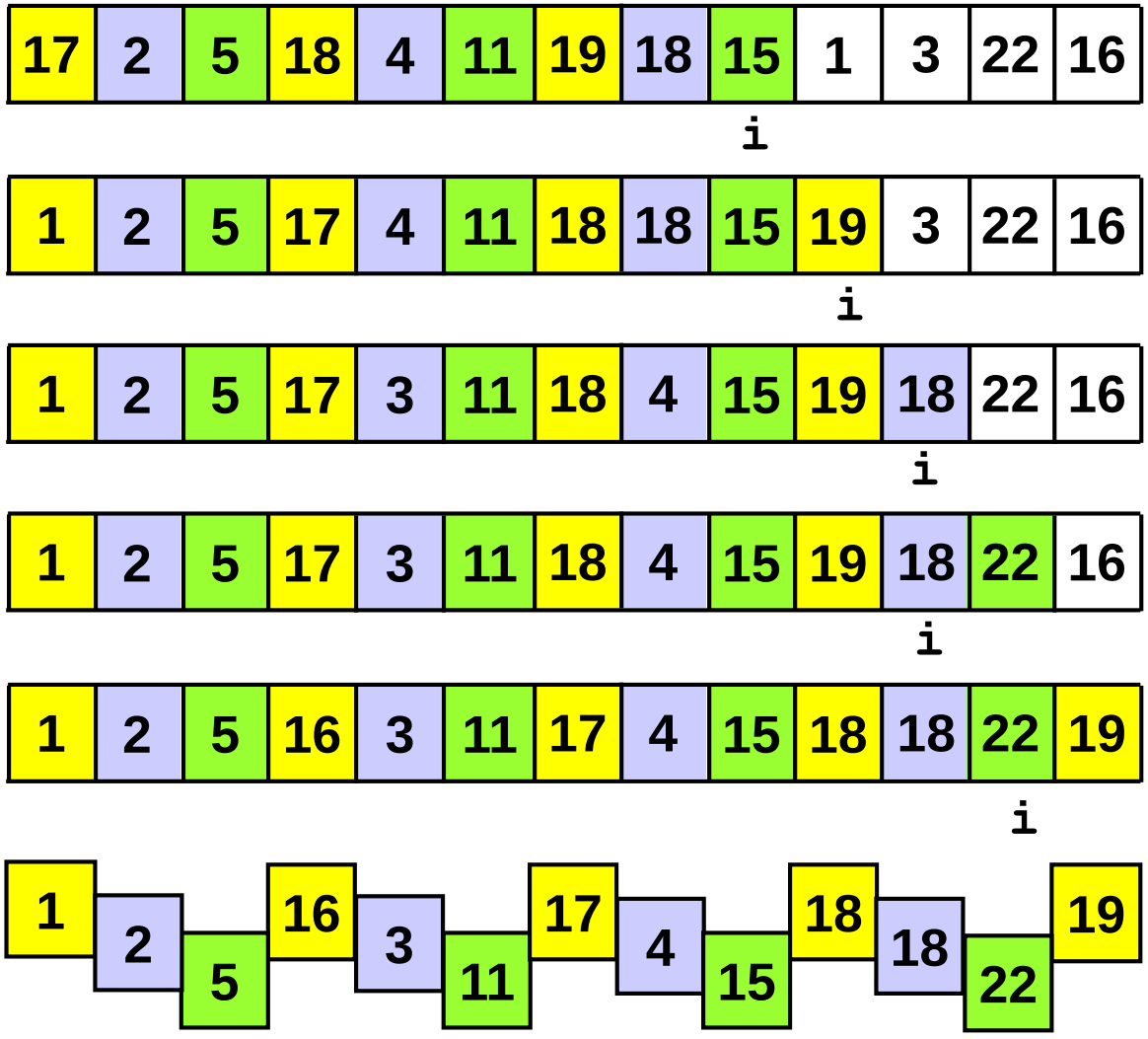
Shell Sort

$h = 3$



Shell Sort

$h = 3$



3-ordenado

Shell Sort - Funcionamento

- Operação (vector com tamanho 150):
  - Para cada valor de h, 40, 13, 4, 1:
    - Utilizar Insertion sort para criar h sub-vectores ordenados dentro de vector com tamanho 150
  - Vector fica h-ordenado
  - Para h = 40 existem 40 sub-vectores ordenados, cada um com 3/4 elementos
  - Para h = 13 existem 13 sub-vectores ordenados, cada um com 11/12 elementos
  - ...
  - Para h = 1 existe 1 (sub-)vector ordenado, com 150 elementos

Escolha da Sequência de Ordenação

- Questão difícil de responder
- Propriedades de muitas sequê
- Possível provar que uma melhores que outras
  - 1, 4, 13, 40, 121, 36 , 1093, 3280, ... (Knuth, $3N+1$)
  - melhor que 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ...(Shell, 2^i) Porquê?
  - mas pior (20%) que 1, 8, 23, 77, 281, 1073, ... ($4^{i+1}+3\times 2^i+1$)
- Na prática utilizam-se sequências que decrescem geometricamente para que o número de incrementos seja logarítmico
- A sequência óptima ainda não foi descoberta

elementos nas posições pares não são comparados com elementos nas posições ímpares até ao passo final

Shell Sort - Complexidade

- Análise rigorosa da complexidade do algoritmo é desconhecida

- Complexidade depende da sequência de valores *h* utilizada
  - Sequência 1, 4, 13, 40, 121, 364, ... ~ $O(N^{3/2})$ comparações
  - Sequência 1, 8, 23, 77, 281, 1073, ... ~ $O(N^{4/3})$ comparações
  - Sequência 1, 2, 3, 4, 6, 9, 8, 12, 18, ... ~ $O(N(\log N)^2)$ comparações

Shell Sort - Avaliação Experimental

N	O	K	G	S	P	I
12500	16	6	6	5	6	6
25000	37	13	11	12	15	10
50000	102	31	30	27	38	26
100000	303	77	60	63	81	58
200000	817	178	137	139	180	126

- O: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ...
- K: 1, 4, 13, 40, 121, 364, ...
- G: 1, 2, 4, 10, 23, 51, 113, 249, 548, ...
- S: 1, 8, 23, 77, 281, ...
- P: 1, 7, 8, 49, 56, 64, 343, 392, 448, 512, ...
- I: 1, 5, 19, 41, 109, 209, 505, 929, ...

Exercício

```
void shellsort(Item a[], int l, int r)
{
  int i, j, h;
  for (h = 1; h <= (r-l)/9; h = 3*h+1)
    ;
  for ( ; h > 0; h /= 3)
    for (i = l+h; i <= r; i++) {
      int j = i;
      Item v = a[i];
      while (j >= l+h && less(v, a[j-h]))
      {
        a[j] = a[j-h];
        j -= h;
      }
      a[j] = v;
    }
}
```

Suponha que a função shellsort é invocada com os seguintes argumentos:

```
a = { 16, 8, 4, 19, 20, 5,
13, 11, 6, 12 },
l = 0, r = 9
```

Indique qual o conteúdo do vector `a` durante a execução da função shellsort após o primeiro valor de `h` ter sido considerado.

Exercício

```
void shellsort(Item a[], int l, int r)
{
  int i, j, h;
  for (h=1; h <= (r-l)/9; h = 3*h+1)
    ;
  for ( ; h > 0; h /= 3)
    for (i = l+h; i <= r; i++) {
      int j = i;
      Item v = a[i];
      while (j >= l+h && less(v, a[j-h]))
      {
        a[j] = a[j-h];
        j -= h;
      }
      a[j] = v;
    }
}
```

Começa com h=1, depois vai para h=4 e pára... Logo o primeiro valor de h será 4

Suponha que a função shellsort é invocada com os seguintes argumentos:

a = { 16, 8, 4, 19, 20, 5, 13, 11, 6, 12 },
l = 0, r = 9

Indique qual o conteúdo do vector **a** durante a execução da função shellsort após o primeiro valor de **h** ter sido considerado.

Agora resta-me aplicar o insertion sort com h=4. Como faço isso?

Exercício

```
a = { 16, 8, 4, 19, 20, 5, 13, 11, 6, 12 }
```

h=4

```
16, 8, 4, 19, 20, 5, 13, 11, 6, 12
```

Resultado:

```
6, 5, 4, 11, 16, 8, 13, 19, 20, 12
```

O Shell Sort é estável? **NÃO!!**