Algoritmos de Pesquisa e Ordenação em Vectores

Pesquisa Sequencial

- Problema *(pesquisa de valor em vector)*:
  - Verificar se um valor existe no vector e, no caso de existir, indicar a sua posição.
  - Possíveis variantes para o caso de vectores com valores repetidos:
    - (a) indicar a posição da primeira ocorrência
    - (b) indicar a posição da última ocorrência
    - (c) indicar a posição de uma ocorrência qualquer
- Algoritmo *(pesquisa sequencial)*:
  - Percorrer sequencialmente todas as posições do vector, da primeira para a última [(a)] ou da última para a primeira [(b)], até encontrar o valor pretendido ou chegar ao fim do vector
    - [(a)] caso se pretenda saber a posição da primeira ocorrência
    - [(b)] caso se pretenda saber a posição da última ocorrência
- Adequado para vectores não ordenados ou pequenos

Implementação da Pesquisa Sequencial em C++

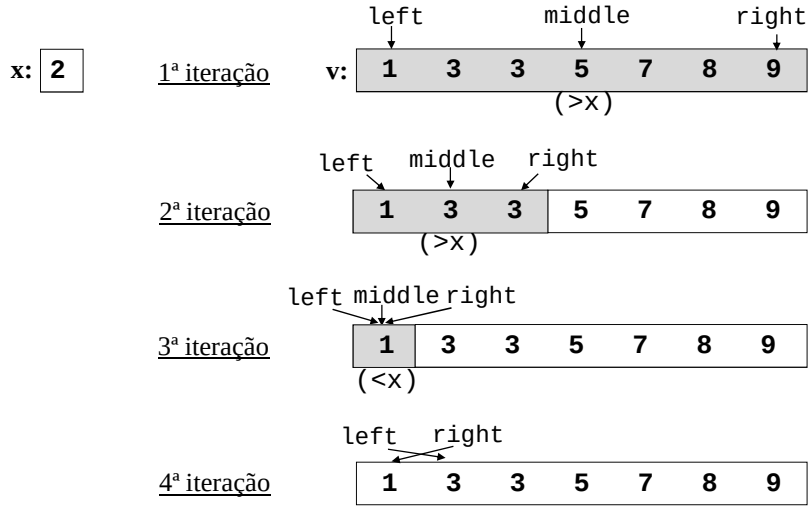
Template de função em C++, na variante (a):

```
/* Procura um valor x num vector v. Retorna o índice da primeira
ocorrência de x em v, se encontrar; senão, retorna -1. Supõe que os
elementos do array são comparáveis com o operador de igualdade ( == )
*/

template <class Comparable>
int sequentialSearch(const vector<Comparable> &v, Comparable x)
{
   for (int i = 0; i < v.size(); i++)
      if (v[i] == x)
         return i; // encontrou

   return -1; // não encontrou
}
```


Pesquisa Binária

- Problema (*pesquisa de valor em vector ordenado*):
  - verificar se um valor (*x*) existe num vector (*v*) previamente ordenado e, no caso de existir, indicar a sua posição
  - no caso de vectores com valores repetidos, consideramos a variante em que basta indicar a posição de uma ocorrência qualquer
- Algoritmo (*pesquisa binária*):
  - Comparar o valor que se encontra a meio do vector com o valor procurado, podendo acontecer uma de três coisas:
    - é igual ao valor procurado ⇒ está encontrado
    - é maior do que o valor procurado ⇒ continuar a procurar (do mesmo modo) no sub-vector à esquerda da posição inspeccionada
    - é menor do que o valor procurado ⇒ continuar a procurar (do mesmo modo) no sub-vector à direita da posição inspeccionada.
  - Se o vector a inspeccionar se reduzir a um vector vazio, conclui-se que o valor procurado não existe.

Exemplo de Pesquisa Binária



vector a inspeccionar vazio ⇒ o valor 2 não existe no vector inicial !

Implementação da Pesquisa Binária em C++

```
/* Procura um valor x num vector v previamente ordenado. Retorna o
índice de uma ocorrência de x em v, se encontrar; senão, retorna -
1. Supõe que os elementos do array são comparáveis com os
operadores de comparação ( ==, < ) */

template <class Comparable>
int binarySearch(const vector<Comparable> &v, Comparable x)
{
    int left = 0, right = v.size() - 1;
    while (left <= right)
    {
        int middle = (left + right) / 2;
        if (x == v[middle]) return middle;  // encontrou
        else if (x < v[middle]) right = middle - 1;
        else left = middle + 1;
    }
    return -1;  // não encontrou
}
```

Ordenação de Vectores

- Problema (*ordenação de vector*)
  - Dado um vector (*v*) com **N** elementos, rearranjar esses elementos por ordem crescente (ou melhor, por ordem não decrescente, porque podem existir valores repetidos)
- Ideias base:
  - Existem diversos algoritmos de ordenação com complexidade **$O(N^2)$** que são muito simples **(**por ex: Ordenação por Inserção, BubbleSort)
  - Existem algoritmos de ordenação mais difíceis de codificar que têm complexidade **O(N log N)**

Algoritmos de Ordenação de Vectores

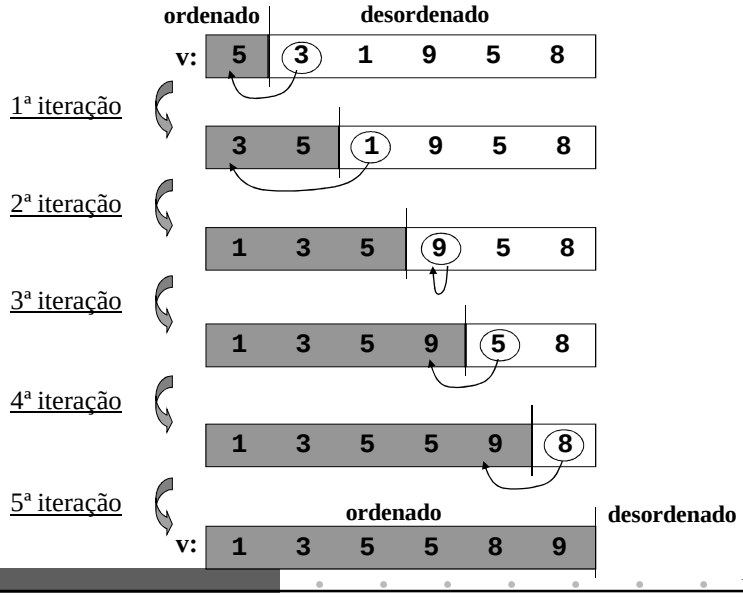
- Algoritmos:
  - Ordenação por Inserção - $O(N^2)$
  - Ordenação por Selecção - $O(N^2)$
  - BubbleSort - $O(N^2)$
  - ShellSort - $O(N^2)$ variante mais popular
  - MergeSort - O(N logN)
  - Ordenação por Partição (QuickSort) - O(N logN)
  - HeapSort - O(N logN)

Ordenação por Inserção

- Algoritmo iterativo de **N-1** passos
- Em cada passo *p:*
  - coloca-se um elemento na ordem, sabendo que elementos dos índices inferiores (entre **0** e **p-1**) já estão ordenados

- Algoritmo (*ordenação por inserção*):
  - Considera-se o vector dividido em dois sub-vectores (esquerdo e direito), com o da esquerda ordenado e o da direita desordenado
  - Começa-se com um elemento apenas no sub-vector da esquerda
  - Move-se um elemento de cada vez do sub-vector da direita para o sub-vector da esquerda, inserindo-o na posição correcta por forma a manter o sub-vector da esquerda ordenado
  - Termina-se quando o sub-vector da direita fica vazio

Ordenação por Inserção



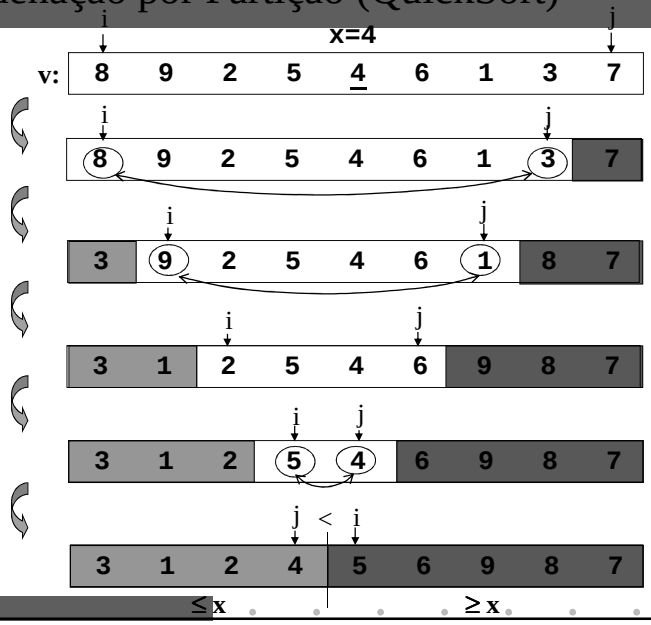
Ordenação por Inserção (implementação)

```
/* Ordena elementos do vector v. Supõe que os elementos
do vector possuem operadores de atribuição e comparação*/

template <class Comparable>
void insertionSort(vector<Comparable> &v)
{
    for (int i = 1; i < v.size(); i++)
    {
       Comparable tmp = v[i];
       int j;
       for (j = i; j > 0 && tmp < v[j-1]; j--)
          v[j] = v[j-1];
       v[j] = tmp;
    }
}
```


Ordenação por Partição (QuickSort)

- Algoritmo (ordenação por partição):
  1. Caso básico: Se o número (*n*) de elementos do vector (*v*) a ordenar for 0 ou 1, não é preciso fazer nada
  2. Passo de partição:
     2.1. Escolher um elemento arbitrário (**x**) do vector (chamado pivot)
     2.2. Partir o vector inicial em dois sub-vectores (esquerdo e direito), com valores ≤ **x** no sub-vector esquerdo e valores ≥ **x** no sub-vector direito
  3. Passo recursivo: Ordenar os sub-vectores esquerdo e direito, usando o mesmo método recursivamente
- Algoritmo recursivo baseado na técnica *divisão e conquista*
  - nota: quando parte do vector a ordenar é de dimensão reduzida, usa um método de ordenação mais simples (p.ex. 'insertionSort')

Ordenação por Partição (QuickSort)



Ordenação por Partição (QuickSort)

- Escolha pivot determina eficiência
  - *pior caso*: pivot é o elemento mais pequeno
    O(N^2)
  - *melhor caso*: pivot é o elemento médio
    O(N logN)
  - *caso médio*: pivot corta vector arbitrariamente
    O(N logN)

- Escolha do pivot
  - um dos elementos extremos do vector:
    - má escolha: O(N^2) se vector ordenado
  - elemento aleatório:
    - envolve uso de mais uma função pesada
  - mediana de três elementos (extremos do vector e ponto médio)
    - recomendado

Ordenação por Partição (QuickSort)

```
/* Ordena elementos do vector v. Supõe que os elementos do
vector possuem operadores de atribuição e comparação */


template <class Comparable>
void quickSort(vector<Comparable> &v)
{
   quickSort(v,0,v.size()-1);
}
```


Ordenação por Partição (QuickSort)

```
template <class Comparable>
void quickSort(vector<Comparable> &v, int left, int right)
{
   if (right-left <= 20)    // se vector pequeno
      insertionSort(v,left,right);
   else {
      Comparable x = median3(v,left,right);  // x é o pivot
      int i = left; int j = right-1;    // passo de partição
      for(; ; ) {
         while (v[++i] < x);
         while (x < v[--j]);
         if (i < j)
            swap(v[i], v[j]);
         else break;
      }
      swap(v[i], v[right-1]);  //repoe pivot
      quickSort(v, left, i-1);
      quickSort(v, i+1, right);
   }
}
```

Ordenação por Partição (QuickSort)

```
/* determina o valor do pivot como sendo a mediana de 3 valores:
elementos extremos e central do vector */

template <class Comparable>
Comparable &median3(vector<Comparable> &v, int left, int right)
{
   int center = (left+right) /2;
   if (v[center] < v[left])
      swap(v[left], v[center]);
   if (v[right] < v[left])
      swap(v[left], v[right]);
   if (v[right] < v[center])
      swap(v[center], v[right]);
  //coloca pivot na posicao right-1
  swap(v[center], v[right-1]);
  return v[right-1];
}
```


Ordenação por Partição (QuickSort)

```
/* quando parte do vector a ordenar é de dimensão reduzida, usa um
método de ordenação mais simples 'insertionSort' */

template <class Comparable>
void insertionSort(vector<Comparable> &v, int left, int right)
{
    for (int i = left+1; i < right+1; i++)
    {
       Comparable tmp = v[i];
       int j;
       for (j = i; j > 0 && tmp < v[j-1]; j--)
          v[j] = v[j-1];
       v[j] = tmp;
    }
}
```

Algoritmos de Ordenação

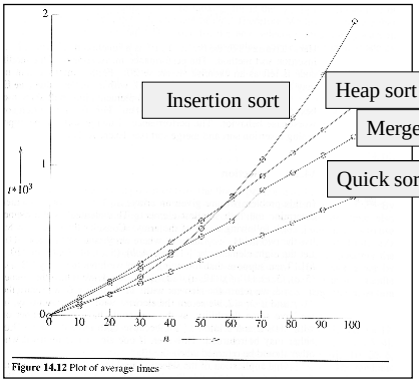


Figure 14.12 Plot of average times

• Cada ponto corresponde à ordenação de 100 vectores de inteiros gerados aleatoriamente

• Fonte: Sahni, "Data Structures, Algorithms and Applications in C++"

– Método de ordenação por partição (*quickSort*) é na prática o mais eficiente, excepto para arrays pequenos (até cerca 20 elementos), em que o método de ordenação por inserção (*insertionSort*) é melhor!

Algoritmos da STL

- Pesquisa sequencial em vectores:

  iterator find(iterator start, iterator end, const TYPE& val);

  procura a 1ª ocorrência entre *[start, end[* de um elemento idêntico a *val* (comparação efectuada pelo operador ==).
  - sucesso, retorna iterador para o elemento encontrado
  - não sucesso, retorna iterador para fim do vector (*v.end()*)

  iterator find_if(iterator start, iterator end, Predicate up);

  procura a 1ª ocorrência entre *[start, end[* para a qual o predicado unário *up* retorna verdadeiro.

- Pesquisa binária em vectores:

  bool binary_search(iterator start, iterator end, const TYPE& val);

  bool binary_search(iterator start, iterator end, const TYPE& val, Comp f);
  - necessário implementar o operador <

Algoritmos da STL

- Ordenação de vectores:

  void sort(iterator start, iterator end);

  ordena os elementos do vector entre *[start, end[* por ordem ascendente, usando o operador <

  void sort(iterator start, iterator end, StrictWeakOrdering cmp);

  ordena os elementos do vector entre *[start, end[* por ordem ascendente, usando a função *StrictWeakOrdering*

  – Algoritmo de ordenação implementado em *sort()* é o algoritmo *introsort*, possui complexidade O(N logN)

Algoritmos *sort* e *find* da STL (exemplo)

```
class Pessoa {
   string BI;
   string nome;
   int idade;
public:
   Pessoa (string BI, string nm="", int id=0);
   string getBI() const;
   string getNome() const;
   int getIdade() const;
   bool operator < (const Pessoa & p2) const;
   bool operator == (const Pessoa & p2) const;
};

Pessoa::Pessoa(string b, string nm, int id):
      BI(b),nome(nm), idade(id) {}

string Pessoa::getBI() const { return BI; }
string Pessoa::getNome() const { return nome; }
int Pessoa::getIdade() const { return idade; }
```

Algoritmos *sort* e *find* da STL (exemplo)

```
bool Pessoa::operator < (const Pessoa & p2) const
{ return nome < p2.nome; }

bool Pessoa::operator == (const Pessoa & p2) const
{ return BI == p2.BI; }


ostream & operator << (ostream &os, const Pessoa & p)
{
   os << "(BI: " << p.getBI() << ", nome: " << p.getNome()
<< ", idade: " << p.getIdade() << ")";
   return os;
}
```


Algoritmos *sort* e *find* da STL (exemplo)

```
template <class T>
void write_vector(vector<T> &v) {
       for (unsigned int i=0 ; i < v.size(); i++)
              cout << "v[" << i << "] = " << v[i] << endl;
       cout << endl;
}


bool compPessoa(const Pessoa &p1, const Pessoa &p2)
{ return p1.getIdade() < p2.getIdade(); }

bool eAdolescente(const Pessoa &p1)
{ return p1.getIdade() <= 20; }
```

Algoritmos *sort* e *find* da STL (exemplo)

```
int main()
{
  vector<Pessoa> vp;
  vp.push_back(Pessoa("6666666","Rui Silva",34));
  vp.push_back(Pessoa("7777777","Antonio Matos",24));
  vp.push_back(Pessoa("1234567","Maria Barros",20));
  vp.push_back(Pessoa("7654321","Carlos Sousa",18));
  vp.push_back(Pessoa("3333333","Fernando Cardoso",33));
  vector<Pessoa> vp1=vp;
  vector<Pessoa> vp2=vp;

  cout << "vector inicial:" << endl;
  write_vector(vp);
```

```
vector inicial:
v[0] = (BI: 6666666, nome: Rui Silva, idade: 34)
v[1] = (BI: 7777777, nome: Antonio Matos, idade: 24)
v[2] = (BI: 1234567, nome: Maria Barros, idade: 20)
v[3] = (BI: 7654321, nome: Carlos Sousa, idade: 18)
v[4] = (BI: 3333333, nome: Fernando Cardoso, idade: 33)
```


Algoritmos *sort* e *find* da STL (exemplo)

```
  sort(vp1.begin(),vp1.end());
  cout << "Apos 'sort' usando 'operador <':“ << endl;
  write_vector(vp1);
```

```
Apos 'sort' usando 'operador <':
v[0] = (BI: 7777777, nome: Antonio Matos, idade: 24)
v[1] = (BI: 7654321, nome: Carlos Sousa, idade: 18)
v[2] = (BI: 3333333, nome: Fernando Cardoso, idade: 33)
v[3] = (BI: 1234567, nome: Maria Barros, idade: 20)
v[4] = (BI: 6666666, nome: Rui Silva, idade: 34)
```

```
  sort(vp2.begin(),vp2.end(),compPessoa);
  cout << "Apos 'sort' usando funcao de comparacao:" << endl;
  write_vector(vp2);
```

```
Apos 'sort' usando funcao de comparacao:
v[0] = (BI: 7654321, nome: Carlos Sousa, idade: 18)
v[1] = (BI: 1234567, nome: Maria Barros, idade: 20)
v[2] = (BI: 7777777, nome: Antonio Matos, idade: 24)
v[3] = (BI: 3333333, nome: Fernando Cardoso, idade: 33)
v[4] = (BI: 6666666, nome: Rui Silva, idade: 34)
```

Algoritmos *sort* e *find* da STL (exemplo)

```
  Pessoa px("7654321");

  vector<Pessoa>::iterator it = find(vp.begin(),vp.end(),px);
  if (it==vp.end())
    cout << "Pessoa " << px << " nao existe no vector " << endl;
  else
    cout << "Pessoa " << px << " existe no vector como: " << *it
<< endl;
```

Pessoa (BI: 7654321, nome: , idade: 0) existe no vector como: (BI: 7654321, nome: Carlos Sousa, idade: 18)

```
  it = find_if(vp.begin(),vp.end(),eAdolescente);
  if (it==vp.end())
    cout << "Pessoa adolescente nao existe no vector " << endl;
  else
    cout << "pessoa adolescente encontrada: " << *it << endl;
}
```

pessoa adolescente encontrada: (BI: 1234567, nome: Maria Barros, idade: 20)