

Vetores: Algoritmos de Ordenação

Algoritmos e Estruturas de Dados

2019/2020



Ordenação

- Problema (*ordenação de vetor*)
 - Dado um vetor (v) com N elementos, reorganizar esses elementos por ordem crescente (ou melhor, por ordem não decrescente, porque podem existir valores repetidos)
 - vetor de N elementos
- 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 $O(N \log N)$



Ordenação

- Algoritmos:
 - Ordenação por Inserção - $O(N^2)$
 - Ordenação por Seleção - $O(N^2)$
 - Bubble Sort - $O(N^2)$
 - ShellSort - $O(N^2)$ variante mais popular
 - MergeSort - $O(N \log N)$
 - Ordenação por Partição (QuickSort) - $O(N \log N)$
 - HeapSort - $O(N \log N)$, a estudar mais tarde
 - BucketSort

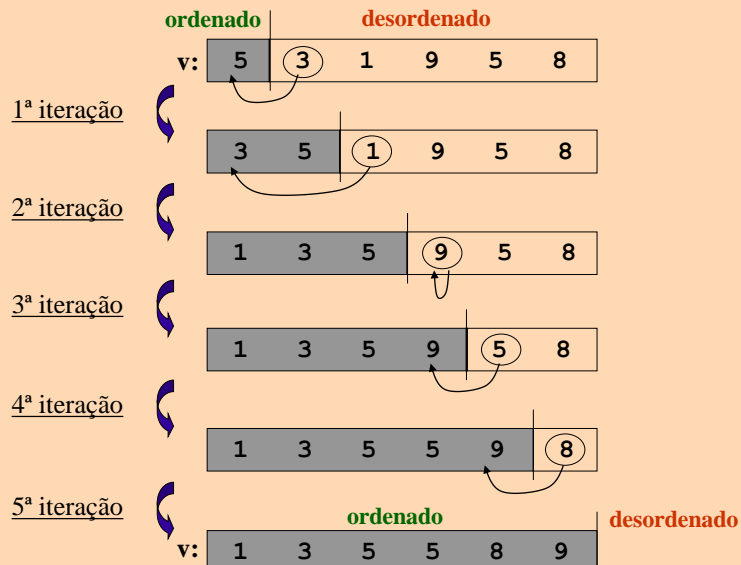


Ordenação por Inserção

- 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 o vetor dividido em dois sub-vetores (esquerdo e direito), com o da esquerda ordenado e o da direita desordenado
 - Começa com um elemento apenas no sub-vetor da esquerda
 - Move um elemento de cada vez do sub-vetor da direita para o sub-vetor da esquerda, inserindo-o na posição correta por forma a manter o sub-vetor da esquerda ordenado
 - Termina quando o sub-vetor da direita fica vazio



Exemplo de Ordenação por Inserção



Ordenação por Inserção usando `vector`

```
// Ordena elementos do vetor v
// Comparable: deve possuir construtor cópia, operadores
// atribuição (=) e comparação(<)

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



Eficiência da Ordenação por Inserção

- o nº de iterações do ciclo `for` interior é:
 - **1**, no melhor caso
 - **n**, no pior caso
 - **n/2**, em média
- o nº total de iterações do ciclo `for` exterior é:
 - no **melhor caso**, $1 + 1 + \dots + 1$ (n-1 vezes) = $n-1 \approx n$
 - no **pior caso**, $1 + 2 + \dots + n-1 = (n-1)(1 + n-1)/2 = n(n-1)/2 \approx n^2/2$
 - em **média**, metade do anterior, isto é, aproximadamente $n^2/4$

$\Rightarrow T(n) = O(n^2)$ (quadrático) (pior caso e caso médio)



Ordenação por Seleção

- Provavelmente é o algoritmo mais intuitivo:
 - Encontrar o mínimo do vetor
 - Trocar com o primeiro elemento
 - Continuar para o resto do vetor (excluindo o primeiro)
- 2 ciclos encaixados, cada um pode ter N iterações :
 - **Complexidade $O(N^2)$**
- Variantes:
 - “stableSort” – insere mínimo na primeira posição (em vez de realizar a troca)
 - “shaker Sort” – procura máximo e mínimo em cada iteração (seleção bidirecional)



Ordenação por Seleção

Algoritmo em C++:

```
// Ordena vetor v de n elementos, ficando v[0] ≤ ... ≤ v[n-1]

template <class Comparable>
void selectionSort(vector<Comparable> &v )
{
    typename vector<Comparable>::iterator it;
    for(it = v.begin(); it != v.end()-1; ++it )
        iter_swap(it, min_element(it, v.end()));
}
```

Existente em STL:

- template <class [ForwardIterator](#)> ForwardIterator
ForwardIterator **min_element**(ForwardIterator first, ForwardIterator last);
- template <class [ForwardIterator1](#), class [ForwardIterator2](#)>
inline void **iter_swap**(ForwardIterator1 a, ForwardIterator2 b);



Bubble Sort

- Algoritmo de Ordenação “BubbleSort” (“Exchange Sort”):
 - Compara elementos adjacentes. Se o segundo for menor que o primeiro, troca-os.
 - Repetir para todos os elementos excepto o último (que já está correcto)
 - Repetir, usando menos um par em cada iteração até não haver mais pares (ou não haver trocas)
- 2 ciclos encaixados, cada um pode ter N iterações:
 - **Complexidade $O(N^2)$**



Bubble Sort

Algoritmo em C++

```
// Ordena vetor v de n elementos inteiros, ficando v[0] ≤ ... ≤ v[n-1]

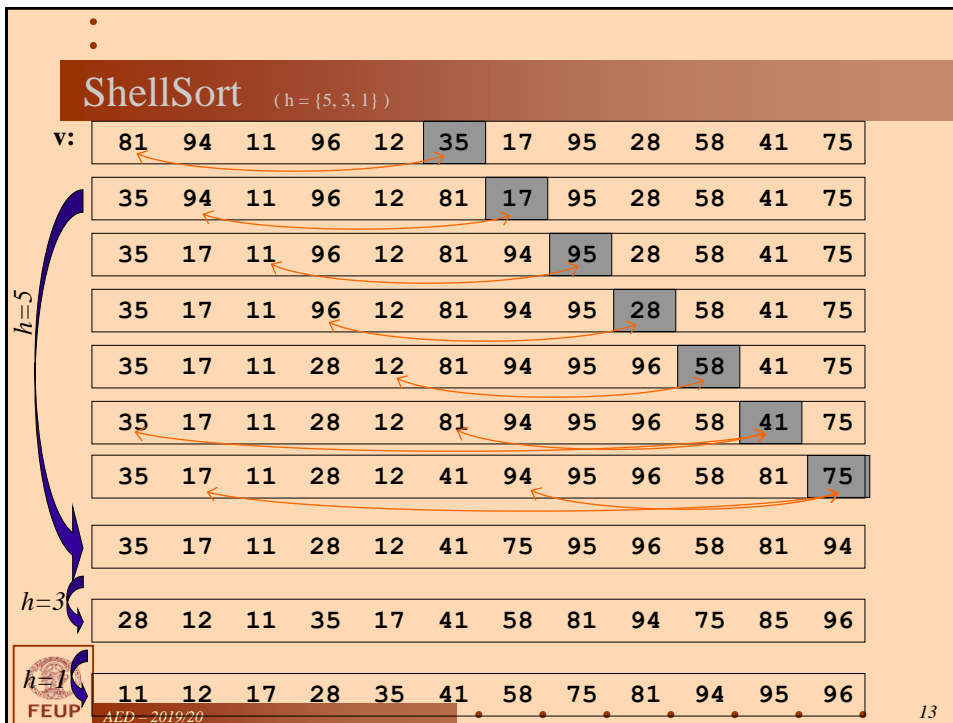
template <class Comparable>
void bubbleSort(vector<Comparable> &v)
{
    for(unsigned int j=v.size()-1; j>0; j--)
    {
        bool troca=false;
        for(unsigned int i = 0; i<j; i++)
            if(v[i+1] < v[i]) {
                swap(v[i],v[i+1]);
                troca = true;
            }
        if (!troca) return;
    }
}
```



ShellSort

- Compara elementos distantes
- Distância entre elementos comparados vai diminuindo, até que a comparação seja sobre elementos adjacentes
 - Usa a sequência h_1, h_2, \dots, h_t ($h_1=1$)
 - Em determinado passo, usando incremento h_k , todos os elementos separados da distância h_k estão ordenados, $v[i] \leq v[i+h_k]$
- Sequência de incrementos:
 - Shell: mais popular, não mais eficiente $O(N^2)$
 - $h_t = N/2, \quad h_k = h_{k+1}/2$
 - Hibbard: incrementos consecutivos não têm fatores comuns
 - $h = 1, 3, 7, \dots, 2^k-1$ $O(N^{5/4})$





13

Implementação de ShellSort

```
// ShellSort - com incrementos de Shell
template <class Comparable>
void shellSort(vector<Comparable> &v)
{
    int j;
    for (int gap = v.size()/2; gap > 0; gap /= 2)
        for (unsigned int i = gap; i < v.size(); i++)
        {
            Comparable tmp = v[i];
            for (j = i; j >= gap && tmp < v[j-gap]; j -= gap)
                v[j] = v[j-gap];
            v[j] = tmp;
        }
}
```

FEUP AED - 2019/20

14

MergeSort

- Abordagem recursiva
 - Divide-se o vetor ao meio
 - Ordena-se cada metade (usando MergeSort recursivamente)
 - Fundem-se as duas metades já ordenadas
- **Divisão e conquista**
 - problema é dividido em dois de metade do tamanho
- Análise
 - Tempo execução: $O(N \log N)$
 - 2 chamadas recursivas de tamanho $N/2$
 - Operação de junção de vetores: $O(N)$
- Inconveniente
 - fusão de vetores requer espaço extra linear



Implementação de MergeSort em C++

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

// internal method that makes recursive calls
// v is an array of Comparable terms
// tmpArray is an array to place the merged result
// left (right) is the left(right)-most index of the subarray
template <class Comparable>
void mergeSort(vector <Comparable> & v,
               vector<Comparable> & tmpArray, int left, int right)
{
    if (left < right)
    {
        int center = (left + right) / 2;
        mergeSort(v, tmpArray, left, center);
        mergeSort(v, tmpArray, center + 1, right);
        merge(v, tmpArray, left, center + 1, right);
    }
}
```



Implementação de MergeSort em C++

```
// internal method that makes recursive calls. v is an array of
// Comparable terms. tmpArray is an array to place the merged result.
// left (right) is the left(right)-most index of the subarray
template <class Comparable>
void merge(vector <Comparable> & v, vector<Comparable> &tmpArray,
          int leftPos, int rightPos, int rightEnd)
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while ( leftPos <= leftEnd && rightPos <= rightEnd )
        if ( v[leftPos] <= v[rightPos] )
            tmpArray[tmpPos++] = v[leftPos++];
        else
            tmpArray[tmpPos++] = v[rightPos++];
    while ( leftPos <= leftEnd )
        tmpArray[tmpPos++] = v[leftPos++];
    while ( rightPos <= rightEnd )
        tmpArray[tmpPos++] = v[rightPos++];
    for ( int i = 0; i < numElements; i++, rightEnd-- )
        v[rightEnd] = tmpArray[rightEnd];
}
```

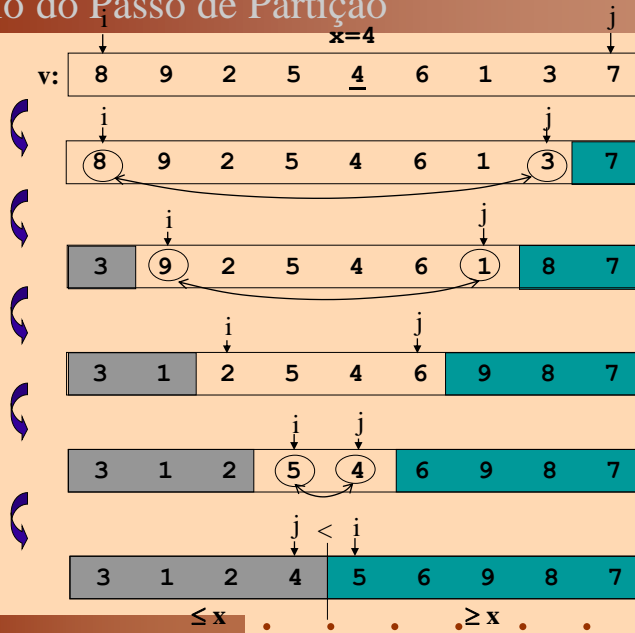


Ordenação por Partição (Quick Sort)

- Algoritmo (ordenação por partição):
 1. Caso básico: Se o número (n) de elementos do vetor (v) a ordenar for muito pequeno, usar outro algoritmo (insertionSort)
 2. Passo de partição:
 - 2.1. Escolher um elemento “arbitrário” (x) do vector (chamado pivot)
 - 2.2. Partir o vetor inicial em dois sub-vetores (esquerdo e direito), com valores $\leq x$ no sub-vetor esquerdo e valores $\geq x$ no sub-vetor direito
 3. Passo recursivo: Ordenar os sub-vetores esquerdo e direito, usando o mesmo método recursivamente
- Algoritmo recursivo baseado na técnica *divisão e conquista*



Exemplo do Passo de Partição



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
 - $O(N \log N)$
 - *caso médio*: pivot corta vetor arbitrariamente
 - $O(N \log N)$
- Escolha do pivot
 - um dos elementos extremos do vetor:
 - má escolha: $O(N^2)$ se vetor ordenado
 - elemento aleatório:
 - envolve uso de mais uma função pesada
 - mediana de três elementos (extremos do vetor e ponto médio)
 - recomendado



Implementação da Ordenação por Partição

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

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



Implementação da Ordenação por Partição

```
template <class Comparable>  
void quickSort(vector<Comparable> &v, int left, int right)  
{  
    if (right - left <= 10) // se vetor 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);  
    }  
}
```



Implementação da Ordenação por Partição

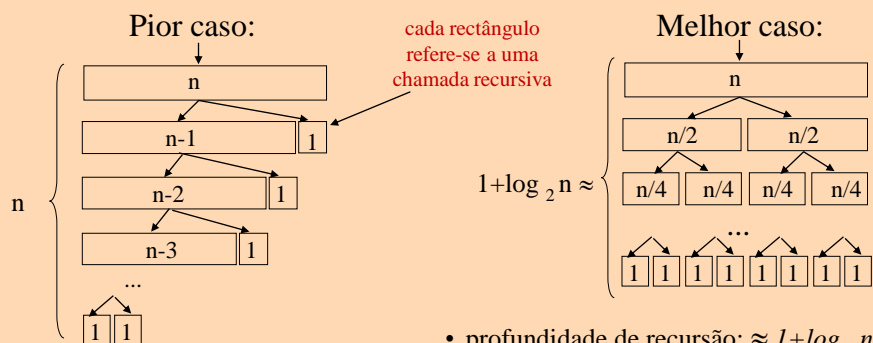
```
/* determina o valor do pivot como sendo a mediana de 3 valores: elementos
extremos e central do vetor */

template <class Comparable>
const 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];
}
```



Eficiência da Ordenação por Partição



- profundidade de recursão: n
- tempo de execução total (somando totais de linhas):

$$T(n) = O[n + n + (n-1) + \dots + 2]$$

$$= O[n + (n-1)(n+2)/2] = O(n^2)$$

- profundidade de recursão: $\approx 1 + \log_2 n$ (sem contar com a possibilidade de um elemento ser excluído dos sub-vetores esquerdo e direito)
- tempo de execução total (uma vez que a soma de cada linha é n):

$$T(n) = O[(1 + \log_2 n) n] = O(n \log n)$$



Complexidade Espacial de QuickSort

- O espaço de memória exigido por cada chamada de **quickSort**, sem contar com chamadas recursivas, é independente do tamanho (n) do vetor
- O espaço de memória total exigido pela chamada de **quickSort**, incluindo as chamadas recursivas, é pois proporcional à profundidade de recursão
- Assim, a complexidade espacial de **quickSort** é:
 - $O(\log n)$ no melhor caso (e no caso médio)
 - $O(n)$ no pior caso
- Em contrapartida, a complexidade espacial de **insertionSort** é $O(1)$

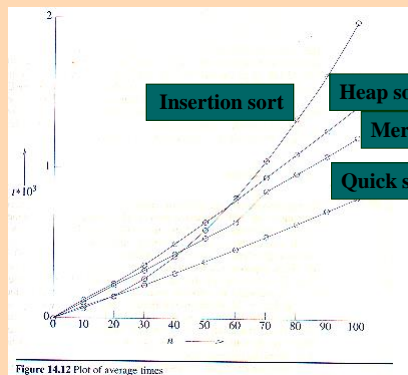


BucketSort

- Ordenação linear
 - usa informação adicional sobre entrada
- Algoritmo
 - vetor de entrada: inteiros positivos inferiores a M
 $a = [A_1, A_2, \dots, A_N]$; $A_i < M$
 - inicializar um vetor de M posições a **0's**
 $count = [c_1, c_2, \dots, c_M]$; $c_j = 0$
 - Ler vetor entrada (a) e para cada valor incrementar a posição respetiva no vetor $count$: $count[A_i]++$
 - Produzir saída lendo o vetor $count$
- Eficiência
 - tempo linear



Algoritmos de Ordenação



Cada ponto corresponde à ordenação de 100 vetores 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 vetores pequenos (até cerca 20 elementos), em que o método de ordenação por inserção (*insertionSort*) é melhor!



Algoritmos da STL

- Ordenação de vetores:

`void sort(iterator start, iterator end);`

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

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

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

- Algoritmo de ordenação implementado em `sort()` é o algoritmo *introsort*, possui complexidade $O(N \log N)$



Algoritmo *sort* 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;
};

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; }
```



Algoritmo *sort* da STL (exemplo)

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

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



Algoritmo *sort* 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();
}
```



Algoritmo *sort* da STL (exemplo)

```
int main()
{
    vector<Pessoa> vp;
    vp.push_back(Pessoa("66666666", "Rui Silva", 34));
    vp.push_back(Pessoa("77777777", "Antonio Matos", 24));
    vp.push_back(Pessoa("1234567", "Maria Barros", 20));
    vp.push_back(Pessoa("7654321", "Carlos Sousa", 18));
    vp.push_back(Pessoa("33333333", "Fernando Cardoso", 33));
    vector<Pessoa> vp1=vp;
    vector<Pessoa> vp2=vp;

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

vector inicial:

v[0] = (BI: 66666666, nome: Rui Silva, idade: 34)
v[1] = (BI: 77777777, 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: 33333333, nome: Fernando Cardoso, idade: 33)



Algoritmo *sort* 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)



AED – 2019/20