4. Vectores: Algoritmos de Ordenação

João Pascoal Faria (versão original)

Ana Paula Rocha (versão 2004/2005)

Luís Paulo Reis (versão 2005/2006)

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 (“sorting”) com complexidade **O(N^2) -** por exemplo Ordenação por Inserção, BubbleSort ou Shellsort que são muito simples
  - Existem algoritmos de ordenação mais difíceis de codificar que têm complexidade **O(N log N)**

Algoritmos de Ordenação de Vectores

- Algoritmos:
  - Ordenação por Inserção
  - Ordenação por Selecção
  - BubbleSort
  - ShellSort
  - MergeSort
  - Ordenação por Partição (QuickSort)
  - BucketSort

3

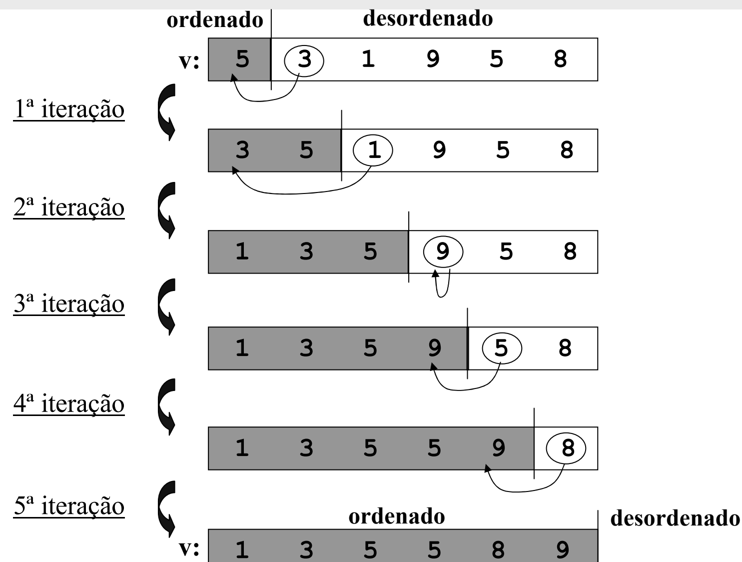
Ordenação por Inserção

- **N-1** passos
- Em cada passo *p* coloca 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

4

Exemplo de Ordenação por Inserção



5

Inserção de valor em vector ordenado

- Sub-problema: inserir um valor num vector ordenado (mantendo-o ordenado)
- Solução em C++ (função InsertSorted) :

```
/* Insere um valor x num array vec com n elementos ordenados.
Após a inserção, o array fica com n+1 elementos ordenados.
Retorna a posição em que inseriu o valor (entre 0 e n). */
template <class T>
int InsertSorted(T vec[], int n, T x)
{
   int j;
   // procura posição (j) de inserção da direita (posição n)
   // para a esquerda (posição 0), e ao mesmo tempo chega
   // elementos à direita (podia usar o próprio n em vez de j)
   for (j = n; j > 0 && x < vec[j-1]; j--)
      vec[j] = vec[j-1];
   vec[j] = x; // insere agora
   return j; // retorna a posição em que inseriu
}
```

6

Implementação da Ordenação por Inserção em C++

- A função em C++ para ordenar um array pelo método de inserção é agora trivial

```
// Ordena array vec de n elementos: vec[0]≤... ≤vec[n-1]
template <class T> void InsertionSort(T vec[], int n)
{
    // i - tamanho do sub-array esquerdo ordenado
    for (int i = 1; i < n; i++)
        InsertSorted(vec, i , vec[i]);
}
```

- Juntando tudo (para ser mais eficiente) ...

```
template <class T> void InsertionSort(T vec[], int n)
{
   for (int i = 1; i < n; i++) {
      T x = vec[i];
      for (int j = i; j > 0 && x < vec[j-1]; j--)
         vec[j] = vec[j-1];
      vec[j] = x;
   }
}
```


Ordenação por Inserção usando `vector`

```
// Ordena elementos do vector vec. Comparable: deve conter
// construtor cópia, operadores igualdade (=,<,>)

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

Análise da Ordenação por Inserção

- 2 ciclos encaixados, cada um pode ter N iterações :
  - **$O(N^2)$**
- Caso mais desfavorável: vector em ordem inversa
  - **$O(N^2)$**
- Caso mais favorável: vector já ordenado
  - **$O(N)$**
- **Conclusão:**
  - **Só pode ser utilizado para vectores pequenos…**

    (ver slides sobre complexidade de algoritmos)

9

Ordenação por Selecção

- Provavelmente é o algoritmo mais intuitivo:
  - Encontrar o mínimo do vector
  - Trocar com o primeiro elemento
  - Continuar para o resto do vector (excluindo o primeiro)
- 2 ciclos encaixados, cada um pode ter N iterações :
  - **Complexidade $O(N^2)$**
- Variantes:
  - "Stable Sort" – Insere mínimo na primeira posição (em vez de realizar a troca)
  - "Shaker Sort" – Procura máximo e mínimo em cada iteração (Selecção bidireccional)

10

Ordenação por Selecção

Algoritmo em C:

*// Ordena array **vec** de **n** elementos inteiros, ficando vec[0]≤.. ≤vec[n-1]*
// usando ordenação por selecção

```
void selectionSort(int vec[], int tam)
{
 int i, j, min, aux;
 for (i=0; i<tam-1; i++) {
   min = i;
   for (j=i+1; j<tam; j++) {
     if (vec[j] < vec[min]) min = j;
   }
   aux = vec[i];
   vec[i] = vec[min];
   vec[min] = aux;
 }
}
```


Ordenação por Selecção

Algoritmo em C++:

*// Ordena array **vec** de **n** elementos, ficando vec[0]≤.. ≤vec[n-1]*
// usando ordenação por selecção

```
template<class T>
void selection_sort(vector<T> &vec )
{
   vector<T>::iterator it1;
  for(it1 = vec.begin(); it1 != vec.end()-1; ++it1 ){
       iter_swap(it1, min_element(it1, vec.end()));
  }
}
```

BubbleSort

- Algoritmo de Ordenação "BubbleSort" ("Exchange Sort"):
  - Compara elementos adjacentes. Se o segundo for menor do que o primeiro, troca-os
  - Fazer isto desde o primeiro até ao último par
  - 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)
- Diversas variantes
- 2 ciclos encaixados, cada um pode ter N iterações:
  - **Complexidade O(N^2)**

13

BubbleSort

Algoritmo em C:

*// Ordena array **vec** de **n** elementos inteiros, ficando vec[0]≤.. ≤vec[n-1]*
// usando BubbleSort

```
void bubble(int vec[], int n)
  {
    int troca, i, j, aux;
    for (i = n-1; i > 0; i--) {
              // o maior valor entre vec[0] e vec[i] vai para a posição vec[i]
      troca = 0;
      for (j =0 ; j < i; j++)  {
        if (vec[j] > vec[j+1]) {
          aux = vec[j]; vec[j] = vec[j+1]; vec[j+1] = aux;
          troca = 1;
        }
      }
      if (!troca) return;
    }
  }
```

14

BubbleSort

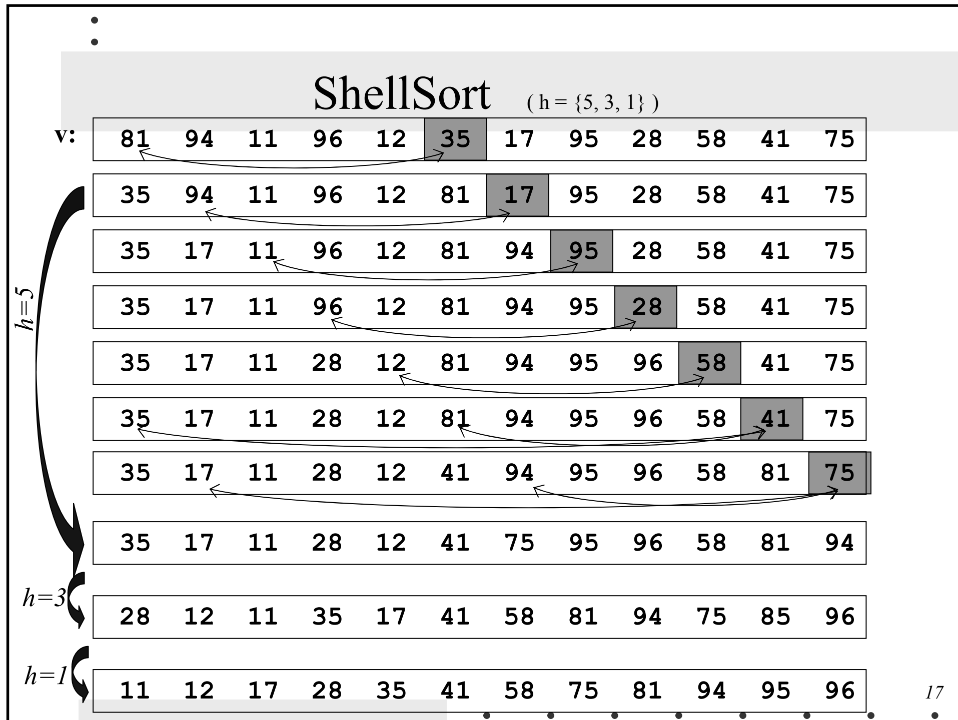
Algoritmo em C++:

*// Ordena array **vec** de **n** elementos inteiros, ficando vec[0]≤.. ≤vec[n-1]*
// usando BubbleSort

```
template<class T>
void bubble_sort(vector<T> &vec )
{
  vector<T>::reverse_iterator it1;
  for(it1 = vec.rbegin(); it1 != vec.rend(); ++it1 ) {
    vector<T>::iterator it2;
    bool troca = false;
    for(it2 = vec.begin(); &*it2 != &*it1; ++it2 )  {
      if (*it2 > *(it2+1)) {
        iter_swap(it2, it2+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 $h_1, h_2, \ldots, h_t$ ($h_1=1$)
  - Em determinado passo, usando incremento h_k, todos os elementos separados da distância h_k estão ordenados, $vec[i] \leq vec[i+h_k]$
- Sequência de incrementos:
  - Shell: mais popular, não mais eficiente $O(N^2)$
    - $h_t = N/2$, $h_k = h_{k+1}/2$
  - Hibbard: incrementos consecutivos não têm factores comuns
    - $h = 1, 3, 7, \ldots, 2^k-1$ $O(N^{5/4})$



ShellSort

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

18

ShellSort

```
// ShellSort genérico. Realiza Ordenação por Inserção nos elementos
   de vec[] com uma dada distância - gap.
// Se gap=1 faz Ordenação por inserção normal
void shellSortPhase(int vec[], int tam, int gap) {
   for (int i = gap; i < tam; ++i) {
        int value = vec[i];
        for (int j = i - gap; j >= 0 && vec[j] > value; j -= gap)
               vec[j + gap] = vec[j];
       vec[j + gap] = value;
   }
}

void shellSort(int vec[], size_t length) {
// gaps[ ] deve ser aproximadamente uma progressão geométrica. A
   sequência apresentada é a melhor
   static const int gaps[] = {1, 4, 10, 23, 57, 132, 301, 701};
   for (int sInd = sizeof(gaps)/sizeof(gaps[0])-1; sInd>=0; --sInd)
        shellSortPhase(vec, length, gaps[sInd]);
}
```


MergeSort

- Abordagem recursiva
  - Divide-se o vector 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 logN)
    - 2 chamadas recursivas de tamanho N/2
    - Operação de junção de vectores: O(N)
- Inconveniente
  - fusão de vectores requer espaço extra linear

MergeSort

```
//Algoritmo em C++:
template <class Comparable>
void mergeSort(vector <Comparable> &vec)
{
    vector<Comparable> tmpVec(vec.size());
    mergeSort(vec, tmpVec, 0, vec.size()-1);
}

/* Método que realiza chamadas recursivas:
   vec é um vector de elementos do tipo Comparable.
   tmpVec é um vector para colocar o resultado da fusão (merge)
   left (right) é o elemento mais à esquerda(direita) do sub vector */
template <class Comparable>
void mergeSort(vector <Comparable> &vec,
        vector<Comparable> &tmpVec, int left, int right)
{
    if (left < right) {
       int center = (left + right)/2;   //divide ao meio
       mergeSort(vec, tmpVec, left, center);  // ordena 1ª metade
       mergeSort(vec, tmpVec, center + 1, right); // ordena 2ª metade
       merge(vec, tmpVec, left, center +1, right); // Junta metades
    }
}
```


MergeSort

```
// Algoritmo de fusão dos dois vectores do MergeSort
template <class Comparable>
void merge(vector <Comparable> &vec, vector<Comparable> &tmpVec,
           int leftPos, int rightPos, int rightEnd)
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while (leftPos <= leftEnd && rightPos <= rightEnd )
       if (vec[leftPos] <= vec[rightPos] )
          tmpVec[tmpPos++] = vec[leftPos++];
       else
          tmpVec[tmpPos++] = vec[rightPos++];
    while (leftPos<=leftEnd) tmpVec[tmpPos++] = vec[leftPos++];
    while (rightPos<=rightEnd) tmpVec[tmpPos++] = vec[rightPos++];
    for ( int i = 0; i < num Elements; i++, rightEnd-- )
       vec[rightEnd] = tmpVec[rightEnd];
}
```

Ordenação por Partição (Quick Sort)

- 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 (podendo existir um 3º sub-vector central com valores =**x**)
  3. Passo recursivo: Ordenar os sub-vectores esquerdo e direito, usando o mesmo método recursivamente
- Algoritmo recursivo baseado na técnica *divisão e conquista*

Refinamento do Passo de Partição

(Supõe-se excluído pelo menos o caso em que n = 0)

2. Passo de partição:

2.1. Escolher para pivot (**x**) o elemento do meio do vector (*vec[n/2]*)

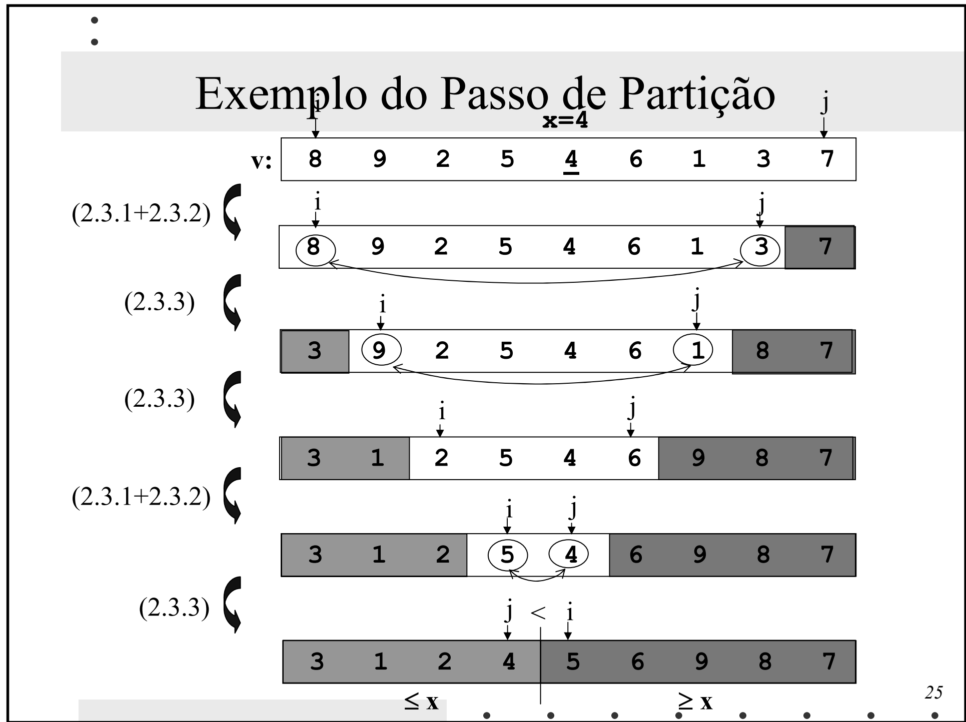
2.2. Inicializar $i = 0$ (índice da 1º posição do vector)
Inicializar $j = n-1$ (índice da última posição do vector)

2.3. Enquanto $i \leq j$, fazer:

- 2.3.1. Enquanto $vec[i] < x$ (é sempre $i \leq n-1$!), incrementar i
- 2.3.2. Enquanto $vec[j] > x$ (é sempre $j \geq 0$!), decrementar j
- 2.3.3. Se $i \leq j$ *então* trocar *vec[i]* com *vec[j]*, incrementar i e decrementar j

2.4. O sub-vector esquerdo (com valores $\leq x$) é *vec[0], ..., vec[j]* (vazio se $j<0$)

2.5. O sub-vector direito (com valores $\geq x$) é *vec[i], ..., vec[n-1]* (vazio se $i>n-1$)



Passo de Partição

- Da 1ª vez, a condição em 2.3. (i ≤ j) é verdadeira, e acontece o seguinte:
  - o ciclo 2.3.1. pára com i na posição do pivot ou à esquerda
  - o ciclo 2.3.2. pára com j na posição do pivot ou à direita
  - assim, antes do passo 2.3.3., 0≤ i ≤ j ≤ n-1
  - ilustram-se a seguir os vários situações possíveis antes do passo 2.3.3.:

tamanho de cada segmento (1 por omissão) →

0+		1+		0+
<x	≥x		≤x	>x
	i		j	

0+			0+
<x	≥x	≤x	>x
	i	j	

0+		0+
<x	x	>x
	i j	

  - uma vez que i≤ j, a troca é realizada, resultando respectivamente as seguintes situações após o passo 2.3.3.:

1+	1+	1+
≤x		≥x
	i j	

(i ≤ j)
CONTINUA!

1+	1+
≤x	≥x
j	i

(i = j+1 > j)
FIM!

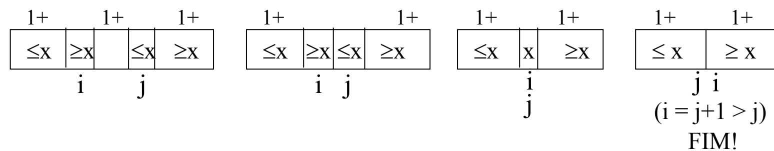
0+		0+
<x	x	>x
j		i

(i = j+2 > j)
FIM!

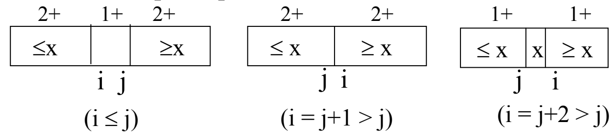
26

Passo de Partição

- No caso da esquerda, o ciclo 2.3. é executado 2ª vez, e acontece o seguinte:
  - o ciclo 2.3.1. pára de certeza, devido à existência de valores ≥ x à direita
  - o ciclo 2.3.2. pára de certeza, devido à existência de valores ≤ x à esquerda
  - antes do passo 2.3.3. verifica-se uma das situações seguintes:



  - nos três casos da esquerda, a troca é realizada, resultando respectivamente as seguintes situações após o passo 2.3.3:



Passo de Partição (conc.) e Passo de Recursão

- No caso da esquerda, o ciclo 2.2.3. é executado 3ª vez, e o esquema repete-se.
- Conclui-se (por indução) que realmente nunca são acedidos elementos fora do vector durante o ciclo 2.3.

- No final do passo de partição, qualquer dos sub-vectores (esquerdo e direito) é de tamanho <n
  - além disso, os sub-vectores são disjuntos, como convém
- Assim, as chamadas recursivas do mesmo algoritmo destinam-se a ordenar vectores cada vez mais pequenos e, portanto, cada vez mais próximos do caso básico. Isto garante que a recursão tem fim.

Implementação da Ordenação por Partição em C++

```
/* Ordena array(vec) entre 2 posições (a e b). Supõe que os elementos
do array são comparáveis com "<" e ">" e copiáveis com "=". */
template <class T> void QuickSort(T vec[], int a, int b)
{
   if (a >= b) return;  // caso básico (tamanho <= 1)
T x = vec[(a+b)/2];  //T x2 = vec[a]; T x3 = vec[b]; // inicializações
If (x1>x2 && x1<x3) x=x1; else If (x2>x1 && x2<x3) x=x1;
   int i = a, j = b;
   // passo de partição
   do {
       while (vec[i] < x) i++;
       while (vec[j] > x) j--;
       if (i > j) break;
       T tmp=vec[i]; vec[i]=vec[j]; vec[j]=tmp;
       i++; j--; //troca
   } while (i <= j);
   // passo recursivo
   QuickSort(vec, a, j);
   QuickSort(vec, i, b);
}
```

29

Programa de teste

```
#include <iostream.h>
#include <stdlib.h>  // Para usar rand()
// inserir aqui a função QuickSort implementada

void imprime(const int vec[], int n)
{
    for (int i = 0; i < n; i++)
        cout << vec[i] << ' ';
    cout << '\n';
}
main()
{
    const int SIZE = 10000;
    int vec[SIZE];
    for (int i = 0; i < SIZE; i++)
         vec[i] = rand();
    imprime(vec, SIZE);
    QuickSort(vec, 0, SIZE - 1);
    imprime(vec, SIZE);
    return 0;
}
```

30

Análise da Ordenação por Partição

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

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

31

BucketSort (BinSort)

- Ordenação linear
  - usa informação adicional sobre entrada
- Algoritmo
  - vector de entrada: inteiros positivos inferiores a **M**
    $Vec = [\ V_1,\ V_2, \ldots,\ V_N\ ] \quad ;\ V_i < M$
  - inicializar um vector de **M** posições a **0's**
    $count = [c_1, c_2, \ldots, c_M] \quad ; c_j = 0$
  - Ler vector entrada (*Vec*) e para cada valor incrementar a posição respectiva no vector *count* : *count[Vi]++*
  - Produzir saída lendo o vector *count*
- Eficiência
  - tempo linear

32