Insertion Sort

Analise de Algoritmos

Integrantes:

Amanda Garcia, Felipe Arruda, Victor Magalhães

Motivação - Descrição Informal

Algoritmos de ordenação são utilizados nas situações onde temos um conjunto de elementos, geralmente armazenados em um vetor, e queremos ordená-los de forma crescente.

Um dos métodos mais conhecidos é o Insertion Sort. A principal característica deste método consiste em ordenar um conjunto de elementos, utilizando um subconjunto ordenado localizado em seu inicio, e em cada interação, acrescentamos a este subconjunto mais um elemento, até que atingimos o último elemento do conjunto fazendo com que todo o vetor se torne ordenado.

Motivação - Definição Formal do Problema

Entrada:

Um vetor de n números desordenados. Ex:(a1, a2, a3, …, an)

Questão:

Ordenar os elementos do vetor em ordem crescente.

Saída:

Um vetor com os mesmos n elementos da entrada ordenados de forma crescente. Ex: (a1 <= a2 <= a3, … <= an)

Motivação - Exemplos de Instâncias do Problema

Exemplo 1:

Entrada: vetor = {2, 4, 5, 1}

Saída: vetor = {1, 2, 3, 4}

Motivação - Exemplos de Instâncias do Problema

Exemplo 2:

Entrada: vetor = {5, 3, 7, 8, 2, 5}

Saída: vetor = {2, 3, 5, 5, 7, 8}

Algoritmo - Ideia

Este algoritmo foca em ordenar os elementos de um vetor, percorrendo o vetor da **direita** para a **esquerda**, e enquanto está executando, este coloca os elementos que estão mais a esquerda ordenados.

Algoritmo - Ideia

Em cada etapa temos duas regiões diferentes no vetor:

- A primeira, que está **ordenada**;
- A segunda, que **não está ordenada**;

E em cada passo o algoritmo vai **percorrendo** o vetor, colocando um dos elementos da região onde não estão ordenados em sua respectiva posição na região dos elementos ordenados.

No final da execução do algoritmo a primeira região equivale a todo o vetor, e a segunda é vazia.

Algoritmo - Exemplo

Vetor: { 2, 4, 3, 1 }

Algoritmo - Exemplo

Inicialmente considera-se o primeiro elemento do arranjo como se ele estivesse ordenado.
Ele será considerado o sub-arranjo ordenado inicial.

{ 2, 4, 3, 1 }

Algoritmo - Exemplo

Agora o elemento imediatamente superior ao o sub-arranjo ordenado, no o exemplo o número **4**, deve se copiado para uma variável auxiliar qualquer.
Após copiá-lo, devemos percorrer o sub-arranjo a partir do último elemento para o primeiro. Assim poderemos encontrar a posição correta da nossa variável auxiliar dentro do sub-arranjo.

{ 2, 4, 3, 1 }

Algoritmo - Exemplo

A variável auxiliar (**4**) é maior que o último elemento do o sub-arranjo ordenado por isso o número **2** e o número **4** se mantém nas mesmas posições.

{ 2, 4, 3, 1 }

Algoritmo - Exemplo

O sub-arranjo ordenado possui agora dois elementos (**2** e **4**). Vamos repetir o processo anterior para que se continue a ordenação.
Copiamos então mais uma vez o elemento imediatamente superior ao o sub-arranjo ordenado para uma variável auxiliar (**3**).

{ 2, 4, <u>3</u>, 1 }

Algoritmo - Exemplo

Logo em seguida vamos comparando nossa variável auxiliar (**3**) com os elementos do sub-arranjo, sempre a partir do último elemento para o primeiro .
Neste caso verificamos que a nossa variável auxiliar é menor que o último elemento do sub-arranjo. Assim, copiamos este elemento para a direita e continuamos com nossas comparações.

{ 2, 3, 4, 1 }

Algoritmo - Exemplo

A nossa variável auxiliar é maior que o elemento do sub-arranjo que estamos comparando. Por isso ele se mantém na sua posição.

{ 2, 3, 4, 1 }

Algoritmo - Exemplo

O sub-arranjo ordenado possui agora três elementos (**2, 3** e **4**). Copiamos então mais uma vez o elemento imediatamente superior ao o sub-arranjo ordenado para uma variável auxiliar (**1**).

{ 2, 3, 4, 1}

Algoritmo - Exemplo

Neste caso verificamos que a nossa variável auxiliar é menor que o último elemento do sub-arranjo. Assim, copiamos este elemento para a direita e continuamos com nossas comparações.

{ 2, 3, 1,4}

Algoritmo - Exemplo

Mais uma vez a nossa variável auxiliar é menor que o elemento do sub-arranjo que estamos comparando.
Por isso ele deve ser copiado para a direita, abrindo espaço para que a variável auxiliar seja colocada em sua posição correta.

{ 2, 1 , 3, 4}

Algoritmo - Exemplo

Novamente a nossa variável é menor que o ultimo elemento do sub-arranjo, e copiamos ele para a direita.

{1, 2 ,3 ,4}

Algoritmo - Exemplo

Agora o vetor está ordenado :

{1, 2 ,3 ,4}

Algoritmo - Descrição Formal

```
PARA j ← 1 ATÉ COMPRIMENTO(A)
    chave ← A[j]
    i ← j - 1
    ENQUANTO i > 0 E A[i] > chave
        A[i + 1] ← A[i]
        i ← i - 1
    A[i + 1] ← chave
```

Algoritmo - Descrição Formal

A cada iteração, percorremos a entrada selecionando um par de chaves de índice i e i+1 e verificando se i+1 é maior que i, caso verdadeiro passamos para o próximo índice assumindo que a primeira parte está ordenada e fazemos o mesmo teste. Caso i+1 for menor que i, trocamos eles de posição e voltamos para onde não está ordenado para reordenar.

Algoritmo - Detalhes

O algoritmo é bem simples em sua estrutura.

Um vetor:

N elementos a serem ordenados

Poucas Variáveis Auxiliares:

Usadas no controle dos índices do vetor

Dois loops:

um para **percorrer** todo o vetor e outro usado para **comparar** cada elemento da iteração atual do primeiro loop com os outros elementos do vetor

Algoritmo - Implementação

```c
void InserctionSort(int n, int vetor[]){
  int j,i, chave;
  for(j = 1; j < n; j++){

   chave = vetor[j];
   i = j - 1;

   while(i >= 0 && vetor[i] > chave){

     vetor[i + 1] = vetor[i];
     i = i - 1;
   }

   vetor[i + 1] = chave;
 }
}
```

Análise - Prova de corretude

Subarranjo A [1 .. j - 1] : elementos já ordenados do vetor;
Subarranjo A [j + 1 .. n] : elementos que ainda faltam ordenar;

Loop Invariante: A[1 .. j - 1]

Análise - Prova de corretude

Inicialização

loop invariante é válido antes da primeira iteração do loop:

$j = 2$, então o subarranjo $A[1 .. j - 1]$ consiste apenas no único elemento $A[1]$.

Análise - Prova de corretude

Manutenção

Cada iteração mantém o loop invariante.

O corpo do loop exterior desloca-se A[j - 1], A[j - 2], A[j - 3]... A[j] onde o A[j] é inserido em sua posição adequada.

Análise - Prova de corretude

Término

O loop externo só termina quando $j = n + 1$

Subarranjo A [1 .. n] equivale aos elementos originalmente contidos em A [1 .. n], mas em sequência ordenada.

Subarranjo A[1.. n] é o arranjo inteiro.

Análise - Demonstração de Complexidade

O Insertion Sort tem para o pior caso a complexidade de tempo **O(n^2)**

Um exemplo do pior caso é quando o vetor a ser ordenado de forma crescente está todo ordenado de forma decrescente.

Ex: **{ 4, 3, 2 ,1}**.

Análise - Demonstração de Complexidade

Operação dominante:

Comparação (entre elementos para saber qual é o menor).

Ex: **A[i] > chave**

Análise - Demonstração de Complexidade

No primeiro loop temos **N - 1** iterações que sempre vão ocorrer.

```
PARA j ← 1 ATÉ COMPRIMENTO(A)
```

Análise - Demonstração de Complexidade

O segundo loop terá **N - 1** iterações no pior caso (se todas as vezes que a comparação “**A[i] > chave**” for verdadeira).

Isso é se o vetor estiver totalmente em ordem decrescente.

```
ENQUANTO i > 0 E A[i] > chave
```

Análise - Demonstração de Complexidade

Com isso temos, no pior caso, **(N - 1) * (N - 1)** operações de dominantes (comparação entre os elementos).

O que nós da uma complexidade assintótica de **$O(n^2)$ no pior caso**.

Conclusão

- Algoritmo simples;
- Eficiente em pequenas listas;
- Sensível a entradas distintas;
- Estável;

Conclusão - Vantagens

Ordena o vetor somente quando necessário.

Isto é, se o vetor já está em ordem, nenhum movimento substancial é realizado.

Conclusão - Desvantagens

- Se os elementos já estiverem em suas posições apropriadas, eles podem ser movidos dessas posições e retornarem mais tarde.

- Se um item está sendo inserido, todos os elementos maiores do que ele têm que ser movidos.

Conclusão - É Possível Fazer Melhor?

Sim, utilizando outros algoritmos de ordenação como o Heap Sort temos a seguinte comparação:

Algoritmo	**Pior Caso**	**Caso Medio**	**Melhor Caso**
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Referências Bibliográficas

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., STEIN, Clifford, Algoritmos: Teoria e Prática, Rio de Janeiro: Editora Campus, 2002, 936 p.

Frederico, A. R., Junior, S. J. (2012) “Analise Empírica de Algoritmos de Ordenação”. Disponível em: <http://pt.slideshare.net/OnOSJunior/anlise-emprica-de-algoritmos-de-ordenao>. Acesso em 23 fev. 2014.

Jaruzo, R. (2010) “Ordenação por inserção (Insertion Sort)”. Disponível em: <http://fortium.edu.br/blog/regis_jaruzo/files/2010/11/BLOG-INSERCTION-SORT.pdf>. Acesso em 23 fev. 2014.

Rosen, K. H., Matemática Discreta e suas Aplicações, McGraw-Hill, 2009, 963 p.

Valverde, I.P. (2010) “Algoritmo de Ordenação Insertion Sort”. Disponível em: <http://www.youtube.com/watch?v=bNcD4Icwo6Q>. Acesso em 23 fev. 2014.

Zoltán, K., László, T. (2011) “Insert-sort with Romanian folk dance”. Disponível em: <http://www.youtube.com/watch?v=ROalU379l3U>. Acesso em 23 fev. 2014.