Insertion Sort

Analise de Algoritmos

Integrantes:

Amanda Barbosa
Felipe Arruda
Victor Magalhães

Motivação

Definição do Problema

Algoritmos de ordenação são utilizados nas situações onde temos um conjunto de elementos, geralmente armazenados em um vetor, e queremos ordená-los em ordem crescente. Existem atualmente diversos métodos para realizar essa tarefa, cada um com suas vantagens e desvantagens. É importante saber qual algoritmo se encaixa melhor em um determinado caso.

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.

Definição Formal:

Entrada: Um vetor de n números desordenados. (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)

Exemplos de Instâncias do Problema

Exemplo 1:

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

Saída: vetor = {1, 2, 3, 4}

Exemplo 2:

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

Saída: vetor = {2, 3, 5, 5, 7, 8}

Insertion Sort

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.
O algoritmo de ordenação por inserção funciona como muitas pessoas ordenam as cartas em um jogo de sueca ou poker. A cada carta selecionada, ela é comparada com as outras cartas a esquerda e inserida no local apropriado.

Assim, em cada etápa 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. De forma que no final da execução do algoritmo a primeira região equivale a todo o vetor, e a segunda é vazia.

Exemplos

Este algoritmo considera o array como contendo uma parte ordenada (subarray da esquerda) e uma parte não ordenada (sub-array da direita).

Caso 1:

Para um vetor com os seguintes elementos: Vetor: { 2, 4, 3, 1 }

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 }

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 }

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 }

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, 3, 1 }

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 }

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 }

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 }
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}

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}

Novamente a nossa variável é menor que o ultimo elemento do sub-arranjo, e copiamos ele para a direita.
{1, 2 ,3 ,4}

Agora o vetor está ordenado : **{1, 2 ,3 ,4}**

Caso 2:

Para um vetor com os seguintes elementos: Vetor: { A, B, H, C}

Inicialmente considera-se o primeiro elemento do arranjo como se ele estivesse ordenado. Ele será considerado o sub-arranjo ordenado inicial.
{ A, B, H, C}

Agora o elemento imediatamente superior ao o sub-arranjo ordenado, 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.
{ A, B, H, C}

A variável auxiliar (B) é maior que o último elemento do o sub-arranjo ordenado por isso se mantém na mesma posição.
{ A, B, H, C}

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 (H).
{ A, B, H, C}

Logo em seguida vamos comparando nossa variável auxiliar (H) com os elementos do sub-arranjo, sempre a partir do último elemento para o primeiro . A variável auxiliar (B) é maior que o último elemento do o sub-arranjo ordenado por isso se mantém na mesma posição.
{ A, B, H, C}

Mais uma vez colocamos o elemento imediatamente superior ao o sub-arranjo ordenado em uma variável auxiliar (C). 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.
{ A, B, C, H}

A variável auxiliar (C) é maior que o último elemento do o sub-arranjo ordenado por isso se mantém na mesma posição.

Agora o vetor está ordenado : **{ A, B, C, H}**

Caso 3:

Entrada : {8 2 4 9 3 6 }

Primeira interação
- Sub-Conjunto inicial : {8}
- Elemento imediatamente superior ao sub-conjunto {2}
- O elemento 2 é menor que o elemento 8 por isso eles trocam de posição
- Final da primeira interação: 2 8 4 9 3 6

Segunda interação

- Sub-Conjunto : {2, 8}
- Elemento imediatamente superior ao sub-conjunto {4}
- O elemento 4 é menor que o elemento 8 por isso eles trocam de posição. O elemento 4 não é menor que o elemento 2 então ele se mantém na posição.
- Final da segunda interação: 2 4 8 9 3 6

Terceira interação
- Sub-Conjunto : {2, 4, 8}
- Elemento imediatamente superior ao sub-conjunto {9}
- O elemento 9 é maior que o elemento 8 por isso ele se mantém na posição
- Final da terceira interação: 2 4 8 9 3 6

Quarta interação
- Sub-Conjunto : {2, 4, 8, 9}
- Elemento imediatamente superior ao sub-conjunto {3}
- O elemento 3 é menor que os elementos 9,8 e 4 e maior que o elemento 2 por isso ele vai para a direita do 4.
- Final da quarta interação: 2 3 4 8 9 6

Quinta interação
- Sub-Conjunto : {2, 3, 4, 8, 9}
- Elemento imediatamente superior ao sub-conjunto {6}
- O elemento 6 é menor que os elementos 9,8 e maior que os elementp 2, 3 e 4 e por isso ele vai para a direita do 8.
- Final da quarta interação: 2 3 4 6 8 9

Descrição Formal

O algoritmo tem como objetivo organizar a entrada. São necessárias duas iterações, uma para percorrer tudo com tamanho n e uma que serve para percorrer e ir ordendando desde o ponto chave. A cada iteração de ordenação, percorremos a entrada desde o ponto chave selecionando um par de chaves de indice i e i + 1 e verificando se i + 1 é maior que i, caso verdadeiro passamos para o próximo indice assumindo que anterior dali todos estão ordenados e fazemos o mesmo teste. Caso i + 1 for menor que i, trocamos eles de posição e voltamos para o ponto chave, que é exatamente onde ainda não temos certeza se está ordenado e continuamos a percorrer até que toda a entrada esteja ordenada.

```
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
```

Detalhes

Este algoritmo é bem simples em termos de estruturas de dados, podendo ser feito utilizando apenas **um vetor** (dos N elementos a serem ordenados) e **poucas variáveis auxiliares** para serem usadas no controle dos índices do vetor, e pode ser feito com apenas **dois loops** (um para percorrer todo o vetor e outro usado para comparar cada elemento com os outros elementos do vetor).

Implementação

Segue abaixo um exemplo de implementação do algoritimo em C++.

```cpp
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 do Algoritmo

Aqui serão descritas analises mais elaboradas do algoritmo Insertion Sort, como sua prova de **corretude** e **demonstração de complexidade** de tempo para o **pior caso**.

Prova de Corretude

No início de cada iteração do loop externo, indexado por j, o subarranjo que consiste nos elementos A [1 .. j - 1], que são os elementos já ordenados do vetor, e os elementos A [j + 1 .. n], que são os elementos que ainda faltam ordenar. Denominamos essa parte A[1 .. j - 1] como loop invariante. Utilizamos o loop invariante para provar a corretude do algoritmo. Partindo desse princípio utilizaremos 3 detalhes sobre o loop invariante:

Inicialização: Começamos provando que o loop invariante é válido antes da primeira iteração do loop, quando j = 2. Então o subarranjo A [1 .. j - 1] consiste apenas no único elemento A[1], que é de fato o elemento original em A[1]. Além disso esse subarranjo é ordenado, e isso mostra que o loop invariante é válido antes da primeira iteração do loop.

Manutenção: Em seguida examinamos a segunda propriedade: a demonstração de que cada iteração mantém o loop invariante. Informalmente, 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.

Término: Finalmente examinamos o que ocorre quando o loop termina. No caso da ordenação por inserção o loop externo só termina quando j = n + 1, substituindo j por n + 1 no enunciado do loop invariante, temos que o subarranjo A [1 .. n] consiste nos elementos originalmente contidos em A [1 .. n], mas em sequência ordenada. Contudo, o subarranjo A[1.. n] é o arranjo inteiro. Isso garante que o algoritmo é correto

Demonstração de Complexidade

Será demonstrado nesta seção que o algoritmo 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}.

Operação dominante: Comparação (entre elementos para saber qual é o menor).
Ex: **A[i] > chave**

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

```
PARA j ← 1 ATÉ COMPRIMENTO(A)
```

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
```

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

O método de ordenação por inserção é um algoritmo **simples** e **eficiente** quando aplicado em **pequenas listas**. O tempo gasto para executar o algoritmo do Insertion Sort depende do valor de entrada (é **sensível a entradas distintas**).

Ordenar milhares de números leva bem mais tempo do que ordenar três números. Além disso, o Insertion Sort pode levar diferentes quantidades de tempo para ordenar duas sequências de entrada de mesmo tamanho dependendo do quanto elas já estão ordenadas.

Uma das **vantagens** de se usar a ordenação por inserção é que ela ordena o vetor somente quando realmente necessário. Se o vetor já está em ordem, nenhum movimento substancial é realizado. Neste caso o algoritmo reconhece que parte do vetor já está ordenado e pára a execução.

Uma **desvantagem** é que não é explorado o fato de que elementos podem já estar em suas posições apropriadas e com isso eles podem ser movidos dessas posições e voltarem mais tarde. Outra desvantagem é que, se um item está sendo inserido,todos os elementos maiores do que ele têm que ser movidos.

O algoritmo de inserção é **estável**, isto é, os registros com chaves iguais sempre irão manter a **mesma posição relativa** de antes do início da ordenação.

Vale ressaltar que existem algoritmos (como o Heap Sort) que são melhores que o Insertion Sort, que em seu **pior caso** tem sua complexidade assintótica na ordem de **$O(n^2)$**.

A tabela a seguir mostra a diferença entre o Heap Sort e o Insertion Sort em diferentes casos:

Algoritmo	Pior Caso	Caso Médio	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=ROaIU379l3U>. Acesso em 23 fev. 2014.