Faculdade Professor Miguel Ângelo Silva Santos

Lucas Barbosa Guimarães - 2001230020

Atividade teórica - Algoritmos de ordenação e complexidade

Macaé - RJ

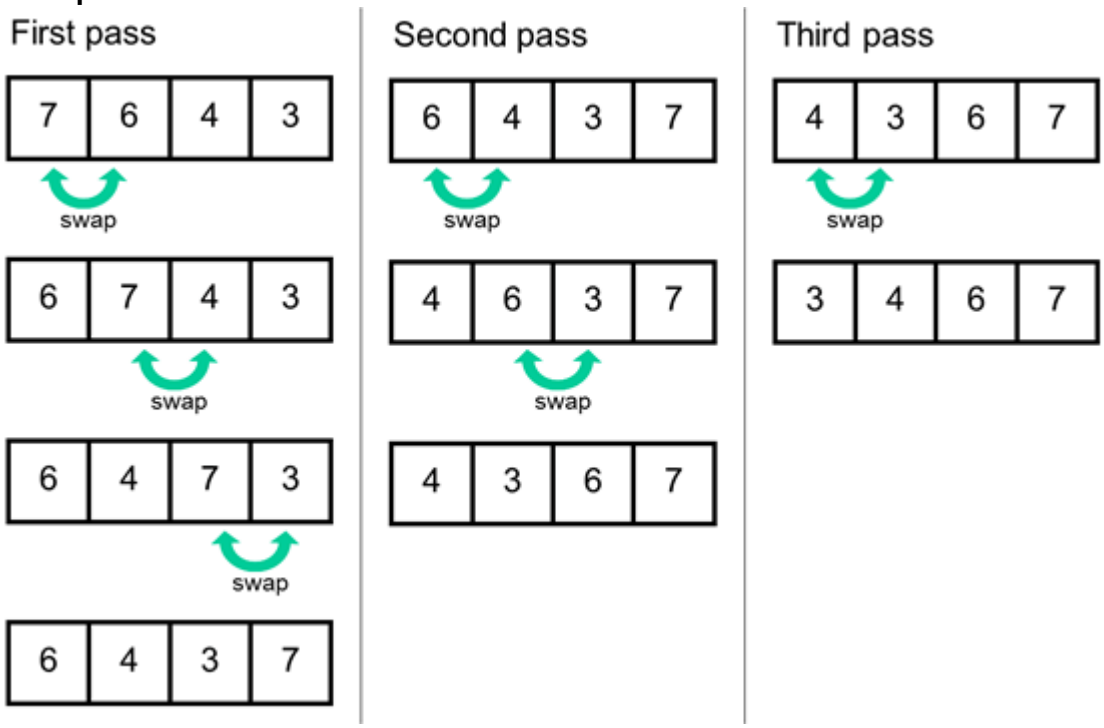
2022

- **Bubble Sort**

O **bubble sort** realiza múltiplas passagens por uma lista. Ele compara itens adjacentes e troca aqueles que estão fora de ordem. Cada passagem pela lista coloca o próximo maior valor na sua posição correta. Em essência, cada item se desloca como uma “bolha” para a posição à qual pertence.

```
35  void bubbleSort(int* vetor, int tamanho)
36  {
37      int k, j, aux;
38
39      for(k=0;k<tamanho;k++)
40      {
41          for(j=0;j<tamanho-1;j++)
42          {
43              if(vetor[j]>vetor[j+1])
44              {
45                  aux=vetor[j];
46                  vetor[j]=vetor[j+1];
47                  vetor[j+1]=aux;
48              }
49          }
50      }
51  }
```

Exemplo:



Complexidade do Bubble Sort:

O algoritmo de Bubble Sort utiliza dois loops, um interno e outro externo. Por conta disso, no **pior caso,** o loop externo percorre O(n) vezes.
Como resultado, a complexidade do Bubble Sort no pior caso é de O(n * n) = O(n^2).

- ## Insertion Sort

Insertion Sort, ou *ordenação por inserção*, é um algoritmo de ordenação que, dado uma estrutura (array, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

A ideia é percorrer um vetor de elementos da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda.

Exemplo: considerando o vetor: [12 11 13 5 6]

1º: Compara o elemento na posição 1 com a posição 0. Como 12 é maior que 11, eles são trocados.

Então, por enquanto, o 11 é armazenado em um subvetor.

Vetor: [11 12 13 5 6].

2º: Compara o elemento na posição 2 com a posição 1. Como 13 é maior que 12, eles não serão trocados. 12 será armazenado no sub-vetor junto com o 11.

Vetor: [11 12 13 5 6].

3º: Agora, compare o elemento na próxima posição. 5 é menor do que 13, então troca.

Ainda assim, o vetor não está ordenado, já que o 5 é menor do que 12, então troca novamente.

Por fim, 5 ainda é menor do que 11, troca outra vez. Após isso, o 5 também é armazenado no sub-vetor. No momento o sub-vetor armazena os seguintes números: [5 11 12].

Vetor: [5 11 12 13 6].

4º: Passando para os próximos elementos do vetor: 13 e 6.

Como eles não estão em ordem, troque-os de posição.

[5 11 12 6 13].

O vetor ainda não está ordenado, já que 12 é maior do que 6, então troca novamente.

[5 11 6 12 13].

Por fim, troque o 11 com o 6:

[5 6 11 12 13].

Por fim, o vetor está ordenado.

Algoritmo:

```
103  void insertionSort(int* vetor, int tamanho)
104  {
105      for(int i = 1; i < tamanho; i++)
106      {
107          int tmp = vetor[i];
108          int j = i;
109          while(j > 0 && tmp < vetor[j - 1])
110          {
111              vetor[j] = vetor[j - 1];
112              j--;
113          }
114          vetor[j] = tmp;
115      }
116  }
```

Complexidade do Insertion Sort:

Como visto no algoritmo, o Insertion Sort utiliza de duas estruturas de loop, o for (loop externo) e o while (loop interno), por conta disso a complexidade de pior caso do Insertion Sort é **O(n^2).**

- ## Selection Sort

A **ordenação por seleção** (do inglês, ***selection sort***) é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os n -1 elementos restantes, até os últimos dois elementos.

É composto por dois laços, um laço externo e outro interno. O laço externo serve para controlar o índice inicial e o interno percorre todo o vetor. Na primeira iteração do laço externo o índice começa de 0 e cada iteração ele soma uma unidade até o final do vetor e o laço mais interno percorre o vetor começando desse índice externo + 1 até o final do vetor.

Algoritmo:

```
68  //  Select Sort
69  void selectSort(int* vetor, int tam) {
70      for (int i=0;i<tam;++i) {
71          int iMenor=i;
72          for (int iNext = i+1; iNext < tam; ++iNext) {
73              if (vetor[iNext] < vetor[iMenor]) {
74                  iMenor = iNext;
75              }
76          }
77          int aux = vetor[i];
78          vetor[i] = vetor[iMenor];
79          vetor[iMenor] = aux;
80      }
81  }
```

Complexidade do Selection Sort:

O *selection sort* compara a cada interação um elemento com os outros, visando encontrar o menor. Dessa forma, podemos entender que não existe um melhor caso mesmo que o vetor esteja ordenado ou em ordem inversa serão executados os dois laços do algoritmo, o externo e o interno. A complexidade deste algoritmo será sempre **O(n^2).**

- ## Quick Sort

O quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o quick sort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. Os passos são:

1. Escolha um elemento da lista, denominado *pivô*;

2. Particiona: rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sub listas não ordenadas. Essa operação é denominada *partição*;
3. Recursivamente ordene a sub lista dos elementos menores e a sublista dos elementos maiores;

O caso base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

A escolha do pivô e os passos do particiona podem ser feitos de diferentes formas e a escolha de uma implementação específica afeta fortemente a performance do algoritmo.

Algoritmo:

```
165  //  Quick Sort
166  void quickSort(int* vetor, int tamanho)
167  {
168    if (tamanho < 2) return;
169
170    int pivo = vetor[tamanho / 2];
171
172    int i, j;
173
174    for (i = 0, j = tamanho - 1; ; i++, j--)
175    {
176      while (vetor[i] < pivo) i++;
177      while (vetor[j] > pivo) j--;
178
179      if (i >= j) break;
180
181      int temp = vetor[i];
182      vetor[i] = vetor[j];
183      vetor[j] = temp;
184    }
185
186    quickSort(vetor, i);
187    quickSort(vetor + i, tamanho - i);
188  }
```

Complexidade do Quick Sort

O pior caso de particionamento ocorre quando o elemento pivô divide a lista de forma desbalanceada, ou seja, divide a lista em duas sub listas: uma com tamanho 0 e outra com tamanho $n - 1$. Isso pode ocorrer quando o elemento pivô é o maior ou menor elemento da lista, ou seja, quando a lista já está ordenada, ou inversamente ordenada.

Se isso acontece em todas as chamadas do método de particionamento, então cada etapa recursiva chamará listas de tamanho igual à lista anterior - 1. Teremos assim, a seguinte relação de recorrência:

T(n) = T(n-1) + T(0) + O(n)

T(n-1) + O(n)

Se somarmos os custos em cada nível de recursão, teremos uma série aritmética que tem valor O(n2), assim, o algoritmo terá tempo de execução igual à **O(**n^2**)**.

- ## Shell Sort

Shell sort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o array a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção.

Em resumo, a ordenação Shell utiliza a quebra sucessiva da sequência a ser ordenada e implementa a ordenação por inserção na sequência obtida. Devido a sua complexidade possui excelentes desempenhos em N muito grandes, inclusive sendo melhor que o Merge Sort.

```
204  // Shell Sort
205  void shellSort (int* vetor, int tamanho)
206  {
207      int i, j, value;
208
209      int h = 1;
210      while(h < tamanho) {
211          h = 3*h+1;
212      }
213      while (h > 0) {
214          for(i = h; i < tamanho; i++) {
215              value = vetor[i];
216              j = i;
217              while (j > h-1 && value <= vetor[j - h]) {
218                  vetor[j] = vetor[j - h];
219                  j = j - h;
220              }
221              vetor[j] = value;
222          }
223          h = h/3;
224      }
225  }
226
```

Complexidade do Shell Sort: a complexidade do algoritmo ainda não é conhecida, porém ela está em algo abaixo ou igual ao O(n^2).

- **Merge Sort**

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Como o algoritmo *Merge Sort* usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

A proposta do Merge Sort é dividir recursivamente a lista em listas menores até termos 1 elemento em cada sub-lista. Após isso, junta e ordena em partes as sublistas, até obter a lista ordenada completamente.

Usando a lista a seguir como um exemplo de como o Merge Sort funciona:

38 - 27 - 43 - 3 - 9 - 82 – 10

1º Dividindo a lista ao meio obtemos as seguintes listas:

38 - 27 - 43 - 3 9 - 82 - 10

2º Repetimos o processo e dividimos novamente as listas:

38 - 27 43 - 3 9 - 82 10

3º Divide novamente:

38 27 43 3 9 82 10

4º Após a divisão, iremos ordenar a sublista de 1 elemento com a sua “lista-mãe” de 2 elementos, deixando o menor elemento no início da lista;

27 - 38 3 - 43 9 - 82 10

5º Continuamos juntando as listas menores e ordenando-nas, como fizemos no passo quatro:

3 - 27 - 38 - 43 9 - 10 - 82

6º Ao juntar as duas sublistas, temos a lista ordenada:

3 - 9 - 10 -27 - 38 - 43 - 82

Complexidade Merge Sort:

Independente do caso (melhor, pior ou médio) o Merge Sort sempre será $O(n*logn)$. Isso ocorre porque a divisão do problema sempre gera dois sub-problemas com a metade do tamanho do problema original, com a ideia de dividir para conquistar.

- ## Radix Sort

Radix sort é um algoritmo de ordenação que ordena os elementos agrupando primeiro os dígitos individuais do mesmo valor posicional . Em seguida, ordena os elementos de acordo com sua ordem crescente/decrescente.

Suponha que temos um array de 8 elementos. Primeiro, vamos ordenar os elementos

com base no valor do lugar da unidade. Em seguida, classificaremos os elementos com

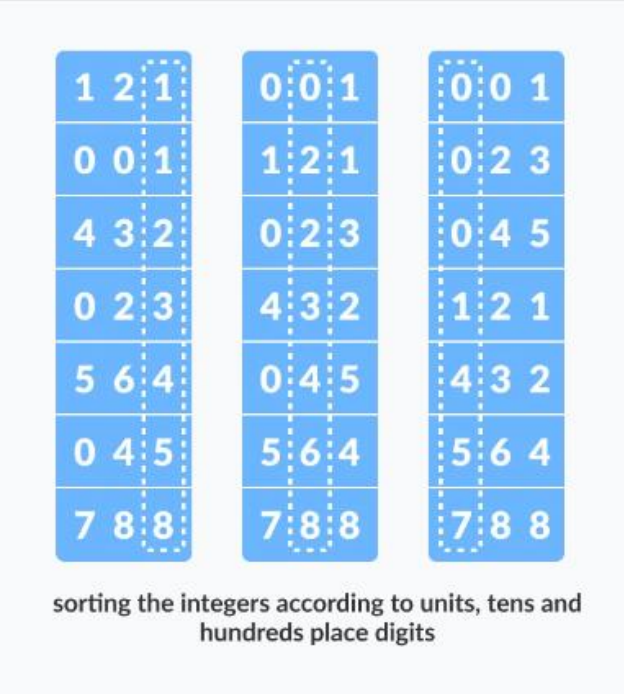
base no valor da décima posição. Este processo continua até o último lugar significativo.

Por exemplo: seja o array inicial [121, 432, 564, 23, 1, 45, 788]. Ele é classificado de acordo com a classificação radix, conforme mostrado na figura abaixo:

1. Encontre o maior elemento no array. X é o número de dígitos.X é calculado porque temos que passar por todos os lugares significativos de todos os elementos. Neste array [121, 432, 564, 23, 1, 45, 788], temos o maior número 788. Possui 3 dígitos (X=3). Portanto, o loop deve ir até a casa das centenas (3 vezes).

2. Ordenando os dígitos das unidades:

3. Agora, ordene os elementos com base nos dígitos na casa das dezenas.



4. Finalmente, ordene os elementos com base nos dígitos na casa das centenas.

Complexidade do Radix Sort:

O radix sort opera na notação Big-O, em O(nk), onde n é o número de chaves, e k é o comprimento médio da chave. É possível garantir esse desempenho para chaves com comprimento variável agrupando todas as chaves que têm o mesmo comprimento e ordenando separadamente cada grupo de chaves, do mais curto para o mais comprido, de modo a evitar o processamento de uma lista inteira de chaves em cada passo da ordenação.