Trabalho da 3ª unidade - Algoritmos e Estrutura de Dados II
Discentes: Wesley S. Costa e Ariane de Souza Valasques

Este trabalho está pautado na análise e comparação dos algoritmos de ordenação: selection, insertion, shell, heap, quick e merge sort.
Logo abaixo estão dispostos os seguintes pontos a serem abordados para cada tipo de ordenação:

- **Código-fonte dos programas.**
- **Explicação do algoritmo**
- **Complexidade**
- **Documentação dos testes.**

Parte da implementação do algoritmo para fazer as verificações de tempo e de operações de cada ordenação:

```
# (10, 100, 1000, 10000, 100000, 1000000).
x = 10
y = 200
# ---------------------------------------
vetorOrd = random.sample(range(0, y), x) #array ordenado
vetorInverOrd = random.sample(range(y, 0, -1), x) #array inversamente ordenado

med = x // 2
vetAux1 = random.sample(range(0, y), med)
vetAux2 = random.sample(range(y, 0, -1), med)

vetorQuaseOrd = vetAux1 + vetAux2 #array meio/quase ordenado

vetorAleatorio = random.sample(range(0, y), x) #array aleatorio
#Eliminando elementos repetidos nos arrays
random.shuffle(vetorAleatorio)
random.shuffle(vetorOrd)
random.shuffle(vetorInverOrd)
random.shuffle(vetorQuaseOrd)
# ---------------------------------------
print("Array antes da ordenação: ", vetorOrd)
t0 = datetime.now()
insertion_sort(vetorAleatorio)
t1 = datetime.now()
diff = t1 - t0
med = (diff.total_seconds() * 1000) / len(vetorOrd)
print( "Tempo da operacao: " + str(med) + " ms" )
print("Array depois dá ordenação: ",vetorOrd)

#Limpando os espaços dos arrays
```

del(vetorOrd, vetorAleatorio, vetorQuaseOrd, vetorInverOrd, vetAux2, vetAux1)

Análise:

Método nº 1: Selection Sort

Código-fonte dos programas:

```
def selection_sort(array):
  for index in range(0, len(array)):
      min_index = index
      for right in range(index + 1, len(array)):
          if array[right] < array[min_index]:
              min_index = right
      array[index], array[min_index] = array[min_index], array[index]
  return array
```

Explicação do algoritmo:

Este algoritmo de ordenação é baseado na análise do menor valor do array/lista, e após isto, colocar este menor valor na primeira posição, em sequência procurar os próximos valores que são maiores que o primeiro(o que já foi ordenado), verificar o menor, e colocá-lo na próxima posição, isso repetindo-se n - 1 vezes com os elementos restantes, isso até que se ordene todo o array/lista.

Este algoritmo é formado por dois laços de repetição, esses laços vão de até n - 1, onde o primeiro laço demarca o índice(**index**) que será utilizado para a troca de posições(valores entre às posições), já o segundo laço de repetição percorre o array/lista procurando o valor que é menor do que o valor que está na posição demarcado pelo índice.

Por exemplo, se pegarmos um determinado array: [9 - 7 - 8 - 1 - 2 - 0 - 4], temos os seguintes estados do array:

1. 9 - 7 - 8 - 1 - 2 - 0 - 4
2. 0 - 7 - 8 - 1 - 2 - 9 - 4
3. 0 - 1 - 8 - 7 - 2 - 9 - 4
4. 0 - 1 - 2 - 7 - 8 - 9 - 4
5. 0 - 1 - 2 - 4 - 8 - 9 - 7
6. 0 - 1 - 2 - 4 - 7 - 9 - 8
7. 0 - 1 - 2 - 4 - 7 - 8 - 9 (**array ordenado**)

Como podemos ver que o array vai sendo ordenado em cascata, onde a cada iteração o array e verificado por completo, isso a partir do valor do índice(**index**). Já na primeira iteração acontece a verificação entre o valor do array na posição **index(array[0])** para encontrar o menor valor e assim substituir no array, e esse outro valor foi o 0, este está na posição 5. e assim sucessivamente até o final do array.

Complexidade do algoritmo:

Devido às sucessivas comparações, onde a cada iteração é comparado um valor com os demais a fim de encontrar o menor, o algoritmo acaba se padronizando na questão da complexidade, ou seja, tanto no pior caso ou no caso médio, o algoritmo tem complexidade $O(n^2)$, já no melhor caso ele é $O(n)$.

Sendo assim, devido a essa padronização em sua complexidade, o selection sort acaba por ser um algoritmo de fácil implementação e que não necessita de muitas variáveis auxiliares, ou mesmo vetores auxiliares para promover a ordenação, contudo essa simplicidade e também a velocidade de ordenação só são mais expressivos em arrays pequenos, ou seja , em arrays maiores ele acaba por ter um baixo índice de eficiência, percorrendo todo o array em todas as iterações, sendo que em muitos casos isso se torna desnecessário, fazendo sempre $(n^2 - n)/2$ comparações, independentemente do array já estar ordenado ou não.

Documentação dos testes:

Foram feitos testes para comprovar a velocidade do algoritmo de ordenação, estes teste estão em milissegundos(**ms**) e foram feitos com a variação da implementação da formação do array, onde estes são: arrays ordenados, inversamente ordenados, meio ordenados e aleatórios. Estes quatro formatos de arrays foram testados com uma variação no seu tamanho, então os tamanhos variam em 10, 100, 1000, 10000, 100000 e 1000000 elementos. Segue abaixo a tabela com os valores relacionados ao tempo de execução, e logo após é feito uma análise dos resultados.

Figura 1 - Tabela de comparações dos arrays no Selection Sort

Array Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
selction	Muito rapido	0.04006 ms	0.13156 ms	1.2364144 ms	-	-
Array Inversamente Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
selction	Muito rapido	0.03997 ms	0.085811 ms	1.0159884 ms	-	-
Array Meio Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
selction	Muito rapido	0.04047 ms	0.096826 ms	1.1435888 ms	-	-
Array Aleatorio						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
selction	Muito rapido	0.06012 ms	0.096709 ms	1.1013756 ms	-	-

Fonte: Computador do Wesley

Comentar tabela.

Método nº 2: Insertion Sort

Código-fonte dos programas:

```
def insertion_sort(array):
   for index in range(1, len(array)):
       current_element = array[index]
       while index > 0 and array[index - 1] > current_element:
           array[index] = array[index - 1]
           index -= 1
```

```
    array[index] = current_element
  return array
```

Explicação do algoritmo:

Muito comparado com o método de ordenação de um baralho, o algoritmo insertion sort é um algoritmo de ordenação quadrático que está baseado na troca de valores entre posições, onde ele é iniciado a partir do segundo elemento do array ou lista, isso porque o algoritmo utiliza a análise dos valores anteriores ao valor demarcado pelo índice(**index**) no array para realizar a troca, passando de posição em posição, comparando com seus antecessores e colocando o valor no local adequado, fazendo isso sucessivas vezes, até que o array/lista esteja ordenado.

Sua formação se constitui de duas estruturas de repetição, onde a primeira(**o for**), norteia qual elemento está sendo analisado, e o outro laço de repetição(**o while**) realiza às comparações e às trocas. Como dito anteriormente, essas comparações irão “encaixar” o valor em uma determinada posição da seguinte forma:

Temos o seguinte array/lista: 9 - 7 - 8 - 1 - 2 - 0 - 4, e utilizando o insertion sort sua análise começa a partir da posição nº 1(**2º elemento do array**), ou seja começamos a analisar o número 7. Logo após fazemos a comparação com o seu antecessor para sabermos se o número 7 é menor que ou seu antecessor. Por ser verdade, ocorre a troca dos valores entre às posições, e o array fica da seguinte forma: 7 - 9 - 8 - 1 - 2 - 0 - 4. Como não tem mais elementos para ser feita a análise, finaliza-se o laço interno, e agora o próximo elemento a ser analisado é o da posição nº 2(**3º elemento do array**), analisa-se os seus antecessores e faz a troca. Isso acontece sucessivas vezes até que se ordene todo o array/lista

Complexidade do algoritmo:

De forma similar ao selection sort ou mesmo o bubble sort(*não abordaremos sua análise neste documento*), o insertion sort é um algoritmo de complexidade quadrática tanto no pior caso, quanto no caso médio, ou seja $O(n^2)$, já no melhor caso ele é $O(n)$. Isso se dá pela sua forma de analisar o array ou a lista para encontrar a posição correta de cada elemento, uma vez que a cada iteração o número de comparações entre posições pode ou não aumentar a depender do array analisado.

Nesta vertente podemos elencar alguns pontos interessante sobre as características desse algoritmo, são elas:

- Implementação e leitura simples, sem uma alta “complexidade de montagem”;
- Sua aplicação é mais eficiente para arrays ou listas de tamanho pequeno;
- Quando o array está ordenado sua complexidade é $O(n)$;
- Quando o array é aleatório sua complexidade é $O(n^2)$;
- Quando o array está em ordem inversa sua complexidade é $O(n^2)$.

Mas ele possui alguns problemas também:

- Troca de itens que são adjacentes unicamente para obter um ponto de inserção de elementos;
- Quando o menor item está alocado na posição mais à direita do array/lista, são realizadas n - 1 comparações para poder movimentá-lo e colocá-lo na posição correta.

Documentação dos testes:

Assim como no outro método de ordenação, foram feitos testes para comprovar a velocidade do algoritmo de ordenação, estes teste estão em milissegundos(**ms**). Logo para a realização destes testes foram tomadas às mesmas diretrizes do teste anterior(**Algoritmo 1: Selection Sort**). Segue abaixo a tabela de comparações, e em sequência uma análise dos dados obtidos, entre outros pontos.

Figura 2 - Tabela de comparações dos arrays no Insertion Sort

Array Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
insertion	Muito rapido	0.04 ms	0.231971 ms	1.1289456 ms	-	-
Array Inversamente Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
insertion	Muito rapido	0.04001 ms	0.127782 ms	1.1064011 ms	-	-
Array Meio Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
insertion	Muito rapido	0.04 ms	0.103663 ms	1.106767 ms	-	-
Array Aleatorio						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
insertion	Muito rapido	0.09715 ms	0.09908 ms	1.0934399 ms	-	-

Fonte: Computador do Wesley

COMENTAR A TABELA.

Método nº 3: Shell Sort

Código-fonte dos programas:

```
def shell_sort(array):
  gap = len(array) // 2
  while gap > 0:
     for i in range(gap, len(array)):
        val = array[i]
        j = i
        while j >= gap and array[j - gap] > val:
           array[j] = array[j - gap]
           j = j - gap
        array[j] = val
     gap //= 2
  return array
```

Explicação do algoritmo:

Este algoritmo de ordenação é uma atualização do método anterior, onde problemas aos quais o insertion sort sofre, são ultrapassados pelo shell sort. Sendo assim, este método acabou se tornando o mais eficiente entre os algoritmo de complexidade quadrática, uma vez que ao invés dele executar a análise por posições e sempre verificar os seus antecessores, o shell faz a análise por segmentos(**permitindo trocas de registros distantes**), para isso ele utiliza a variável **gap** - isso no algoritmo que estamos analisando -, esta variável delimita os elementos que que serão comparados, onde este distanciamento é demarcado pela metade do tamanho do array/lista.

Deste modo a análise fica mais eficiente, e a partir disso analisa-se se um valor é menor que o outro, logo após acontece a troca dos valores nas posições. Depois **gap** é atualizado tendo seu valor reduzido pela metade, contudo isso só acontece quando às análises anteriores acabam e alguns valores trocam ou não de posição(isso depende do array que está sendo analisado), após isto é retomado às análises,, agora com um intervalo menor, e isso vai acontecendo até que o array esteja totalmente ordenado.

Complexidade do algoritmo:

Este algoritmo tem complexidade quadrática, contudo isso acaba tendo uma alteração visto que a composição do array/lista pode ser diferente em vários casos, ou seja no melhor caso ele pode se O(n log n), já no caso médio e no pior caso ele pode ser O(n^2). Isso se da pelas diferenciação de casos a serem abordados e ordenados pelo algoritmo, onde alguns arrays podem exigir mais ou menos comparações, inferindo diretamente na eficiência do algoritmo e também em seu tempo de execução, ou seja com a troca de itens remotos tem se um ganho em eficiência, mas isto não depende unicamente do método, mas também do objeto a ser analisado, no caso o array ou a lista.

Documentação dos testes:

Assim como no outro método de ordenação, foram feitos testes para comprovar a velocidade do algoritmo de ordenação, estes teste estão em milissegundos(**ms**). Logo para a realização destes testes foram tomadas às mesmas diretrizes dos teste anteriores. A seguir está a tabela de comparações, e em sequência uma análise dos dados obtidos, entre outras coisas.

Figura 3 - Tabela de comparações dos arrays no Shell Sort

Array Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
shell	Muito rapido	0.039589 ms	0.007832 ms	0.0106379 ms	0.0202778 ms	-
Array Inversamente Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
shell	Muito rapido	0.04 ms	0.003719 ms	0.0119652 ms	0.01991962 ms	-
Array Meio Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
shell	Muito rapido	0.04 ms	0.003839 ms	0.01302099 ms	0.02088724 ms	-
Array Aleatorio						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
shell	Muito rapido	0.04001 ms	0.003969 ms	0.0111265 ms	0.01971402 ms	-

Fonte: Computador do Wesley

COMENTAR A TABELA.

Método nº 4: Quick Sort

Código-fonte dos programas:

```
def partition(arr, low, high):#Função auxiliar
  i = (low - 1)
  pivot = arr[high]
  for j in range(low, high):
     if arr[j] < pivot:
        i = i + 1
        arr[i], arr[j] = arr[j], arr[i]
  arr[i + 1], arr[high] = arr[high], arr[i + 1]
  return (i + 1)

def quick_sort(array, low=0, high=None):
  if high == None:
     high = len(array) - 1
  if low < high:
     pi = partition(array, low, high)
     quick_sort(array, low, pi - 1)
     quick_sort(array, pi + 1, high)
  return array
```

Explicação do algoritmo:

Na ampla lista de algoritmos de ordenação, o quick sort é um dos mais utilizados, isso por causa de sua ampla capacidade de ordenação em vários casos, tornando se um dos métodos mais conhecidos. Este algoritmo segue pela vertente da divisão do problema em partes menores, e depois de resolvê-las(ordenar às partes) juntam-se às partes para formar um array/lista completamente ordenado. Nesse sentido, apesar de ser mais complexo em questão de passos de ordenação, ele acaba por ser um dos mais eficientes.

O algoritmo é recursivo, justamente para auxiliar na divisão do array, uma vez que esta parte é a minuciosa e característica do algoritmo, além do mais, é utilizado também uma função que auxilia a ordenação, esta função se chama(no nosso caso): **partition,** e como o nome sugere, ela realiza o processo de particionamento do array/lista, análise dos termos e troca dos elementos nas posições do array/lista.

Para promover o particionamento do array, é escolhido um pivô, no nosso caso ele é o **pivot,** a partir da seleção deste pivô é possível dividir o array, após isto, com o auxílio da recursão a primeira parte do array/lista será percorrido e ordenado com os elementos do array sendo menores que o pivô, em seguida a outra parte, que por sua vez tem os elementos maiores que o pivô, são ordenados, ao final, juntam-se às partes e tem-se o array/lista ordenado.

Complexidade do algoritmo:

Este algoritmo possui complexidade O(n log n) isto para o seu melhor caso, pois a partir da variação do objeto a ser analisado(**array/lista**), portanto no pior caso o algoritmo tem complexidade $O(n^2)$. Sendo assim é um algoritmo com boa eficiência, isso vem através de suas vantagens, onde ele é muito bom na ordenação de arquivos de dados, utiliza-se dá recursão, assim necessitando de uma pequena pilha para auxiliar no particionamento, sendo essa pilha uma memória auxiliar para o algoritmo; e outro ponto é a própria questão de sua complexidade que necessita de n log n comparações para realizar a ordenação.

Contudo este algoritmo possui às suas desvantagens, por exemplo a questão de ter uma implementação mais complexa e delicada, onde os pontos de análise devem estar corretos e bem definidos, em caso contrário, o algoritmo não irá realizar a ordenação de forma correta, e como no material fornecido pelo professor diz: "...pode levar a efeitos inesperados para algumas entradas de dados".

Documentação dos testes:

Assim como nos outros métodos de ordenação, foram feitos testes para comprovar a velocidade do algoritmo de ordenação, estes teste estão em milissegundos(**ms**). Logo para a realização destes testes foram tomadas às mesmas diretrizes dos teste anteriores. A seguir está a tabela de comparações, e em sequência uma análise dos dados obtidos, entre outras coisas.

Figura 4 - Tabela de comparações dos arrays no Quick Sort

Array Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
quick	Muito rapido	0.03967 ms	0.003996 ms	0.0068038 ms	0.00747969 ms	-
Array Inversamente Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
quick	Muito rapido	0.04 ms	0.007938 ms	0.0051668 ms	0.00707963 ms	-
Array Meio Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
quick	Muito rapido	0.039979 ms	0.004039 ms	0.0051932 ms	0.00748256 ms	-
Array Aleatorio						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
quick	Muito rapido	0.04 ms	0.007839 ms	0.0067759 ms	0.00768136 ms	-

Fonte: Computador do Wesley

COMENTAR TABELA.

Método nº 5: Heap Sort

Código-fonte dos programas:

```
def heapify(arr, n, i):
  largest = i
  l = 2 * i + 1
  r = 2 * i + 2
  if l < n and arr[i] < arr[l]:
     largest = l
  if r < n and arr[largest] < arr[r]:
     largest = r
  if largest != i:
     arr[i], arr[largest] = arr[largest], arr[i]
     heapify(arr, n, largest)


def heap_sort(array):
```

```
n = len(array)
# constroi heapmax
for i in range(n, -1, -1):
    heapify(array, n, i)
    # remove os elementos 1 a 1
for i in range(n - 1, 0, -1):
    array[i], array[0] = array[0], array[i]
    heapify(array, i, 0)
return array
```

Explicação do algoritmo:

Com princípio similar ao do selection sort, o heap sort também utiliza-se de sucessivas buscas pelo menor valor e a própria troca deste elemento com o que está na primeira posição do array. Entretanto este algoritmo se difere pelo fato de impor uma prioridade no momento da ordenação e dá verificação dos elementos, isto é chamado de: fila de prioridades. Isto acaba por dar mais eficiência para o algoritmo, uma vez que descarta-se a utilização de sucessivas e incessantes verificações em todo array para poder ordená-lo.

Para tanto, uma forma de analisar e visualizar todo esse processo de ordenação e prioridades, é o de considerarmos o array como uma árvore binária(estrutura de dados), este parâmetro de análise é chamado de **heap** - daí o nome do método **heap sort** -, a partir disto a troca dos elementos passa a ser mediante a análise entre às folhas e o seu “pai”, onde, se uma das folhas for maior que o seu “pai”, ocorre a troca dos valores.

Após estas comparações, encontra-se o maior valor, que por sua vez, depois do vetor ser reordenado, ele será o nó raiz(nó pai), com isso ele será trocado pelo último nó e por já estar ordenado, ele é “eliminado” da análise, uma vez que ele já é o maior valor do array. Em seguida a análise continua efetuando esses passos e ordenando o array. Sendo assim o heap sort é uma das formas de maior eficiência a depender dos casos.

Complexidade do algoritmo:

Este algoritmo tem complexidade de O(n log n),e isso se adequa ao caso médio, ao melhor caso e ao pior também. As trocas e comparações realizadas podem ser descritas da seguinte forma:

- Comparações no pior caso: $2n \log_2 n + O(n)$ é o mesmo que $2n \lg n + O(n)$
- Trocas no pior caso: $n \log_2 n + O(n)$ é o mesmo que $n \lg n + O(n)$
- Melhor e pior caso: $O(n \log_2 n)$ é o mesmo que $O(n \lg n)$

Contudo, uma das desvantagens do heap sort é a questão de não ser um algoritmo estável, ou seja, continuar a fazer os mesmo passos para elementos semelhantes dentro do array/lista, outro ponto é a própria construção do heap para poder realizar as trocas, a construção dele influencia diretamente o tempo de execução, fazendo com que este algoritmo não seja recomendado para arquivos com poucos registros.

Documentação dos testes:

Da mesma forma que os outros métodos de ordenação, foram feitos testes para comprovar a velocidade do algoritmo de ordenação, estes teste estão em milissegundos(**ms**). Logo para a realização destes testes foram tomadas às mesmas diretrizes dos teste anteriores. A seguir está a tabela de comparações, e em sequência uma análise dos dados obtidos, entre outras coisas.

Figura 5 - Tabela de comparações dos arrays no Heap Sort

Array Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
heap	Muito rapido	0.039979 ms	0.007829 ms	0.0191738 ms	0.01911963 ms	-
Array Inversamente Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
heap	Muito rapido	0.03988 ms	0.007718 ms	0.0155743 ms	0.01924002 ms	-
Array Meio Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
heap	Muito rapido	0.04058 ms	0.007826 ms	0.0154775 ms	0.01962106 ms	-
Array Aleatorio						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
heap	Muito rapido	0.04084 ms	0.003781 ms	0.0228516 ms	0.01901682 ms	-

Fonte: Computador do Wesley

COMENTAR TABELA

Método nº 6: Merge Sort

Código-fonte dos programas:

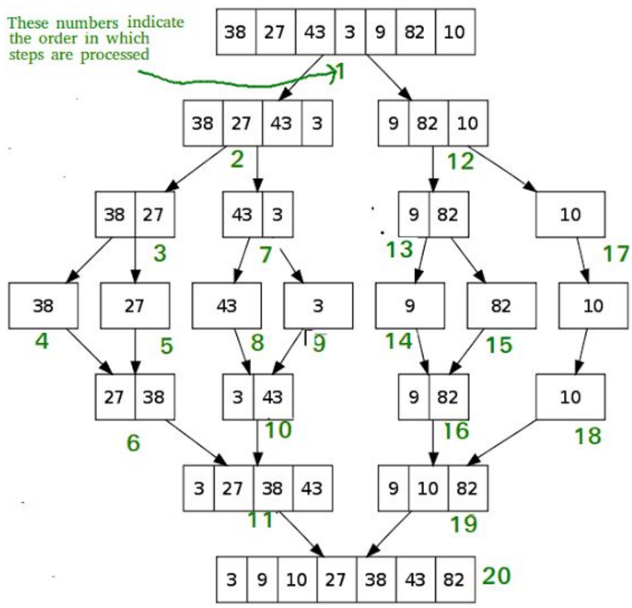
```
def merge_sort(array):
   if len(array) > 1:
       mid = len(array) // 2
       L = array[:mid]
       R = array[mid:]
       merge_sort(L)
       merge_sort(R)
       i = j = k = 0
       while i < len(L) and j < len(R):
           if L[i] < R[j]:
               array[k] = L[i]
               i += 1
           else:
               array[k] = R[j]
               j += 1
           k += 1

       while i < len(L):
           array[k] = L[i]
           i += 1
           k += 1

       while j < len(R):
           array[k] = R[j]
           j += 1
           k += 1
```

return array

Explicação do algoritmo:

Por fim chegamos a explicação do merge sort, este algoritmo de ordenação utiliza dos princípios já vistos aqui de separar o problema em partes menores e assim resolvê-los, e em seguida juntar tudo. Isto está diretamente relacionado a intercalação no processo de divisão em partes menores e ordenação destas partes, uma vez que neste método a utilização da recursão é imprescindível para que se possa analisar e comparar os subconjuntos formados a partir do array original.

Está ordenação se inicia a partir do momento em que os sub-conjuntos(sub-problemas) chegam ao tamanho 1, ou seja, quando os arrays auxiliares tem apenas 1 elemento, neste momento a recursão acaba e os valores são ordenados comparando os arrays auxiliares e assim remontando o array, isso acontece até o array voltar ao tamanho original e ser ordenado completamente. Logo abaixo esta representação de como o algoritmo funciona:



Fonte: Site Geeks for geeks

Complexidade do algoritmo:

Este algoritmo possui complexidade O(n log n), isso para os três casos(pior, melhor e caso médio), contudo, devido ao alto consumo de memória o merge sort não e torna um algoritmo tão eficiente, contrastando com os algoritmos anteriores, entretanto ele ainda é mais eficaz do por exemplo o selection sort ou mesmo o insertion sort.

Documentação dos testes:

Da mesma forma que os outros métodos de ordenação, foram feitos testes para comprovar a velocidade do algoritmo de ordenação, estes teste estão em milissegundos(**ms**). Logo para a realização destes testes foram tomadas às mesmas diretrizes dos teste anteriores. A seguir está a tabela de comparações, e em sequência uma análise dos dados obtidos, entre outras coisas.

Figura 5 - Tabela de comparações dos arrays no Heap Sort

Array Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
merge	Muito rapido	0.04033 ms	0.016047 ms	0.0095994 ms	0.01232001 ms	-
Array Inversamente Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
merge	Muito rapido	0.03997 ms	0.007962 ms	0.0107666 ms	0.01387968 ms	-
Array Meio Ordenado						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
merge	Muito rapido	0.04035 ms	0.012116 ms	0.0108346 ms	0.01300782 ms	-
Array Aleatorio						
tamanho do array / método de ordenação	10	100	1000	10000	100000	1000000
merge	Muito rapido	0.03975 ms	0.007999 ms	0.0187138 ms	0.01237453 ms	-

Fonte: Computador do Wesley

COMENTAR TABELA

Considerações finais:

Bibliografias:

- Material fornecido pelo professor: ordenacao.pdf
- Selection Sort. Geekies for geekies, Desconhecido. Disponível em: <https://www.geeksforgeeks.org/selection-sort/>. Acesso em: 28 de Outubro de 2019.
- Algoritmos de Ordenação: Selection Sort. Medium, 2018. Disponível em: <https://medium.com/@henriquebraga_18075/algoritmos-de-ordenação-ii-selection-sort-8ee4234deb10>. Acesso em: 28 de Outubro de 2019.
- Selection sort. Wikipedia, Desconhecido. Disponível em: <https://pt.wikipedia.org/wiki/Selection_sort>. Acesso em: 28 de Outubro de 2019.
- Insertion sort. Khan Academy, Desconhecido. Disponível em: <https://pt.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort>. Acesso em: 27 de Outubro de 2019.
- HENRIQUE, Braga. Algoritmos de Ordenação: Insertion Sort. Medium, 2018. Disponível em:<https://medium.com/@henriquebraga_18075/algoritmos-de-ordenação-iii-insertion-sort-bfade66c6bf1>. Acesso em: 27 de Outubro de 2019.

- Insertion Sort. Wikipédia, Desconhecido. Disponível em: <https://pt.wikipedia.org/wiki/Insertion_sort#Análise_com_outros_algoritmos_de_ordenação_por_comparação_e_troca>. Acesso em: 28 de Outubro de 2019.
- MENOTTI, David. Ordenação - Shellsort. UFPR, Desconhecido. Disponível em: <http://web.inf.ufpr.br/menotti/ci056-2015-2-1/slides/aulaORDShellSort.pdf>. Acesso em: 28 de Outubro de 2019.
- Ordenação: Shellsort. UFMG, ano. Disponível em: <https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/shellsort.pdf>. Acesso em: 28 de Outubro de 2019.
- Shell Sort. Wikipédia, Desconhecido. Disponível em: <https://pt.wikipedia.org/wiki/Shell_sort>. Acesso em: 28 de Outubro de 2019.
- ShellSort. Geeks for Geeks, Desconhecido. Disponível em: <https://www.geeksforgeeks.org/shellsort/>. Acesso em: 28 de Outubro de 2019.
- Quicksort. Instituto de Matemática e Estatística | IME-USP, Desconhecido. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/quick.html>. Acesso em: 28 de Outubro de 2019.
- Visão geral do quicksort. Khan Academy, Desconhecido. Disponível em: <https://pt.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort>. Acesso em: 28 de Outubro de 2019.
- Quicksort. Wikipédia, Desconhecido. Disponível em: <https://pt.wikipedia.org/wiki/Quicksort>. Acesso em: 28 de Outubro de 2019.
- Ordenação Quicksort. DCC/FCUP - Universidade do Porto , 2018. Disponível em: <https://www.dcc.fc.up.pt/~pbv/aulas/progimp/teoricas/teorica18.html>. Acesso em: 28 de Outubro de 2019.
- Heapsort. Instituto de Matemática e Estatística | IME-USP, Desconhecido. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>. Acesso em: 28 de Outubro de 2019.
- Heapsort. Wikipédia, Desconhecido. Disponível em: <https://pt.wikipedia.org/wiki/Heapsort>. Acesso em: 28 de Outubro de 2019.

- Merge Sort. Geeks for Geeks, Desconhecido. Disponível em: <https://www.geeksforgeeks.org/merge-sort/>. Acesso em: 28 de Outubro de 2019.
- Merge sort. Wikipédia, Desconhecido. Disponível em: <https://pt.wikipedia.org/wiki/Merge_sort>. Acesso em: 28 de Outubro de 2019.