Análise de desempenho dos algoritmos de ordenação Heapsort, Quicksort e Shell Sort

Hélio Rocha V. de C. Júnior[1], Jederson Sousa Luz[1], Patrick Ryan Sales dos Santos[1], Vitória de Carvalho Brito[1]

[1] Universidade Federal do Piauí (UFPI)
Campus Senador Helvídio Nunes de Barros

Abstract. *This work presents complexity and runtime analyzes of three of the most important sort algorithms, Heap Sort, Quicksort, and ShellSort. Choosing an appropriate sort method should be the first action to take before any implementation, because for each problem there is a more appropriate sorting method (s), and a wrong choice can greatly affect software performance.*

Resumo. *Este trabalho apresenta análises de complexidade e de tempo de execução de três dos principais algoritmos de ordenação existentes, o Heapsort, o Quicksort e o Shell Sort. A escolha de um método de ordenação adequado deve ser a primeira ação a se fazer antes de qualquer implementação, pois para cada problema há um método (ou métodos) de ordenação mais indicado, e uma escolha equivocada pode afetar consideravelmente o desempenho do software.*

1. Introdução

A ordenação é uma das primitivas mais importantes em vários sistemas, por exemplo, sistemas de banco de dados, uma vez que é frequentemente conduzida e consome muito tempo. Para obter um algoritmo de ordenação praticamente eficiente, os pesquisadores não apenas investigaram a complexidade computacional, mas também o desempenho experimental. Embora a complexidade computacional seja uma boa métrica assintótica de eficiência, às vezes um algoritmo de ordenação inferior (no sentido de complexidade computacional) excede o desempenho experimental de algoritmos superiores. Por exemplo, o quicksort é mais popular do que o merge sort, já que o quicksort geralmente tem melhor desempenho, embora sua complexidade computacional seja pior do que a do algoritmo Merge Sort. Uma das classes mais famosas de ordenação é por comparação. Uma ordenação por comparação determina a ordem com base apenas nas comparações entre os elementos de entrada. As ordenações por comparação incluem vários algoritmos de ordenação bem conhecidos e eficientes, como quicksort, shell sort, heapsort e merge sort [Hamada et al. 2013].

Este trabalho tem como objetivo realizar um estudo sobre alguns dos algoritmos de ordenação por comparação, analisando seu desempenho a partir de testes com diferentes entradas, levando em consideração o melhor caso, o pior caso e o caso médio. Esses algoritmos são o heapsort, o quicksort e o shell sort.

O trabalho está organizado da seguinte forma: a Seção 2 apresenta as definições dos algoritmos estudados; A Seção 3 aborda a metodologia utilizada para a execução dos testes de desempenho; A Seção 4 apresenta e discute os resultados obtidos na execução da metodologia; e, por fim, a Seção 6 mostra a conclusão deste trabalho.

2. Referencial teórico

A fim de proporcionar um claro entendimento sobre a metodologia dos testes de desempenho apresentados neste trabalho, este tópico explana cada um dos algoritmos de ordenação utilizados, apresentando suas características e explicando sua funcionalidade.

2.1. Heapsort

O heapsort introduz outra técnica de projeto de algoritmos: o uso de uma estrutura de dados, nesse caso uma estrutura que chamamos "heap"(ou "monte") para gerenciar informações durante a execução do algoritmo. A estrutura de dados heap não é útil apenas para o heap- sort (ou ordenação por heap); ela também cria uma eficiente fila de prioridades [Cormen et al. 2009].

O algoritmo heapsort é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção. Foi desenvolvido em 1964 por Robert W. Floyd e J.W.J Williams. Levando em consideração o tempo de execução, o heapsort tem um desempenho satisfatório em conjuntos de dados ordenados aleatoriamente, tem um uso de memória estável e a grande vantagem que deve ser destacada é que seu desempenho no caso médio é praticamente igual ao desempenho no seu pior caso, tendo assim um comportamento invariável em relação aos seus casos. Na literatura existem algoritmos que trabalham bem no melhor caso e no caso médio, porém o heap consegue entregar O (n lg n) no seu pior caso, ordem essa considerada satisfatória para um n suficientemente grande.

O funcionamento do heapsort é baseado na estrutura heap. Assim, ao final das inserções os elementos podem ser removidas da raiz da heap, na ordem desejada. O heap pode ser representada como uma árvore, uma árvore binária com propriedades especiais, ou como um vetor para uma ordenação decrescente, deve ser construída uma heap mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma heap máxima (o maior elemento fica na raiz). A Figura 1 apresenta o algoritmo heapsort escrito na linguagem python.

2.2. Quicksort

O quicksort (ordenação rápida) é um algoritmo de ordenação cujo tempo de execução do pior caso é 0(n^2) sobre um arranjo de entrada de n números. Apesar desse tempo de execução lento no pior caso, o quicksort com frequência é a melhor opçáo prática para ordenação, devido a sua notável eficiência na média: seu tempo de execuçáo esperado é O(n lg n), e os fatores constantes ocultos na notação O(n lg n) são bastante pequenos. [Cormen et al. 2009].

O algoritmo quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante. O tempo de execução de quicksort depende do fato de o particionamento ser balanceado ou não balanceado, e isso por sua vez depende de quais elementos são usados para particionar. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente de forma tão lenta quanto a ordenação por inserção [Cormen et al. 2009]. O pior caso de particionamento ocorre quando o elemento pivô divide a lista de forma $\Theta(n^2)$, no melhor caso e no caso médio

```python
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 heapSort(arr):
    n = len(arr)
    for i in range(n, 0, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Figura 1. Algoritmo heapsort. Adaptado de [GeeksforGeeks 2017a].

e temos um complexidade igual a $\Theta(log^2 n)$. A Figura 2 apresenta o algoritmo Quicksort escrito na linguagem python.

```python
class Quick(object):
    def particao(self, a, ini, fim):
        pivo = a[fim-1]
        start = ini
        end = ini
        for i in range(ini,fim):
            if a[i] > pivo:
                end += 1
            else:
                end += 1
                start += 1
                a[start-1], a[i] = a[i], a[start-1]
        return start-1

    def quickSort(self, a, ini, fim):
        if ini < fim:
            pp = self.randparticao(a, ini, fim)
            self.quickSort(a, ini, pp)
            self.quickSort(a, pp+1,fim)
        return a

    def randparticao(self,a,ini,fim):
        rand = random.randrange(ini,fim)
        a[fim-1], a[rand] = a[rand], a[fim-1]
        return self.particao(a,ini,fim)
```

Figura 2. Algoritmo Quicksort. Adaptado de [Wikipedia contributors 2019].

2.3. Shell sort

O algoritmo básico de Shellsort está entre os primeiros métodos de ordenação a serem descobertos por [L. Shell 1959] e é o mais eficiente entre os de complexidade quadrática (não utiliza recursividade). Esse algoritmo é uma extensão do algoritmo de ordenação por inserção, com a diferença de que a ordenação shell utiliza a quebra sucessiva da sequência a ser ordenada e implementa a ordenação por inserção na sequência obtida. Além disso, enquanto o método de inserção troca apenas elementos adjacentes, o shellsort pode trocar elementos distantes um do outro.

O algoritmo funciona da seguinte forma: os itens separados por h posições são agrupados e ordenados, o valor de h é decrementado seguindo uma sequência qualquer (não há uma sequência definitiva) até que atinja o valor unitário, onde a ordenação corresponde ao algoritmo de inserção, mas nesse ponto nenhum elemento precisa se mover para uma posição muito distante [Ziviani 2004]. A Figura 3 apresenta o algoritmo shellsort escrito na linguagem python.

```
def shellSort(array):
    "Shell sort using Shell's (original) gap sequence: n/2, n/4, ..., 1."
    gap = len(array) // 2
    # loop over the gaps
    while gap > 0:
        # do the insertion sort
        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 -= gap
            array[j] = val
        gap //= 2
```

Figura 3. Algoritmo shell sort. Adaptado de [GeeksforGeeks 2017b].

Quando é considerado o melhor caso (ou seja, quando os elementos do vetor já estão ordenados) e o pior caso (quando os elementos do vetor estão na ordem reversa), a melhor complexidade conhecida é: O(n log n). No entanto, quando o caso considerado é o caso médio, a complexidade depende de h, portanto não pode ser definida.

3. Metodologia

Os algoritmos de ordenação apresentados neste trabalho foram escritos na linguagem de programação python e executados em duas máquinas diferentes. A primeira máquina é um notebook Acer, que possui um processador Intel Core i3-6100U, com 4 núcleos de 2.3GHz, e uma memória RAM de 8GB. A segunda é um desktop HP, com uma memória RAM de 8GB e um processador Intel Core i5-7500, que dispõe de 4 núcleos de 3.8GHz.

Os casos de teste foram divididos em três: o melhor caso, o caso médio e o pior caso. O melhor caso é quando o vetor de entrada está totalmente ordenado, o caso médio é quando a ordem dos elementos no vetor foi definida aleatoriamente, e o pior caso é quando os elementos estão totalmente desordenados no vetor.

Os testes foram organizados da seguinte forma: para cada caso de teste foram calculados o tempo de execução de cada algoritmo para 3 conjuntos de entradas diferentes, ou seja, a primeira entrada com 50.000 números, a segunda com 100.000 e a terceira com 500.000. Esse processo foi realizado nas duas máquinas citadas no primeiro parágrafo.

4. Resultados

Este tópico apresenta os resultados obtidos a partir da execução da metodologia apresentada na Seção 3. As Tabelas 1, 2 e 3 mostram os tempos de execução alcançados pelos 3 algoritmos quando executados na máquina 1, enquanto as Tabelas 4, 5 e 6 apresentam os tempos de execução dos mesmos algoritmos, só que executados na máquina 2.

Tabela 1. Melhor caso utilizando a máquina 1.

N	50000	100000	500000
Heapsort	1,78096	2,82331	11,46178
Quicksort	1,26722	1,83107	7,95973
Shell sort	***0,56578***	***0,96425***	***3,55115***

Tabela 2. Caso médio utilizando a máquina 1.

N	50000	100000	500000
Heapsort	1,18399	3,17329	13,20231
Quicksort	***0,99155***	***1,86501***	***8,47488***
Shell sort	1,43306	2,60189	18,15181

Tabela 3. Pior caso utilizando a máquina 1.

N	50000	100000	500000
Heapsort	1,25361	2,6405	10,4845
Quicksort	0,98008	1,5911	8,83264
Shell sort	***0,65664***	***1,12005***	***5,57095***

Tabela 4. Melhor caso utilizando a máquina 2.

N	50000	100000	500000
Heapsort	0,57657	1,22789	6,70795
Quicksort	0,42782	0,85980	4,50282
Shell sort	***0,17244***	***0,36199***	***1,98534***

Tabela 5. Caso médio utilizando a máquina 2.

N	50000	100000	500000
Heapsort	0,61089	1,34302	7,76350
Quicksort	***0,44331***	***0,93761***	***5,01390***
Shell sort	0,65292	1,49029	10,37809

Tabela 6. Pior caso utilizando a máquina 2.

N	50000	100000	500000
Heapsort	0,53615	1,13496	6,28293
Quicksort	0,41137	0,85896	4,46352
Shell sort	***0,27725***	***0,59009***	***3,17710***

Analisando essas tabelas, percebe-se que os algoritmos possuem um tempo de execução menor quando executados na máquina 2. Portanto, a fim de possibilitar uma análise geral dos algoritmos, independente de qual máquina estejam sendo processados, foi feita uma média do tempo de execução obtido nas duas máquinas para o maior conjunto de entradas aplicado nos testes (N = 500.000), já que é importante considerar um N suficientemente grande na hora de analisar a complexidade de um algoritmo. A Figura 4 ilustra esse resultado.

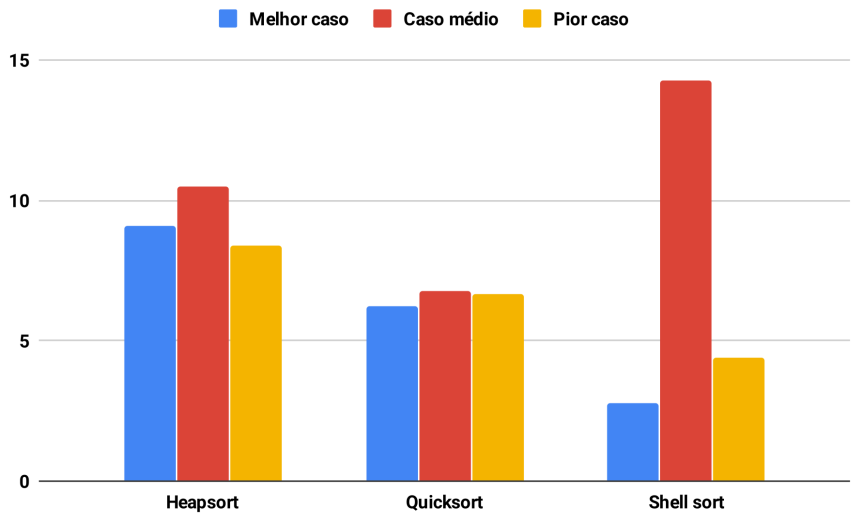

Figura 4. Média dos tempos de execução dos algoritmos nas duas máquinas para N = 500.000.

5. Discussão

Analisando os resultados apresentados na Seção 4, percebe-se que o algoritmo shell sort tem um melhor desempenho quando aplicado ao melhor e pior caso de teste. Já o algoritmo quicksort mostra-se mais eficiente que os demais quando executado no caso médio, enquanto o heapsort não apresentou resultados eficientes, se comparado aos demais.

Seguindo a análise, percebe-se que o tempo de execução não é uma métrica de desempenho adequada, já que os resultados obtidos contradizem as ordens assintóticas dos algoritmos, encontradas na literatura, como apresentado na Tabela 7. Acreditamos que essa diferença entre os resultados esperados e os encontrados devam-se ao fato de que os processos que estão sendo executados no sistema operacional da máquina interferem na velocidade da execução dos algoritmos, ainda que estejam implementados na mesma linguagem de programação.

Tabela 7. Complexidades dos algoritmos encontradas na literatura.

	Melhor caso	**Caso médio**	**Pior caso**
Heapsort	O(n log n)	O(n log n)	O(n log n)
Quicksort	O(n log n)	O(n log n)	O(n^2)
Shell sort	O(n log n)	Depende do gap	O(n log n)

6. Conclusão

Este trabalho buscou analisar o desempenho de três dos principais algoritmos de ordenação presentes na literatura: heapsort, quicksort e shell sort. Para essa análise, foi utilizada como métrica de desempenho o tempo de execução dos algoritmos, que foram executados com três conjuntos de entradas diferentes e nos três casos de teste: melhor, pior e caso médio.

Com base nos resultados obtidos, pode-se inferir que o shell sort é o mais indicado para o melhor e pior caso, enquanto no caso médio o quicksort tem um melhor desempenho. No entanto, ao compararmos tal conclusão com as complexidades esperadas para os algoritmos, entende-se que apenas o tempo não é uma boa métrica de desempenho, pois pode variar bastante dependendo da máquina e da linguagem de programação, portanto, é importante levar em conta as principais operações dos algoritmos ao analisar a complexidade, além de verificar o espaço de memória de cada um utiliza.

Com isso, podemos concluir que a escolha do algoritmo de ordenação ideal depende de vários fatores, principalmente da aplicação à qual está sendo destinado. De modo geral, o heapsort tem um desempenho satisfatório em conjuntos de dados ordenados aleatoriamente. Já o quicksort é rápido e eficiente, dependendo do tipo de particionamento e do pivô escolhido. Por fim, o shell sort é uma ótima opção para listas de tamanho médio, pois é um método simples e eficiente.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

GeeksforGeeks (2017a). Heapsort. `https://www.geeksforgeeks.org/heap-sort/`. [Online; accessed 25-April-2019].

GeeksforGeeks (2017b). Shellsort. `https://www.geeksforgeeks.org/shellsort/`. [Online; accessed 25-April-2019].

Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., and Takahashi, K. (2013). Practically efficient multi-party sorting protocols from comparison sort algorithms. In Kwon, T., Lee, M.-K., and Kwon, D., editors, *Information Security and Cryptology – ICISC 2012*, pages 202–216, Berlin, Heidelberg. Springer Berlin Heidelberg.

L. Shell, D. (1959). A high-speed sorting procedure. *Communications of the ACM*, 2:30–32.

Wikipedia contributors (2019). Quicksort — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=894166511`. [Online; accessed 25-April-2019].

Ziviani, N. (2004). *Projeto de algoritmos: com implementações em Pascal e C*. Pioneira Thomson Learning.