UNIVERSIDADE FEDERAL DE SÃO JOÃO DELREI

TP 2 - AEDS II

Análise dos Algorítimos de Ordenação

Alunos: Davi Medeiros Gonçalves (212050085)

Rafael Carvalho Avidago Geraldo (212050057)

Professor: Rafael Sachetto Oliveira

Introdução

O segundo trabalho prático de AEDS II, consistiu em uma analise de diferentes metodos de ordenacao:

Insertion Sort, Selection Sort, Shellsort, QuickSort, HeapSort e MergeSort com determinadas quantidades de chaves, sendo elas **20, 500, 5000, 10000 e 200000** elementos.
Os testes foram feitos utilizando um computador com configuração:
I5 11400H -Hexa Core- 2,5GHz (Boost 4,5 GHz) // 16 GB RAM -dual channel- 3200MHz

Implementação

Void heapsort: O Heapsort é um algoritmo de ordenação eficiente que utiliza uma estrutura de dados chamada Heap para organizar os elementos de um array. A etapa de construção do Heap é fundamental para estabelecer a propriedade do Heap, onde cada nó pai é maior ou igual aos seus nós filhos. Durante a construção, percorremos o array a partir do elemento do meio até o primeiro, chamando a função "refazer" em cada iteração. A função "refazer" compara o nó pai com seus nós filhos, realizando trocas quando necessário para manter a propriedade do Heap. Ao final do processo de construção, o array será transformado em um Heap.

Na etapa de refazer do Heapsort, garantimos a propriedade do Heap após a construção inicial. A função "refazer" é chamada durante a construção do Heap e também no processo de remoção dos elementos para obter a sequência ordenada. Ela recebe como parâmetros o índice do nó pai, o índice do último elemento do Heap, o array de itens, bem como os contadores de comparações e atribuições. A função compara o nó pai com seus nós filhos, selecionando o maior elemento para trocar de posição. Esse processo é repetido até que a propriedade do Heap seja restabelecida. A função "refazer" desempenha um papel fundamental na manutenção da estrutura do Heap durante o HeapSort.

O Heapsort é concluído na etapa de ordenação propriamente dita. Após a construção do Heap, iniciamos o processo de remoção dos elementos para obter a sequência ordenada. Inicialmente, trocamos o primeiro elemento do Heap, que possui o maior valor, com o último elemento não ordenado. Em seguida, reduzimos o tamanho do Heap em uma unidade e chamamos a função "refazer" novamente para ajustar a estrutura do Heap. Repetimos esse processo até que todos os elementos tenham sido removidos e o Heap seja esvaziado. Ao final do Heapsort, o array estará ordenado de forma crescente. O Heapsort possui um desempenho eficiente devido à utilização da estrutura de dados Heap e sua complexidade é O(n log n), tornando-o uma escolha viável para ordenar grandes conjuntos de dados.

Void inserção: A função insercao() implementa o algoritmo de ordenação por inserção (insertion sort) para ordenar um array de itens (a[]) em ordem crescente. O algoritmo funciona construindo uma sequência ordenada à medida que cada elemento é inserido na posição correta. A função utiliza dois loops, um para percorrer os elementos do array e outro para comparar e mover os elementos até encontrar a posição correta. O algoritmo é eficiente para arrays pequenos ou parcialmente ordenados, pois requer um número mínimo de comparações e trocas. Sua complexidade é O(n^2) no pior caso, mas pode ter um desempenho próximo a O(n) no melhor caso.

Void MergeSort: A função merge() implementa a etapa de mesclagem (merge) do algoritmo de ordenação Merge Sort. Ela combina duas sequências ordenadas em uma única sequência ordenada. A função utiliza três índices (i, j, e k) para percorrer os elementos dos sub arrays a serem mesclados. A cada iteração, compara os valores das chaves dos elementos nos índices **i** e **j** e os insere no array auxiliar (aux[]) de forma ordenada. Ao final, os elementos do array auxiliar são copiados de volta para o array original (a[]) na posição correta, completando a etapa de mesclagem.

A função mergesort() é a função principal do algoritmo Merge Sort. Ela realiza a divisão recursiva do array em subarrays menores e chama a função merge() para mesclar esses subarrays. O algoritmo utiliza a estratégia "dividir para conquistar" para ordenar eficientemente o array. A função mergesort() recebe como parâmetros o array de itens (a[]), os índices de início (inicio) e fim (fim), o array auxiliar (aux[]) e os ponteiros para as variáveis de controle de estatísticas. Através da recursão, o algoritmo divide o array em sub arrays até que cada sub array contenha apenas um elemento. Em seguida, realiza a mesclagem dos sub arrays, aplicando a função merge(). Esse processo é repetido até que todos os sub arrays sejam mesclados em uma única sequência ordenada. O Merge Sort possui uma complexidade de tempo média e pior caso de O(n log n), tornando-o uma opção eficiente para ordenar grandes quantidades de dados.

Void Quicksort: O código apresentado implementa o algoritmo de ordenação Quick Sort. O Quicksort é um algoritmo eficiente de ordenação que utiliza a estratégia "dividir para conquistar". A função particionar() desempenha um papel fundamental no Quicksort, sendo responsável por particionar o array em duas partes, posicionando um elemento pivô em sua posição correta. O pivô é selecionado a partir da média dos índices **i** e **j** e os elementos menores são colocados à esquerda do pivô, enquanto os maiores são colocados à direita. Esse processo de particionamento é realizado por meio de dois loops internos que avançam os índices i e j até que se cruzem. Durante o particionamento, o número de comparações e atribuições é contabilizado.

A função quicksort() é a função principal do algoritmo Quicksort. Ela inicializa as variáveis comparações e atribuições com zero e chama a função ordenar() para ordenar o array. A função ordenar() é uma função recursiva que divide o array em partições menores e chama a função particionar() para posicionar o pivô corretamente. Em seguida, a função ordenar() é chamada recursivamente para as partições à esquerda e à direita do pivô. Esse processo é repetido até que todas as partições tenham sido ordenadas. Ao final, o array estará completamente ordenado. A escolha do pivô e a estratégia de dividir e conquistar do Quicksort contribuem para sua eficiência, tornando-o uma opção popular para a ordenação de grandes conjuntos de dados.

Void seleção: A função selecao() implementa o algoritmo de ordenação por seleção (Selection Sort). Esse algoritmo percorre o array várias vezes, procurando o menor elemento em cada iteração e trocando-o com o elemento no início do array ordenado. A função utiliza um loop externo para iterar sobre os elementos não ordenados e um loop interno para encontrar o menor elemento. A cada iteração do loop externo, o menor elemento restante é movido para a posição correta no array ordenado através de uma troca. O número de comparações e atribuições realizadas é contabilizado por meio de ponteiros.

O algoritmo de ordenação por seleção possui uma complexidade de tempo de O(n^2), tornando-o menos eficiente do que outros algoritmos como Quick Sort ou MergeSort. No entanto, ele é simples de entender e implementar, sendo uma escolha adequada para ordenar pequenos conjuntos de dados ou quando a eficiência não é um fator crítico. O algoritmo de seleção é recomendado especialmente quando a quantidade de atribuições é mais custosa do que o número de comparações. Portanto, o Selection Sort é uma opção viável em cenários específicos onde a simplicidade e a otimização de atribuições são prioritárias.

Void shellsort: A função shellsort() implementa o algoritmo de ordenação Shell Sort. A ideia central do Shellsort é comparar e trocar elementos distantes uns dos outros em vez de elementos adjacentes. O algoritmo começa definindo um valor inicial para o intervalo de comparação h. Em seguida, ele reduz gradualmente o valor de h dividindo-o por 3 a cada iteração. O loop externo é executado enquanto h for maior ou igual a 1. Dentro desse loop, ocorre um loop interno que percorre o array a partir do valor h. Para cada elemento, é realizado um processo similar ao algoritmo de inserção, mas comparando e movendo elementos com um intervalo de h posições.

O algoritmo de Shell Sort possui uma complexidade de tempo média de O(n log n), o que o torna mais eficiente do que algoritmos de ordem quadrática, como Bubble Sort ou SelectionSort. No entanto, o ShellSort não é tão eficiente quanto outros algoritmos mais avançados, como Quick

Sort ou Merge Sort. Mesmo assim, o ShellSort ainda é amplamente utilizado e pode ser uma opção interessante quando o desempenho é uma preocupação, mas a simplicidade de implementação é valorizada.

Análise Gráfica dos Algorítimos

Aqui estão as representações gráficas comparativas dos algoritmos: Insertion Sort, Selection Sort, Shellsort, QuickSort, HeapSort e MergeSort.

O eixo Y mostra o tempo médio de execução de cada algoritmo em segundos (calculado a partir de 10 testes para cada algoritmo), enquanto o eixo X representa os tamanhos dos vetores.

Os gráficos estao hospedados num site online para uma melhor consulta.

Entrada com Valores Aleatórios:

A tabela dos tempos em segundos:

Tamanhos	Insertion	Selection	Shellsort	Quicksort	Heapsort	Mergesort
Vetor[20]	0.0000017000	0.0000010000	0.0000007000	0.0000020000	0.0000017000	0.0000015000
Vetor[500]	0.0001492000	0.0234304000	0.0000277000	0.0000420000	0.0000426000	0.0000469000
Vetor[5000]	0.0154890000	0.0235805000	0.0006929000	0.0005120000	0.0005988000	0.0006180000
Vetor[10000]	0.0416403000	0.0879710000	0.0014744000	0.0010400000	0.0011633000	0.0014428000
Vetor[200000]	22.5485036000	33.195642	0.0454625000	0.0264150000	0.0360860000	0.0344565000

Gráfico: https://infogram.com/tempo-de-execucao-aleatorio-1h8n6m3d39vzz4x?live

As tabelas com os valores de atribuições e de comparações segue a seguir,respectivamente:

Atribuições: Tamanhos	Insertion	Selection	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	141.4	45	123	78	157.8	176
Vetor [500]	63031	14973	8711	3519	6292.2	8976
Vetor [5000]	6099689	14973	136299	47178	79608.9	123616
Vetor [10000]	24689275	29973	311607	99783	169231.1	267232
Vetor [200000]	1409051766	599958	10563562	2596023	4249994.9	7075712

Comparações: Tamanhos	Insertion	Selection (n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	94.4	190	39	42	79.3	61
Vetor [500]	62033	12497500	3797	3373	4227	3846
Vetor [5000]	6089691	12497500	66131	48071	58815.3	55235
Vetor [10000]	24669277	49995000	161121	116125	127747.3	120418
Vetor [200000]	1408651768	1474936480	6429270	3098563	3419864.6	3272291

Entrada com Valores Ordenados Decrescente:

A tabela do tempo em segundo:

Tamanhos	Insertion	Selection	Shellsort	Quicksort	Heapsort	Mergesort
Vetor[20]	0.0000010000	0.0000020000	0.0000020000	0.0000020000	0.0000020000	0.0000030000
Vetor[500]	0.0003140000	0.0003110000	0.0000220000	0.0000110000	0.0000340000	0.0000270000
Vetor[5000]	0.0292990000	0.0012080000	0.0000450000	0.0001580000	0.0004720000	0.0003800000
Vetor[10000]	0.1111670000	0.1221020000	0.0004620000	0.0003840000	0.0010040000	0.0008900000
Vetor[200000]	43.2833830000	35.7233710000	0.0147270000	0.0077670000	0.0246420000	0.0200310000

Gráfico: https://infogram.com/tempo-de-execucao-ordenado-decrescente-1hdw2jpwp5pjj2l?live

As tabelas com os valores de atribuições e de comparações seguem a seguir, respectivamente:

Atribuições

Tamanhos	Insertion	Selection (n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	228	30	114	66	144	176
Vetor [500]	125748	750	6718	1512	5922	8976
Vetor [5000]	12507498	1500	14834	16356	75826	123616
Vetor [10000]	50014998	15000	204190	32712	161974	267232
Vetor [200000]	1474536482	300000	5570738	693210	4096866	7075712

Comparações

Tamanhos	Insertion	Selection (n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	190	190	30	38	72	40
Vetor [500]	124750	124750	1804	3006	4006	2216
Vetor [5000]	12497500	499500	3920	46834	56641	29804
Vetor [10000]	49995000	49995000	53704	1036644	123372	64608
Vetor [200000]	1474936480	1474936480	1436446	2937892	3327606	1730048

Entrada com Valores Ordenados Crescentes:

A tabela dos tempos em segundos:

Tamanhos	Insertion	Selection	Shellsort	Quicksort	Heapsort	Mergesort
Vetor[20]	0.0000010000	0.0000010000	0.00000100000	0.0000020000	0.0000020000	0.0000020000
Vetor[500]	0.0000020000	0.0002570000	0.0000150000	0.0000130000	0.0000360000	0.0000290000
Vetor[5000]	0.0000180000	0.0261320000	0.0000240000	0.0001290000	0.0004390000	0.0003480000
Vetor[10000]	0.0000350000	0.1127180000	0.0002840000	0.0002760000	0.0008480000	0.0009010000
Vetor[200000]	0.0000690000	33.9900340000	0.0098280000	0.0084440000	0.0227040000	0.0193210000

Gráfico: https://infogram.com/tempo-de-execucao-valores-ordenados-crescentes-1h7k230z0pymv2x?live

As tabelas com os valores de atribuições e de comparações seguem a seguir, respectivamente:

Atribuições:

Tamanhos	Insertion	Selection (n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	38	0	84	36	166	176
Vetor [500]	998	0	4914	765	6592	8976
Vetor [5000]	9998	0	10914	8856	83462	123616
Vetor [10000]	19998	0	150486	17712	176422	267232
Vetor [200000]	39998	0	4134292	393212	4394750	7075712

Comparações:

Tamanhos	Insertion	Selection(n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	0	190	0	54	80	48
Vetor [500]	0	124750	0	3498	4402	2272
Vetor [5000]	0	12497500	0	51822	61140	32004
Vetor [10000]	0	49995000	0	113631	132263	69008
Vetor [200000]	0	1474936480	0	3137875	3513725	1807808

Vetor aleatório tipo Char:

Tamanhos	Insertion	Selection(n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	0.0000012000	0.0000011000	0.0000010000	0.0000012000	0.0000015000	0.0000016000
Vetor [500]	0.0003454000	0.0002828000	0.0000340000	0.0000295000	0.0004530000	0.0000616000
Vetor [5000]	0.0215257000	0.0216865000	0.0006142000	0.0004011000	0.0007296000	0.0008841000
Vetor [10000]	0.0943977000	0.0802238000	0.0015748000	0.0009805000	0.0018020000	0.0017391000
Vetor [200000]	26.7422562222	32.4350955000	0.0284027000	0.0202008000	0.0358731000	0.0375112000

Segue abaixo as comparações e atribuições feitas nas ordenações:

Comparações tipo Char:

Tamanhos	Insertion	Selection (n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	122	190	40	87	118	61
Vetor [500]	59870	124750	2556	4551	7350	3832
Vetor [5000]	6063555	12497500	29005	69570	105524	54764
Vetor [10000]	24174575	49995000	60014	153327	229910	119366
Vetor [200000]	1023546070	1474936480	1302098	4353699	6252170	3232873

Atribuições tipo Char:

Tamanhos	Insertion	Selection(n^2)	ShellSort	QuickSort	HeapSort	MergeSort
Vetor [20]	160	51	124	44	13	81
Vetor [500]	60868	1443	7360	1814	5713	4488
Vetor [5000]	6073553	14427	99173	17033	72947	61808
Vetor [10000]	24194573	28875	210500	35192	155221	133616
Vetor [200000]	1023946068	576543	5436390	806261	3931534	3537856

Análise dos Algorítimos

Após uma análise minuciosa dos algoritmos, tanto em termos de implementação de código quanto de desempenho, foram identificados os pontos fracos de cada um, especialmente quando lidando com chaves de tamanho considerável.

Inicialmente, o Selection Sort parecia ser uma escolha razoável devido à sua implementação intuitiva. No entanto, ele se mostrou extremamente ineficiente para chaves acima de 50000, devido à sua complexidade $O(n^2)$. Observando as tabelas, fica evidente que o Selection Sort realiza n^2 comparações em qualquer caso. Em entradas aleatórias, esse algoritmo se mostrou o pior em termos de tempo de execução e comparações.

O Insertion Sort, por sua vez, inicialmente foi considerado o segundo pior algoritmo, logo após o Selection Sort. No entanto, ele surpreendeu ao se tornar o mais rápido quando o vetor estava ordenado de acordo com sua implementação. Por outro lado, ele se revelou o pior algoritmo quando o vetor estava ordenado inversamente à implementação.

O Shellsort, uma variação aprimorada do Insertion Sort, possui uma implementação relativamente simples, embora sua ordem de complexidade ainda não seja bem definida. Embora existam conjecturas de que sua complexidade se aproxime de $O(n \log n)$, não é possível descartar a possibilidade de casos desfavoráveis. No entanto, nos casos testados, o Shellsort se mostrou bastante eficiente.

O QuickSort foi considerado o melhor algoritmo entre todos os testados. Ele se destacou como a melhor opção para ordenar qualquer tipo de entrada, exceto em comparação com o Insertion Sort em casos específicos em que o vetor estava ordenado da mesma forma que o Insertion Sort foi implementado. Além disso, o QuickSort teve uma probabilidade mínima de atingir seu pior caso, $O(n^2)$, em nossos testes com entradas muito grandes. Embora sua implementação seja complexa, o QuickSort é amplamente utilizado para ordenação.

O HeapSort é um algoritmo extremamente eficiente para qualquer caso de entrada, com uma complexidade constante de $O(n \log n)$. Embora não tenha se destacado como o melhor algoritmo, exceto em relação ao caso particular do Insertion Sort e casos "normais" do QuickSort, o HeapSort não teve um caso ruim significativo. Além disso, ele não requer memória adicional para a ordenação. A desvantagem do HeapSort está na complexidade de sua implementação, exigindo um conhecimento mais avançado.

O MergeSort, embora tenha uma complexidade de $O(n \log n)$, pareceu ser pior do que os outros algoritmos para casos muito pequenos. Em comparação com os algoritmos mencionados anteriormente, que usam recursão em sua implementação, o MergeSort possui um código mais fácil de ser implementado e um método de fácil compreensão. Para casos com entradas muito grandes, o MergeSort se mostrou tão eficiente quanto o Shellsort.

Conclusão

Durante a realização deste trabalho prático, enfrentamos um desafio significativo relacionado às entradas de tamanho considerável para o Selection Sort e o Insertion Sort. Esses algoritmos possuem uma complexidade $O(n^2)$, o que resultava em um tempo de execução bastante longo para ordenar completamente os conjuntos de dados.

No entanto, essa atividade foi extremamente valiosa para aprofundar nossa compreensão dos algoritmos estudados em sala de aula. Além disso, proporcionou uma oportunidade prática de analisar a complexidade dos algoritmos, confirmando a eficiência dos algoritmos com complexidade $O(nlogn)$ e a falta de eficiência dos algoritmos com complexidade $O(n^2)$ ao lidar com chaves de tamanho considerável. Também pudemos compreender melhor a usabilidade desses algoritmos, uma vez que cada um deles possui aplicações diferentes, dependendo das necessidades do programa.

Em resumo, essa atividade prática nos permitiu aprimorar nossos conhecimentos e habilidades em relação aos algoritmos de ordenação estudados, bem como destacar a importância da escolha correta do algoritmo para uma determinada situação, levando em conta a eficiência e a complexidade temporal.