Trabalho Prático de Algoritmos e Estrutura de Dados II
Ordenação

1 Introdução

Em ambientes de pesquisa que lidam com um grande volume de dados, na maioria das vezes é essencial a implementação de um tipo de estrutura de organização desses dados, para assim facilitar e otimizar a leitura e interpretação desses dados. Tendo isso em mente, a automatização e organização desses dados é uma solução efetiva para sempre manter uma hierarquia que facilite sua manipulação, que será feita toda vez que houver uma alteração desses dados, sendo eles alterando, inserindo ou removendo dados já existentes.

O objetivo principal deste trabalho é modelar e implementar uma estrutura de ordenação de dados onde é usado o Bucket Sort(*bucketsort*[0]) como algoritmo principal. A proposta é dar ao usuário a opção de escolher um tamanho de entrada n_1, logo após inserir n_1 valores para serem ordenados. Também é dada ao usuário a opção onde ele informa qual o tipo de ordenação será usada internamente dentro de cada Bucket(Intervalo). O objetivo secundário é realizar testes de complexidade de tempo e de espaço, avaliando a eficácia dessas combinações de algoritmos de ordenação em relação ao tamanho da sua entrada n_1. Os algoritmos disponibilizados para a ordenação interna dos intervalos são: Quick Sort(*quicksort*[1]), Heap Sort(*heapsort*[2]), Merge Sort(*mergesort*[3]), Inserção(*insertion*[4]) e Seleção(*selection*[5]).

2 Solução do Problema

Levando em consideração que o usuário escolhe o tipo de ordenação interna dos intervalos, todas as entradas foram padronizadas como demonstrado a seguir.

- tipoDeOrdenação(1ª, 2ª, 3ª)
    1. O 1ª parâmetro recebe um vetor de inteiros.
    2. O 2ª parâmetro recebe 0.
    3. O 3ª parâmetro recebe o tamanho do vetor de inteiros.

*Internamente o quicksort[1] e mergesort[3] são os únicos que utilizam o 2ª parâmetro em sua implementação. No heapsort[2], insertion[4] e selection[5] a entrada não é utilizada e foi implementada apenas para a padronização de entradas.

2.1 Ordenação dos Buckets

A avaliação de complexidade de tempo foi feita em termos assintóticos utilizando a notação O. Na avaliação de complexidade de espaço, não é realizada internamente nenhuma alocação e/ou movimentação de memória.

1 - Quick Sort[1]: Sendo a melhor opção para ordenar um grande volume de dados, o quicksort[1] recursivo apresenta uma implementação simples e de fácil entendimento. Suas desvantagens são de se tornar quadrático no pior caso, e não funcionar para elementos repetidos.

Melhor Caso

$C(n) = O(nlog(n))$

Caso Médio

$C(n) = O(nlog(n))$

Pior Caso

$C(n) = O(n^2)$

2 - Heap Sort[2]: Sendo a melhor opção para um grande volume de dados com elementos repetidos, o heapsort[2] apresenta sempre um comportamento de nlog(n) apesar de não ser tão rápido quanto o quicksort[1].

Todos os casos

$C(n) \approx O(nlog(n))$

3 - Merge Sort[3]: Disputando espaço com o heapsort[2], sua vantagem é que pode ser transformado em estável garantindo assim uma ótima performance em aplicações que tenham restrição de tempo para serem executadas. Sua desvantagem é a utilização de memória auxiliar, sendo mais lento na prática que o quicksort[1] e o heapsort[2].

Todos os casos

$C(n) \approx O(nlog(n))$

4 - Inserção[4]: Extremamente útil em um pequeno volume de dados, o insertion[4] e estável e apresenta um comportamento excelente em vetores quase ordenados, sendo grande parte das vezes mais rápido que o quicksort[1]. Em desvantagens apresenta um custo muito grande em movimentações e apresenta comportamento quadrático em muitos casos.

Melhor Caso

$C(n) = O(n)$

Caso Médio

$C(n) \approx O(n^2)$

Pior Caso

$C(n) = O(n^2)$

5 - Seleção[5]: Apresentando uma das menores taxas de movimentação entre elementos, a casos em que pode haver ganho de tempo em estruturas de dados complexas. Sua desvantagem é que sempre apresenta um comportamento quadrático.

Todos os caso

$C(n) = O(n^2)$

2.2 Bucket Sort

bucketsort(1ª, 2ª, 3ª, 4ª, 5ª)

1. O 1ª parâmetro recebe um ponteiro para a função de ordenação a ser utilizada internamente.
2. O 2ª parâmetro recebe um vetor de inteiros.
3. O 3ª parâmetro recebe o tamanho n_1 do vetor de inteiros.
4. O 4ª parâmetro recebe o número de intervalos a serem utilizados.
5. O 5ª parâmetro determina se a função de ordenação tem início com n_1 ou (n_1- 1) elementos.

Complexidade de tempo:

legenda: n = tamanho do vetor e k = número de intervalos

1. O algoritmo se inicia achando o menor e maior elemento do vetor:

   `for (int i = 0; i < n; i++)` n+1 comparações ou O(n)

2. E dividido em intervalos:

   `for (int i = 0; i < k; i++)` k+1 comparaçoes

   `for (int j = 0; j < n; j++)` n+1 comparações

   (n+1)*(k+1) = n*k + n + k + 1 comparações ou O(k*n)

3. E ordenado e reinserido em no vetor inicial

   `for (int i = 0; i < k; i++)` k+1 comparaçoes

   `for (int i = 0; i < b->tam; i++)` inter*b->tam = n

   (k+1) + (n+1) comparações ou O(k+n).

Utilizando a definição soma O(f(n)) + O(g(n)) = O(max(f(n), g(n))) é selecionado o termo de maior crescimento para definir o comportamento assintótico da função chegando a complexidade de O(k*n):

O(n) + O(k*n) + O(k+n) = O(max(n, k*n, k+n)) = O(k*n)

*A complexidade de tempo do bucketsort^0 pode variar de acordo com o algoritmo selecionado para ordenar cada bucket. Para definir a complexidade completa do bucketsort^0, no item 3 é necessário a avaliação da complexidade de tempo de cada algoritmo de ordenação interna, levando em consideração o melhor e pior caso de cada um, assim o atribuindo a soma final e destacando o termo de maior crescimento para definir sua complexidade em termos assintóticos.

Complexidade de espaço:

Na implementação do bucketsort^0 e utilizado a TAD Lista com os seguintes atributos:

```
struct lista{
        Bucket *primeiro;
        Bucket *ultimo;
}
```

```
struct bucket{
        int *buck;
        int tam;
        Bucket *prox;
```

}

Na formulação dos requisitos para a implementação, é mencionada a necessidade da implementação de uma lista encadeada para a separação dos intervalos. O que impacta diretamente a complexidade de espaço da função.

Legenda: L = Lista, B = Bucket, I = inteiro, n = tamanho do vetor e k = número de intervalos

1. Primeiramente é criada uma lista vazia com um bucket vazio.
   L+B
2. Depois são criados k intervalos
   k*B
3. E dentro de cada intervalo é criado um vetor de inteiros proporcionais ao número de valores a serem inseridos em cada um.
   k*B->tam que e igual a n*I

Com a soma final de todas as alocações de memória temos L + k*B + n*I + B, ou seja, temos a complexidade de espaço do bucketsort^{0} de:

1*Lista + k*Bucket + 1 *Bucket + n*int

3 Análise Experimental

Para os testes a seguir foram utilizados os seguintes vetores com tamanho 10 Mil, 100 Mil e 1 Milhão, sendo ele ordenado, aleatório(caso médio) e invertido sem elementos repetidos, e foi utilizado um número de intervalos do Bucket Sorte com os seguintes valores: 5, 10, 20, 100 e 1000.

No primeiro teste foi usado um vetor *v com 10 Mil elementos com o número de intervalos de 5, 10, 20 e 100 e se obteve em resposta o tempo de execução(ms) de:

Ordenação	Elementos	Intervalos	Ordenado	Caso Medio	Invertido
Quicksort	10 Mil	5	0.0191	0.0218	0.0198
Heapsort	10 Mil	5	0.0400	0.0416	0.0380
Mergesort	10 Mil	5	0.0508	0.0547	0.0506
Inserção	10 Mil	5	0.0084	0.4597	0.9157
Seleção	10 Mil	5	0.6466	0.8487	0.9209
Quicksort	10 Mil	10	0.0223	0.0240	0.0234
Heapsort	10 Mil	10	0.0436	0.0396	0.0389
Mergesort	10 Mil	10	0.0515	0.0565	0.0526
Inserção	10 Mil	10	0.0122	0.2357	0.4467
Seleção	10 Mil	10	0.3338	0.3470	0.3685
Quicksort	10 Mil	20	0.0289	0.0332	0.0312
Heapsort	10 Mil	20	0.0431	0.0455	0.0442
Mergesort	10 Mil	20	0.0626	0.0615	0.0653
Inserção	10 Mil	20	0.0190	0.1358	0.2478
Seleção	10 Mil	20	0.1891	0.1830	0.1999
Quicksort	10 Mil	100	0.0834	0.1037	0.1001
Heapsort	10 Mil	100	0.0985	0.1051	0.0992
Mergesort	10 Mil	100	0.1090	0.1266	0.1159
Inserção	10 Mil	100	0.0796	0.1069	0.1420
Seleção	10 Mil	100	0.1074	0.1262	0.1238

Tabela 1(T^{1})

No segundo teste foi usado um vetor *v com 100 Mil elementos com o número de intervalos de 5, 10, 20 e 100 e se obteve em resposta o tempo de execução(ms) de:

Ordenação	Elementos	Intervalos	Ordenado	Caso Medio	Invertido
Quicksort	100 Mil	5	0.1978	0.2507	0.2318
Heapsort	100 Mil	5	0.4911	0.4847	0.4656
Mergesort	100 Mil	5	0.5856	0.6126	0.5897
Inserção	100 Mil	5	0.0817	43.2843	108.4647
Seleção	100 Mil	5	66.0894	65.9354	69.5170
Quicksort	100 Mil	10	0.2289	0.2851	0.2572
Heapsort	100 Mil	10	0.5091	0.5089	0.4997
Mergesort	100 Mil	10	0.5711	0.6660	0.5783
Inserção	100 Mil	10	0.1202	26.7085	43.1927
Seleção	100 Mil	10	32.8723	31.4410	34.9062
Quicksort	100 Mil	20	0.2931	0.3672	0.3340
Heapsort	100 Mil	20	0.5659	0.5842	0.5666
Mergesort	100 Mil	20	0.6259	0.7176	0.6520
Inserção	100 Mil	20	0.1910	11.1040	22.2143
Seleção	100 Mil	20	15.8669	16.7234	19.9063
Quicksort	100 Mil	100	0.8369	0.9979	0.9360
Heapsort	100 Mil	100	1.0697	1.1439	1.0771
Mergesort	100 Mil	100	1.1439	1.3132	1.2319
Inserção	100 Mil	100	0.7528	3.8267	6.4668
Seleção	100 Mil	100	3.9065	4.0362	4.3264

Tabela 2(T^2)

No Terceiro teste foi usado um vetor *v com 1 Milhão elementos com o número de intervalos de 100 e 1000 e se obteve em resposta o tempo de execução(ms) de:

Ordenação	Elementos	Intervalos	Ordenado	Caso Medio	Invertido
Quicksort	1 Milhão	100	8.5732	10.3155	9.6012
Heapsort	1 Milhão	100	11.2768	12.3734	11.5797
Mergesort	1 Milhão	100	12.2042	13.9028	12.7479
Inserção	1 Milhão	100	7.3465	226.7129	450.6379
Seleção	1 Milhão	100	319.3166	337.7789	402.5327
Quicksort	1 Milhão	1000	69.9981	82.4256	78.8809
Heapsort	1 Milhão	1000	71.6498	83.4915	79.8809
Mergesort	1 Milhão	1000	72.4386	84.9517	81.0324
Inserção	1 Milhão	1000	68.5315	102.9689	120.3378
Seleção	1 Milhão	1000	111.7184	113.1880	111.5381

Tabela 3(T^3)

Com a análise das tabelas T^1, T^2 e T^3, tem-se a comprovação das análises assintóticas dos algoritimos quicksort[1], heapsort[2], mergesort[3], insertion[4] e selection[5], em todos os testes eles apresentam o crescimento esperado, variando apenas quanto a o aumento de elementos e número de intervalos.

O quicksorte[1] apresentou o melhor tempo de ordenação tanto com o vetor aleatório quanto o invertido em todos os testes, e perdeu apenas para o insertion[4], mas apenas nos testes com o vetor ordenado.

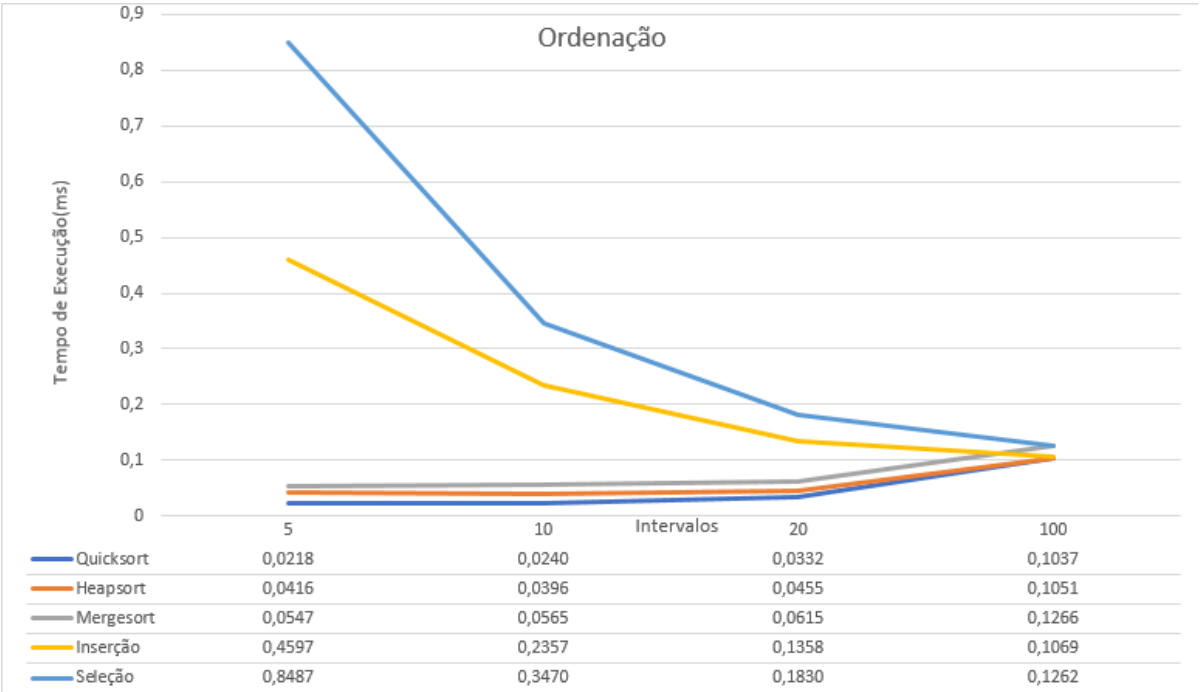
O heapsort[2] apresentou o comportamento esperado, e perdeu em todos os testes para o quicksort[1], mas tendo um ganho de tempo em relação ao mergesorte[3].

O mergesort[3] manteve tempos muito parecidos em todos os casos, mas estranhamente perdeu em estabilidade de tempo para o heapsort[2] em alguns casos. O que leva a duas possíveis conclusões: 1ª a versão implementada detém algum erro de lógica que a manteve um pouco instável ou 2ª é mais provável sendo a variação causada devido a diversos fatores como de cachês ou variações do sistema.

O insertion[4] teve o melhor tempo com o vetor ordenado em todos os testes, e teve ganho em relação ao selection[5] em todos os casos médios, mas foi o pior em todos os testes com o vetor invetido.

O selection[5] teve o pior desempenho de todos os algoritmos de ordenação, tendo ganho apenas em relação ao insertion[4] quando o vetor aparece invertido. Em todos os testes não houve nenhum que se enquadra como sendo situação em que seria útil.

A seguir é mostrado como o uso dos intervalos impactou no tempo de execução do programa. Os dados utilizados para criação G[1] foi um vetor com 10 mil elementos aleatórios, com o número de intervalos de 5, 10, 20 e 100.



	5	10	20	100
Quicksort	0,0218	0,0240	0,0332	0,1037
Heapsort	0,0416	0,0396	0,0455	0,1051
Mergesort	0,0547	0,0565	0,0615	0,1266
Inserção	0,4597	0,2357	0,1358	0,1069
Seleção	0,8487	0,3470	0,1830	0,1262

Gráfico 1(G[1])

A escolha do uso de um vetor com elementos aleatórios foi para simular um caso onde todos os algoritmos estejam em seu caso médio. Percebe-se que há um grande ganho de tempo em algoritmos quadráticos sendo eles o insertion[4] e o selection[5], a medida que os números de intervalos aumentam. Já os outros algoritmos sendo eles quicksort[1], heapsort[2] e mergesort[3] apresentam uma perda de velocidade à medida que o número de intervalos aumentam. É importante observar que em um vetor com 10 mil elementos aleatórios não repetidos, a partir de 100 intervalos os algoritmos quadráticos mostram uma melhor performance em relação aos outros.

4 Conclusão

Em todos os testes, os algoritmos tiveram o comportamento esperado. Levando em conta a implementação dos algoritmos de de ordenação interna, a maior dificuldade encontrada usando como base os algoritmos apresentados em aula, foi a adaptação do heapsort[2] para rodar em um vetor que tem início no índice 0. A segunda dificuldade foi a implementação e adaptação da lista encadeada para ser usada em conjunto com o bucketsort[0], que foi o que demandou o maior tempo de implementação. Mas a maior dificuldade foi a lógica da implementação de separação do vetor em intervalos, que após vários lógicas de implementação chegou-se a complexidade de O(k*n) onde k é o número de intervalos e n e o número de elementos do vetor, o que se tornou o termo de maior crescimento da função, mas acredito que dê para melhorar.

Após todos os testes possíveis, conclui-se que o trabalho apresenta uma aplicação funcional e um comportamento estável com uma implementação compacta e de fácil entendimento.

5 Bibliografia

1. Slides da disciplina de algoritmo e estrutura de dados II.
2. CORMEN, Thomas H. Algoritmos teoria e prática. 3 edição. São Paulo GEN LTC 2012
3. CORMEN, Thomas H. Desmistificando algoritmos. Rio de Janeiro GEN LTC 2013