Universidade Federal Fluminense
Instituto de Computação
Nomes: Fernando Costa Rodrigues e Lucas Alexandre Siqueira
Análise e Projeto de Algoritmos – 2018.1

Relatório T2: comparativo entre algoritmos de ordenação.

1- Ambiente utilizado:

Antes de iniciarmos as análises é importante apresentarmos o ambiente utilizado para a execução dos testes. A máquina utilizada tinha como configurações:

- **CPU**: Intel Core I5 5200U 2.20GHz
- **RAM**: 6gb DDR3L 1600MHz SDRAM
- **OS**: Ubuntu 16.04 - 64 bits

2 - Análise das execuções:

Primeiramente, para analisarmos todos os 11 casos de entrada, foram implementados os três algoritmos de ordenação: *selection*, *insertion* e *merge*. Na tabela abaixo, podemos observar características como:

- **Teste**: número de identificação do teste utilizado como entrada.
- **Tamanho**: número de elementos do vetor a ser ordenado.
- **Tempo de execução**: esta coluna apresenta o tempo de execução dos testes, em milissegundos, para cada algoritmo implementado.
- **Ordenado** (S/N): apresenta a letra **S** caso a entrada do teste esteja ordenada de forma crescente e **N** caso o contrário.

Teste	Tamanho	Ordenado?	Tempo de execução		
			Selection	Insertion	Merge
1	100	N	0.000078	0.000067	0.000041
2	1000	N	0.003441	0.003655	0.00031
3	5000	N	0.043203	0.048841	0.00083
4	10000	N	0.163353	0.190803	0.001645
5	50000	N	3.967801	4.879773	0.009641
6	50000	N	3.899937	3.28063	0.007338
7	50000	S	3.983710	0.000154	0.000485
8	50000	N	4.291243	9.86772	0.005349
9	100000	N	16.579813	22.145268	0.02242
10	500000	N	413.265147	503.675345	0.130753
11	1000000	N	1612.02476	2099.944958	0.237897
Total	1816100	-	2054.238776	2644.037214	0.409371

*Tabela 01: análise temporal dos testes

2.1 - Corretude:

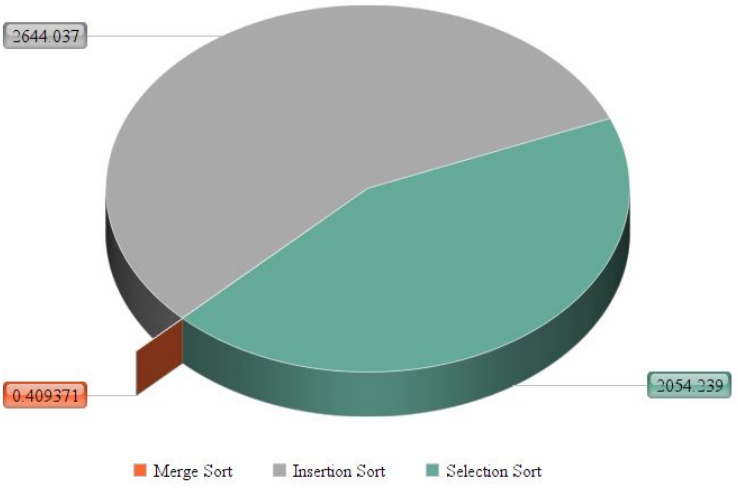
De forma geral podemos afirmar que todos os algoritmos testados obtiveram êxito em todos os casos de teste, ou seja, todos resolveram o problema proposto. Para chegar nesta conclusão foi implementada uma função **estaOrdenado(),** que retornava 1 no caso do vetor esteja ordenado e 0 caso o contrário. Por fim, com utilização da mesma após todas as chamadas às funções de ordenação, obtivemos resultado positivo para todas as entradas.

2.2 - Desempenho:

2.2.1 - Tempo de execução

Agora, analisando as informações e a implementação, podemos deduzir alguns possíveis cenários:

- Cenário 1: Em geral o algoritmo *merge sort* obteve um desempenho superior em relação aos demais algoritmos ao levarmos em consideração os seus tempos de execução, como podemos observar no Gráfico 1, que mostra a soma dos tempos de execução de cada algoritmo em todos os testes dados.



- Cenário 2: Em casos com entradas muito grandes e/ou vetores longe do estado ordenado (ou ordenado de forma inversa), caímos nos piores casos do *insertion sort* (chegando próximo ou igual a complexidade do *selection de* $O(n^2)$). Logo, o que fará a diferença será o custo das operações, onde o *selection* terá uma leve vantagem (vide Gráfico 1), visto que a instrução de comparação tem menor custo em relação a troca de lugar entre dois elementos (operações em memória).

- Cenário 3: Quando tivermos vetores parcialmente ordenados, ou perto deste estado, estaremos tendendo ao melhor caso do *insertion sort*, visto que o mesmo trabalha movendo os elementos até sua posição correta no vetor ordenado. Assim, mesmo tendo instruções de um maior custo, o número de execuções das mesmas será inferior em relação ao *selection sort*, o que fará com que tenhamos um desempenho superior ao utilizarmos a ordenação por inserção.

- Cenário 4: Casos onde o vetor a ser ordenado já se encontram na ordem desejada irão configurar o melhor caso do *insertion sort*, tendo assim uma execução em tempo linear. Isso acontecerá pois o mesmo não precisará efetuar as trocas entre elementos, e só passará pelo vetor comparando valores que já estão na sua posição em ordem. Com isso temos que o mesmo sai em vantagem em relação ao *merge sort*, que em melhor caso roda em O(n log n), e ao selection sort, $O(n^2)$.

2.2.2 - Uso de memória:

O uso de memória pelos algoritmos depende muito das suas implementações, de forma que alguns tornam-se menos eficientes que outros nesse aspecto.

Considerando esta informação, na implementação do merge sort, utilizamos a estratégia de manipular um vetor auxiliar, que é criado em uma única chamada do merge sort, e fazendo a manipulação por índices e ponteiros. Isso implica de uma forma positiva neste algoritmo, visto que em implementações mais usuais cópias ou alocações de memória são feitas em funções recursivas, o que torna o uso de memória excessivo. Desta forma, mesmo com uma chamada recursiva teremos apenas as cópias das variáveis usadas nas chamadas recursivas e um vetor auxiliar de N elementos

como gasto extra de memória, o que torna o uso da mesma não excessivo, porém superior aos demais testados.

Em relação aos outros dois algoritmos utilizados não teremos problemas com relação a memória, considerando que operam *in-place,* ou seja, utilizando apenas a memória necessária para o vetor que guarda os números já que as operações são feitas dentro do mesmo.

2.3 - Escolhendo o melhor algoritmo caso a caso:

Teste 01: Vetor não ordenado, contendo apenas 100 elementos. Nesse caso, o Merge Sort apresentou o menor tempo de execução e apesar de usar mais memória que os demais (devido ao vetor auxiliar), isso não representou um problema dado o tamanho do vetor. Já o Selection Sort e Insertion Sort apresentaram tempo de execução maiores e próximos entre si. Apesar do Selection executar mais passos o Insertion ($O(n^2)$), o tempo ficou equilibrado devido ao fato das instruções do Insertion (manipular memória, trocando várias vezes elementos no vetor) ser mais custosas que a do Selection (faz mais uso de comparações). Nesse caso, optaria pelo Merge, Insertion e Selection nesta ordem. (Cenário 1)

Teste 02: Vetor não ordenado, com 1000 elementos. O Merge Sort foi consideravelmente superior que os demais algoritmos. Além disso, podemos ver que o Selection Sort ($O(n^2)$ no melhor e pior caso), mesmo executando um maior número de passos se comparado ao Insertion Sort, foi um pouco mais eficiente já que suas instruções são menos custosas. Nesse caso, optaria pelo Merge, Selection e Insertion nessa ordem. (Cenários 1 e 2).

Teste 03: Vetor não ordenado, com 5000 elementos. Teste extremamente similar ao anterior, contudo com um vetor 5x maior. Apesar disso, seu tamanho ainda não é significativamente grande para o uso de memória representar um problema (em nosso contexto). Por isso, optaria novamente pelo Merge, Selection e Insertion nessa ordem. (Cenários 1 e 2).

Teste 04: Vetor não ordenado, com 10000 elementos. Novamente, similar aos dois casos anteriores, mas agora o Selection Sort teve uma eficiência significativamente superior se comparado ao Insertion Sort. Como o Selection utiliza mais de comparações e pouca manipulação na memória (troca de elementos), ele se saiu superior ao Insertion Sort que faz diversas trocas de elementos no vetor em seu processo. Optaria pelo Merge, Selection e Insertion. (Cenários 1 e 2)

Teste 05: Vetor não ordenado, com 50000 elementos. Analisando seus resultados, podemos observar que, se um vetor não está parcialmente ordenado, o Insertion Sort consegue ser menos escalável que o Selection Sort (na questão da eficiência temporal). O Merge Sort ainda se mostra extremamente superior aos demais, sendo muito mais escalável. Contudo, como tudo tem seu preço, essa eficiência custa mais memórias que os demais algoritmos, dado que precisamos utilizar um vetor auxiliar e cópias de variáveis no processo. Ainda assim, esse uso superior de memória não representa um problema dado a quantidade de memória que os computadores atuais possuem (dado o contexto onde os testes foram aplicados), por isso optaria novamente pelo Merge Sort, Selection e Insertion, nesta ordem. (Cenários 1 e 2)

Teste 06: Vetor não ordenado, com 50000 elementos. Agora, o Insertion Sort foi consideravelmente mais eficiente que o Selection Sort, divertindo dos resultados de testes anteriores. Analisando o motivo, vemos que um único elemento aparece cerca de 27.000 vezes no vetor, caindo em um bom caso do Insertion Sort, onde o mesmo não terá que fazer muitas trocas de elementos na ordenação (compensando o fato de suas instruções serem mais pesadas). Ainda assim, o Merge Sort novamente foi bem superior, e pelos mesmos motivos já citados anteriormente, optaria por ele, insertion e selection, nesta ordem. (Cenário 1 e 3)

Obs: nesse vetor, temos que o elemento “9911203675” aparece cerca de 27.000 vezes como dito acima. Contudo, vemos que ele extrapola o tamanho de um “int” em C, causando um Overflow que altera seu valor real. Para solucionar esse problema, poderíamos optar pelo “long int” para esse caso, mas como o overflow acontece da mesma forma todas as vezes que o elemento aparece, preferimos manter a variável como um “int” já que isso não altera a real intenção do cenário, que é o de vermos como os algoritmos se comportariam quando temos vários elementos repetidos no vetor.

Teste 07: Vetor ordenado, com 50000 elementos. Nesse caso, o Insertion Sort cai em seu melhor caso, onde ele só percorre o vetor uma vez e não precisa efetuar nenhuma troca. Já o Merge Sort e o Selection Sort tem seu melhor caso rodando em O(n log n) e $O(n^2)$, respectivamente. Por isso, nesse caso optaria pelo Insertion, Merge e Selection nesta ordem. (Cenários 4)

Teste 08: Vetor com 50000 elementos, ordenado de forma inversa (por isso o classificamos como não ordenado na tabela). Nesse caso, vemos que o Insertion Sort foi muito inferior ao Selection Sort (em uma proporção maior do que vimos anteriormente). Isso se dá porque o caso onde o vetor está ordenado de forma inversa cai no pior caso do algoritmo ($O(n^2)$ com instruções custosas). Por isso essa diferença tão grande de desempenho. Além disso, o Merge Sort se sai bem superior aos dois. Nesse caso, escolheria o Merge, Selection e Insertion, nessa ordem. (Cenário 1 e 2).

Teste 09: Vetor não ordenado, com 100000 elementos. Nesse teste podemos ver a absoluta superioridade do Merge Sort no quesito escalabilidade. Agora, como o vetor tem um número de elementos considerável, a questão da memória pode vir a se tornar um problema. Caso eu esteja em uma situação onde tenha uma quantidade bem limitada de memória disponível, optaria pelo Selection Sort, que mesmo executando mais passos que o Insertion Sort, é consideravelmente mais rápido para esse caso (já explicado em teste anteriores). Contudo, como no meu caso possuo um ambiente onde um vetor auxiliar de 100000 elementos na memória principal não representa um problema, optaria novamente pelo Merge, tendo o selection e insertion como segunda e terceira opção, respectivamente. (Cenário 1 e 2).

Teste 10: Vetor não ordenado, com 500000 elementos. Similar ao teste anterior, mas agora com um vetor ainda maior. Vale destacar os mesmos pontos citados anteriormente. Como a memória não representa um problema para minha situação atual, optaria pelo Merge, Selection e Insertion, respectivamente. (Cenário 1 e 2).

Teste 11: Vetor não ordenado, com 1000000. Similar aos dois últimos testes, mas agora com um vetor de 1mi de elementos. Ainda assim, optaria pelo Merge Sort caso a memória não representasse um problema ou o Selection Sort caso eu tivesse uma quantidade bem limitada de memória (vetor não está parcialmente ordenado, fazendo o Selection ter vantagem no quesito de tempo de execução se comparado ao Insertion). (Cenário 1 e 2)

3 - Conclusão:

O problema da ordenação é um dos mais discutidos na área da computação, isso porque existem diversos algoritmos que resolvem o problema e operam de formas completamente diferentes. A partir disso, temos que levar bastante em consideração certas características sobre os dados a serem ordenados antes de escolhermos o algoritmo "certo" para a tarefa, e algumas destas características afetam das seguintes formas na eficiência do mesmo:

- Tamanho do vetor: o tamanho do vetor claramente vai sempre ser levado em consideração para o tempo de execução final visto que cada algoritmo terá uma forma de operar, sendo alguns mais escaláveis que outros.

- Distribuição dos valores no vetor: vimos que em vetores com valores distribuídos de forma que os mesmos estivessem mais próximos de estarem ordenados podem ter um desempenho superior dependendo do funcionamento do algoritmo. Exemplo disso pode ser o melhor caso do Insertion Sort, que opera em tempo linear com o vetor já ordenado como foi visto acima.

- Custo das operações utilizadas: algumas operações são mais custosas em relação a outras, de forma que, em certos casos, mesmo algoritmos que efetuam menos instruções no geral, tem um desempenho inferior devido a essa pequena diferença que em grande escala pode ter bastante influência. Isso pôde ser visto com as operações de comparação e troca de posições dos algoritmos selection e insertion sort, visto que ambas lêem a memória, mas após isso uma apenas compara estes valores enquanto a outra necessita escrevê-los de volta.

- Utilização de memória: dependendo dos métodos utilizados na implementação do algoritmo o uso extra de memória pode se tornar um diferencial negativo. Algoritmos que envolvem chamadas recursivas e estruturas auxiliares efetuarão cópias de valores na memória, o que pode ser um grande problema em entradas muito grandes. Em contrapartida, temos algoritmos que trabalham

em um escopo bem definido sem necessitar de memória “extra”, os que são chamados de *in-place*, necessitando apenas o espaço do vetor principal para trabalhar.

- Ordenação Estável: se apenas ter um vetor ordenado não bastar e você queira manter a ordem de registro dos elementos, será necessário optar por alguns algoritmos (implementações) que mantenham essa integridade (como é o caso de nossa implementação do Insertion Sort, por exemplo).

Por fim, podemos dizer que, generalizadamente, nenhum dos algoritmos testados é superior ao outro. Isso porque eficiência, tanto em relação a tempo quanto a espaço, e corretude (não neste caso, porém em casos como algoritmos lineares podemos ter problemas) serão sempre diretamente dependentes das características dos casos onde os mesmos serão utilizados, sendo sempre recomendável efetuar uma análise prévia do cenário para que os resultados sejam encontrados corretamente e da forma mais eficiente o possível.