UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA
CIÊNCIA DA COMPUTAÇÃO
ESTRUTURAS DE DADOS II

ORDENAÇÃO DE DADOS

Aluno: Daniel Favoreto

Professora: Mariella Berger

Vitória
2 de Dezembro de 2015

Introdução

Este trabalho tem como finalidade a implementação e comparação de algoritmos de ordenação diversos.

Existem inúmeros algoritmos de ordenação de vetores, sendo desde aqueles que utilizam métodos simples como bubble sort,selection sort e insertion sort até os mais sofisticados como por exemplo: merge sort, heap sort e radix sort.

Foram implementados 13 algoritmos de ordenação, sendo a exceção o radix binário.

Os algoritmos foram implementados utilizando a linguagem C e compilados no GCC.

As seções seguintes mostram primeiramente uma visão a respeito da implementação dos algoritmos e em seguida é dado um comparativo dos algoritmos.

Implementação

As estruturas utilizadas assim como a assinatura das funções tanto para a geração dos números, quanto para os algoritmos estão nos arquivos **utility.h** e **gera.h**.

Houve a necessidade da criação da estrutura `MaxHeap` para o algoritmo de heap sort. A estrutura se encontra disponível no arquivo **utility.h**.

Importe ressaltar que todas as funções recebem como parâmetros: o vetor a ser ordenado e o tamanho do vetor. As funções `quicksortcentral`,`quicksortrandom`,`quicksortmediana3` e `quicksortprimeiro` além dos parâmetros já citados, recebem também o índice da posição inicial do vetor e o índice da posição final do vetor a ser ordenado.

A função `mergeSort` utiliza uma função auxiliar `merge` que tem o objetivo de fazer um "merge"dos vetores L e R no vetor a ser ordenado A. Essa função além de receber esse vetores como parâmetros, utiliza o tamanho de cada vetor, sendo "leftcount"o tamanho do vetor L (sub vetor à esquerda de A) e "rightcount"o tamanho do vetor R (sub vetor à direita de A).

Para o algoritmo de heap, foi escolhido utilizar uma MaxHeap. A função `heap` utiliza uma função auxiliar `createAndBuildHeap` que tem o objetivo de criar e alocar uma MaxHeap, além de utilizar a função auxiliar `maxHeapify` que tem o objetivo de expandir a heap de cima para baixo. Uma função auxiliar `swap` foi utilizada para trocar dois números.

As funções `quicksortcentral`,`quicksortrandom`,`quicksortmediana3` e `quicksortprimeiro` foram alteradas para poderem se adequar ao pivot intrínseco de cada uma. A `quicksortcentral` exigiu que calculasse a posição central do vetor e então o pivot fosse escolhido dada essa posição. A `quicksortrandom` utilizou a função rand para aleatoriamente escolher uma posição de 0 até n-1, sendo n-1 o tamanho do vetor menos 1, para então escolher o pivot dada essa posição. A função `quicksortmediana3` utilizou uma comparação do conteúdo do vetor na primeira posição, na posição do meio e na posição final, uma vez que é necessário a escolha do elemento intermediário dentre essas 3 posições para a escolha do pivot. Já a função `quicksortprimeiro` escolhe o pivot apenas capturando o primeiro elemento do vetor a cada partição.

Como a escolha da implementação do rank sort e radix sort foi pela não comparação de chaves, algumas mudanças foram feitas. A função `rank sort` teve de ser implementada pelo próprio aluno para que não houvesse comparação por chaves e utilizasse o bucket sort para ordenação dos buckets. Para tanto, por recomendação da professora, optou-se por utilizar `calloc` para 1 milhão e uma posições de memória a fim de maiores problemas de execução. Já o `radix sort`, para evitar a comparação por chaves, setou-se a variável "maior"para 1 milhão.

Na seção de bibliografia, é dado a referência para cada algoritmo extraído da web. Embora quase todos os algoritmos possuem referências da web, foram necessárias algumas modificações para que pudessem se adequar ao comportamento exigido na especificação do trabalho.

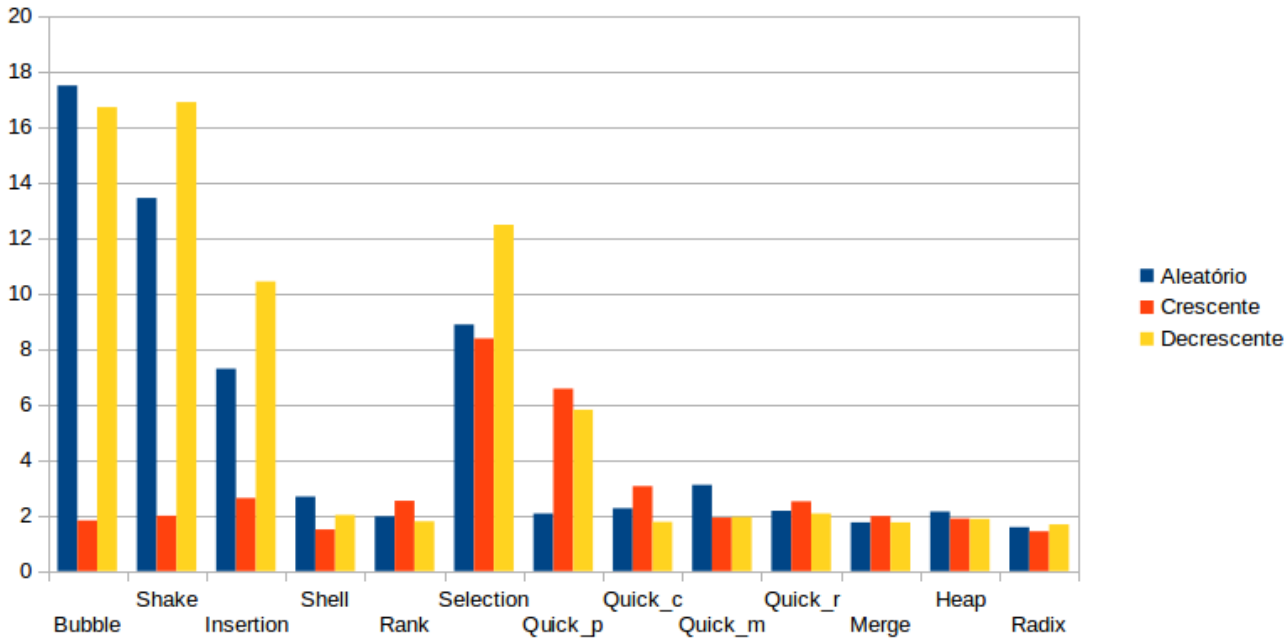
No arquivo `gera.c` está a implementação das 3 diferentes formas de gerar números: crescente, aleatório e decrescente. As 3 formas de implementação foram parecidas, todas utilizando-se a função `rand` para gerar números aleatórios.

A principal diferença é devido a crescente e decrescente gerarem números aleatórios entre 0 e o tamanho escolhido e depois ordenarem os números em um vetor, de forma crescente ou decrescente respectivamente.

Análise

Foram elaborados vários testes com as implementações dos algoritmos, mas para que houvesse uma melhor análise, utilizou-se entradas aleatórias,crescentes e decrescentes para cada um dos mesmos. Aqui serão mostradas entradas para 50 mil números, 100 mil números e 500 mil números. É Importante salientar que a partir de 175 mil números, a função `quicksortprimeiro` apresentou segmentation fault, por isso, não aparece seus resultados na entrada de 500 mil.

A seguir tem-se resultados da entrada para 50 mil números. À esquerda os valores apresentados de 0 a 20 estão em segundos e utilizou-se abreviações para as variações do QuickSort.



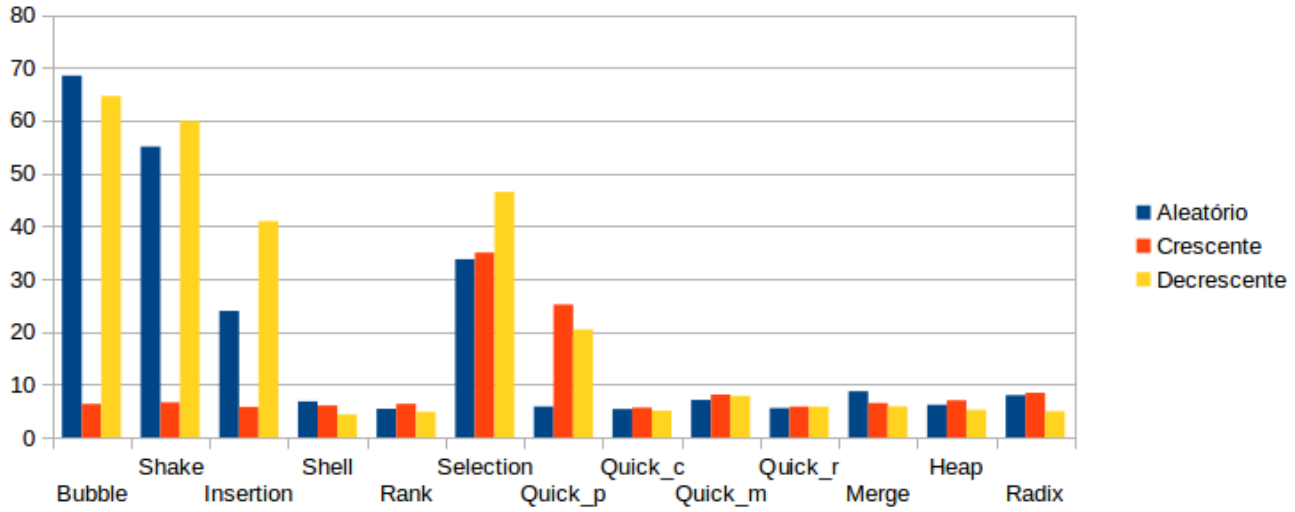
Pela análise do gráfico pode-se observar que o `radix sort` obteve o melhor desempenho para a entrada aleatória e decrescente, sendo superado por uma pequena margem pelo algoritmo `shell sort` se tratando da entrada crescente. Entretanto, se levar em consideração a variação de desempenho entre os tipos de entrada, os algoritmos `radix` ,`heap sort` e `merge sort` possuem poucas diferenças de desempenho entre os tipos de entradas, sendo que o `heap sort` leva alguns segundos a mais se tratando da entrada aleatória enquanto o `merge sort` leva alguns segundos a mais na entrada crescente.

Todas as 4 variações do `quicksort` com exceção do `quicksort primeiro` obtiveram desempenho abaixo ou igual aos 3 segundos para os 3 tipos de entrada. O `quicksort primeiro` curiosamente obteve o melhor desempenho dentre as 4 variações para a entrada aleatória, embora tenha apresentado um desempenho ruim em relação aos demais nas entradas crescente e decrescente. `Rank sort` e `shell sort` figuraram um desempenho muito bom nos 3 tipos de entradas, entretanto, o `shell sort` obteve uma leve variação entre a entrada aleatória e crescente.

Os 4 piores desempenhos são formados por bubble sort,shake sort,insertion sort e selection sort, sendo o `bubble` o pior de todos para a entrada aleatória e `shake sort` o pior para entrada decrescente seguido de perto por `bubble sort`. Para a entrada crescente, o `selection sort` obteve o pior desempenho. Curiosamente, com exceção de `selection sort` e `quicksort primeiro`, todos os outros algoritmos obtiveram resultados bons para a entrada crescente.

Observa-se que a diferença entre o pior desempenho (`bubble sort`) e o melhor (`radix sort`) chega a ser uma diferença da ordem de 10 vezes. Interessante notar que dentre os algoritmos de ordem quadrática, o `shell sort` foi mais eficiente em relação aos demais.

A seguir tem-se a entrada de 100 mil números. À esquerda os valores apresentados de 0 a 80 estão em segundos.



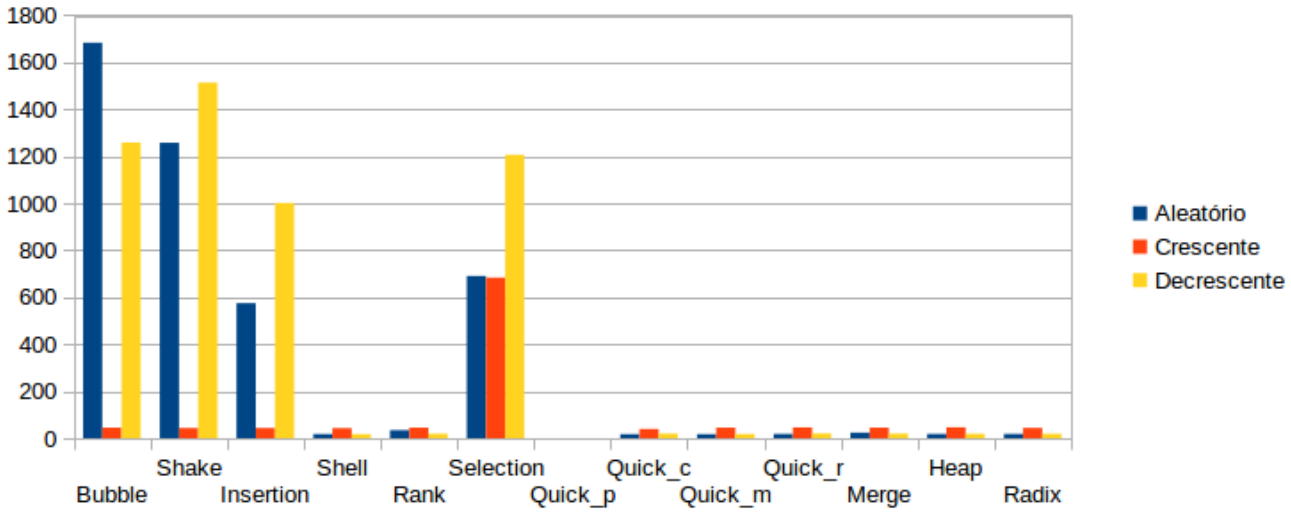
Como pode-se analisar, dobrou-se o tamanho da entrada para 100 mil e temos um resultado um pouco diferente do anterior. Dessa vez o quicksort central obteve o melhor desempenho para as entradas aleatórias ao lado do `rank sort`, seguidos de perto por `quicksort random` e `quicksort primeiro`. Já `bubble sort, shake sort, selection sort e insertion sort` obtiveram os piores desempenhos para este tipo de entrada. Destaque para `bubble sort` que alcançou um tempo na casa dos minutos.

Se tratando da entrada crescente, todos os algoritmos obtiveram bom ou ótimo desempenho com exceção do `selection sort` e `quicksort primeiro`. Para a entrada decrescente, `shell sort` foi o mais eficiente, seguido de perto por `rank sort`, `quicksort central` e `radix sort`. Em contrapartida, os piores desempenhos para este tipo de entrada ficaram com `bubble sort,shake sort, selection sort, insertion sort e quicksort primeiro`. `Bubble sort e shake sort` alcançaram tempo de 1 minuto ou mais para este tipo de entrada.

Pela análise do gráfico pode-se concluir que o `quicksort central` obteve as menores variações de desempenho entre os algoritmos testados assim como `quicksort random`, mesmo ambos mantendo um ótimo desempenho de ordenação. Também é possível observar a grande volatilidade de desempenho para diferentes tipos de entrada, principalmente entre os algoritmos de complexidade quadrática como `bubble sort, shake sort e insertion`, uma vez que os mesmos obtiveram um bom desempenho para as entradas crescentes, embora a aleatória e decrescente fossem muito demoradas.

A diferença entre o pior desempenho (`bubble sort`) e um dos melhores desempenhos (`quicksort central`) chega a ser 13 vezes maior.

A seguir tem-se a entrada de 500 mil números. Sendo importante ressaltar que o quicksort primeiro obteve segmentation fault acima de 175 mil números, por isso, optou-se por omitir seu desempenho. À esquerda os valores apresentados de 0 a 1800 estão em segundos.



Pela análise do gráfico pode-se observar primeiramente que o único algoritmo a obter um desempenho considerado insatisfatório para todos os 3 tipos de entrada é o `selection sort` e tem-se que para a entrada crescente, todos os demais algoritmos obtiveram desempenho abaixo de 1 minuto, sendo o destaque para o `quicksort central` que obteve o menor tempo de execução, seguido de perto por `shell sort, shake sort e insertion sort`. Entretanto, é válido destacar que a diferença para os demais algoritmos foi da ordem de no máximo 3 segundos.

Se tratando da entrada aleatória, o algoritmo `quicksort central` obteve novamente o melhor desempenho dentre todos os algoritmos, sendo que os algoritmos `quicksort mediana3`,`shell sort`,`quicksort random`,`radix sort` e `heap sort` também obtiveram um desempenho semelhante em relação ao `quicksort central` para a entrada aleatória. Em contrapartida, `bubble sort` levou cerca de 28 minutos para completar sua execução enquanto `shake sort` alcançou 20 minutos. `Selection sort` e `insertion sort` obtiveram entre 10 minutos e 11 minutos de execução para a entrada aleatória.

Para o tipo de entrada decrescente, `shake sort` obteve o pior desempenho alcançando 25 minutos de execução. `Bubble sort` e `selection sort` alcançaram em torno de 20 minutos de execução, seguidos de `insertion sort` que obteve em torno de 16 minutos de execução para ordenar os 500 mil elementos, exceto por estes algoritmos, os demais obtiveram um desempenho bom ou ótimo. Os melhores desempenhos ficaram com `shell sort` e `quicksort mediana3`. O `quicksort central` dessa vez obteve cerca de 2 segundos a mais em relação aos melhores algoritmos nos testes para a entrada decrescente.

O `quicksort primeiro` ficou sem o respectivo resultado em razão de apresentar segmentation fault para entradas acima de 175 mil números. Suspeita-se que o pior caso do `quicksort primeiro` tenha levado à esta situação, somado à um excessivo empilhamento do programa, já que é recursivo.

De acordo com estudos e análises de testes para variadas entradas, pode-se apresentar alguns fatos observados a respeito dos algoritmos aqui estudados e implementados:

- Para entradas aleatórias de aproximadamente até 50 mil números, `radix sort` obteve o melhor desempenho em todos os testes.
- Para entradas aleatórias de qualquer grandeza, `bubble sort` se mostrou o pior dos algoritmos.
- Para entradas decrescentes acima de 100 mil números, `shake sort` obteve o pior desempenho.

- `Selection sort ,insertion sort e shake sort` possuem o pior desempenho para entradas decrescentes independentemente do tamanho.
- Das 4 variações de `quicksort` implementadas, tratando-se de entradas crescentes e decrescentes a variação com a escolha do pivot sendo o primeiro elemento apresentou pior desempenho.
- Com exceção de `selection sort` e `quicksort primeiro`, todos os demais algoritmos executam em um tempo semelhante para entradas previamente ordenadas.
- `Quicksort random` e `heap sort` apresentaram menor sensibilidade ao tipo de entrada.

Conclusão

Após feita a análise de cada algoritmo, pode-se apresentar algumas conclusões. Primeiramente, se houvesse a necessidade de opção por algum algoritmo independentemente do tamanho da entrada e o seu tipo, a escolha do `shell sort` seria muito bem vinda, uma vez que tanto para as entradas de 50 mil, 100 mil e 500 mil o algoritmo esteve dentre os melhores, ora sendo entrada crescente, ora decrescente ou aleatória. Se houvesse o desconhecimento do tipo de entrada, `quicksort random e heap sort` seriam ótimas escolhas, já que mostraram desempenhos ótimos ou bons independentemente do tipo de entrada.

Definitivamente, qualquer uma das 3 opções: `bubble sort, shake sort ou insertion sort` seriam escolhas ruins de ordenação devido ao desempenho muito aquém em relação aos demais se tratando da entrada aleatória ou decrescente. Se tratando de entradas previamente ordenadas, todos os algoritmos com exceção do `selection sort` obteriam boa ou ótima performance. Importante notar que a pior escolha para qualquer tipo de entrada e qualquer tamanho seria o `selection sort`, por se mostrar um algoritmo que possui um desempenho ruim independentemente do tipo de entrada e cujo pior caso (entrada decrescente) se agrava de acordo com o crescimento da entrada.

Entretanto, não se pode afirmar muita coisa a respeito do `quicksort primeiro` devido à falha de segmentação apresentada quando as entradas foram superiores a 175 mil números.

Diante das análises feitas, a implementação e análise dos algoritmos implementados trouxe uma melhor compreensão a respeito do impacto da variação do tamanho e tipo de entrada (ordenado, desordenado aleatoriamente e desordenado decrescentemente) no desempenho dos algoritmos de ordenação. Por certo, os 14 algoritmos pedidos representam apenas uma parcela da vastidão de algoritmos de ordenação existentes, mas ainda sim são suficientes para exemplificar e desmistificar as particularidades intrínsecas de cada um e são representantes de conjuntos maiores, como por exemplo, dos algoritmos de ordem quadrática (exemplos são: bubble sort, insertion sort e selection sort) e dos algoritmos linearítmicos (exemplos são: heap sort e merge sort) e etc.

Portanto, alguns dos algoritmos com complexidade quadrática mostraram-se eficientes para qualquer tipo de entrada e tamanho (quicksort random por exemplo), sendo a principal exceção o `selection sort` que possui complexidade quadrática tanto para o melhor caso, médio caso e pior caso, mas ainda assim o pior caso compromete alguns dos mesmos no desempenho (bubble sort por exemplo). Já algoritmos de ordem logarítmica, como `heap sort e merge sort` mostraram que tanto para o melhor caso, médio caso e pior caso possuem um desempenho com pouquíssimas variações. O `quicksort` com suas 4 variações aqui apresentadas, mostrou que escolher o elemento central como pivot parece ser a melhor das 4 formas por parecer a mais consistente.

Bibliografia

Sites da web:

https://pt.wikipedia.org/wiki/Radix$_sort$

https://pt.wikipedia.org/wiki/Quicksort

http://geeksquiz.com/heap-sort/

https://gist.github.com/mycodeschool/9678029

https://pt.wikipedia.org/wiki/Selection$_sort$

https://pt.wikipedia.org/wiki/Shell$_sort$

https://pt.wikipedia.org/wiki/Insertion$_sort$

http://www.programming-algorithms.net/article/40270/Shaker-sort

http://www.programmingsimplified.com/c/source-code/c-program-bubble-sort

Livro:

ZIVIANI, N. Projeto de Algoritmos, Cengage Learning.