Trabalho 2 – Algoritmos e Estruturas de Dados II (CI1056)

Caio Eduardo Ferreira de Miranda – GRR20232359
Departamento de Informática – UFPR
Curitiba – PR
caio.miranda@ufpr.br

I. Como o estudo foi feito

Vetores de diferentes tamanhos foram usados de entrada para todos os algoritmos (um pequeno, um médio e um grande), forçando o pior e o melhor caso em cada um deles. As comparações foram contadas com uma variável do tipo *unsigned long* e o tempo de execução foi medido usando variáveis do tipo *clock_t,* da biblioteca *time.h* da linguagem *C.* O ambiente de teste dos algoritmos foi um computador pessoal, cujo SO é Fedora Linux 39.

II. Testes

Melhor caso (comparações // tempo de execução):

Algoritmos	**Vetor peq.**	**Vetor médio**	**Vetor grande**
Heapsort	118300 // 0.000929s	1512990 // 0.020336s	18393386 // 0.142975s
Quicksort (*)	70774 // 0.002134s	934547 // 0.014412s	11635996 // 0.090881s
Merge Sort	55138 // 0.000644s	716655 // 0.007071s	8811999 // 0.082975s
Counting Sort	0 // 0.000079s	0 // 0.000580s	0 // 0.004993s
Introsort (**)	163170 // 0.003048s	2056514 // 0.014218s	24662529 // 0.150150s

Tabela I: Testes para os vetores no melhor caso. Tamanhos dos vetores: 5000 (pequeno), 50000 (médio) e 500000 (grande). O limite dos valores no Counting Sort é 10 (RNG_CAP = 11). O cenários (*) e (**) nos mostram algumas coisas interessantes em relação à variedade de elementos no Quicksort (*) e a quando um vetor já está ordenado no Introsort (**). Essas coisas serão abordadas, respectivamente, nos tópicos IV e V.

Pior caso (comparações // tempo de execução):

Algoritmos	**Vetor peq.**	**Vetor médio**	**Vetor grande**
Heapsort	118300 // 0.000929s	1512990 // 0.020336s	18393386 // 0.142975s
Quicksort	12497500 // 0.067463s	1249975000 // 6.392254s	7199940000 // 36.437805s (***)
Merge Sort	55138 // 0.000644s	716655 // 0.007071s	8811999 // 0.082975s
Counting Sort	0 // 23.111517s	0 // 23.788017s	0 // 29.101805s
Introsort	670586 // 0.004386s	8533427 // 0.058233s	103266508 // 0.650499s

Tabela II: Testes para os vetores no pior caso. Os tamanhos dos vetores são os mesmos da tabela I, exceto por (***), que será abordado no tópico VI. O limite dos valores no Counting Sort é 4,294,967,295 (RNG_CAP = 4,294,967,296).

III. Introsort

O algoritmo extra escolhido foi o *Introsort,* que funciona da seguinte maneira:

- O algoritmo começa calculando um valor de profundidade (baseado no logaritmo na base 2 do tamanho do vetor), que será usado para determinar quantas chamadas recursivas foram feitas;
- Se o tamanho do vetor for menor que 64, executa um *Insertion Sort* no vetor;
- Se a profundidade do algoritmo chegar a 0, ou seja, se o algoritmo estiver em um nível muito profundo de recursão, executa um *Heapsort* para garantir que o pior caso não tenha um desempenho ruim, que seria o do *Quicksort*;
- Caso nenhum dos casos acima seja verdadeiro na atual chamada da função, realiza uma chamada recursiva típica do *Quicksort,* com a profundidade decrementada em 1 unidade.

O que torna esse algoritmo interessante é o fato dele alternar entre outros algoritmos para que se tenha o melhor desempenho possível em cima de um *Quicksort,* eliminando o seu pior caso e o substituindo pelo do *Heapsort*, que é tão bom quanto o melhor caso. Então, podemos dizer que o *Introsort* é um algoritmo $\Theta(n \log n)$, com n sendo o tamanho do vetor.

IV. O melhor caso do Quicksort

A *macro RNG_CAP,* em *complementar.h,* define a variedade dos elementos na aleatorização dos vetores. Modificando esse valor, conclui-se que o *Quicksort* funciona melhor com uma quantidade maior de valores únicos, tanto que no melhor caso, *RNG_CAP* foi definido em 2000001. Isso nos dá um desempenho melhor em relação a um *Quicksort* em um vetor com valores que pertencem a um intervalo relativamente menor. Abaixo, estão os testes para valores em [0, 10] e valores em [0, 2000000], respectivamente:

```
Tamanho do vetor a ser ordenado: 500000
1: Heapsort; 2: Quicksort; 3: Merge Sort; 4: Counting Sort; 5: Introsort
Selecione qual algoritmo deve ser usado para ordenar o vetor: 2

Comparações: 11635996
Tempo de execução: 0.090881s
```

Figura I: Teste para um vetor de 500000 elementos, variando de 0 até 2000000.

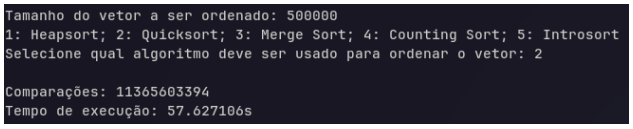

Figura II: Teste para um vetor de 500000 elementos, variando de 0 até 10.

V. O melhor caso do Introsort

Em meio aos testes com o *Introsort,* foi descoberto que um vetor já ordenado gera um resultado melhor para o algoritmo. Abaixo, os testes para um vetor ordenado e um vetor não ordenado, respectivamente:

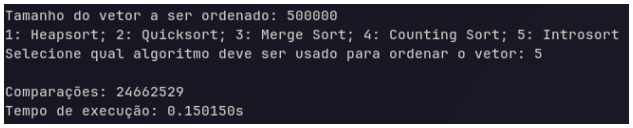

Figura III: Teste para um vetor ordenado de 500000 elementos.

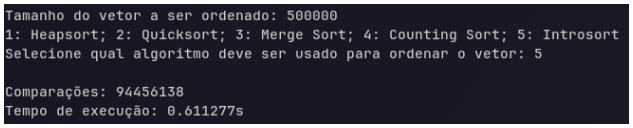

Figura IV: Teste para um vetor não ordenado de 500000 elementos.

Isso se deve ao fato do *Introsort* utilizar *Insertion Sort* para ordenar subvetores que tenham tamanho menor que 64 dentro do vetor.

VI. A exceção do Quicksort

No pior caso do *Quicksort*, em vez de 500000 elementos para um vetor grande, foram usados apenas 120000. Essa decisão de implementação foi feita pois ocorre uma falha de segmentação para valores muito maiores que 120000 para o tamanho do vetor. O erro acontece por conta do grande número de chamadas recursivas que são feitas no *Quicksort* para um vetor ordenado, resultando em um *stack overflow* (verificado com *valgrind*).

VII. Conclusões

Tendo os resultados dos testes, podemos concluir que:

- O *Counting Sort* pode ser o melhor algoritmo de todos, contanto que o escopo de possibilidades para os valores do vetor seja relativamente baixo e inclua apenas valores naturais;
- Por conta do desbalanceamento do vetor, o pior caso do *Quicksort* acaba sendo extremamente custoso. Além disso, é o caso mais ineficiente dentre os demais algoritmos, tanto em relação às comparações quanto ao tempo de execução;
- Valores únicos tendem a melhorar o desempenho do *Quicksort;*
- Quando o vetor está ordenado, o *Introsort* tem um desempenho melhor;
- Em um caso geral, o *Merge Sort* se sairá melhor que os outros algoritmos testados, por conta de sua versatilidade e desempenho. Entretanto, é importante ressaltar que é necessário $2n$ de memória para executar o *Merge Sort* (n sendo o tamanho do vetor);
- Em um caso no qual não se tem $2n$ de memória, é preferível que se use o *Heapsort* em vez do *Quicksort,* por conta de seu bom desempenho em todos os casos ($\Theta(n \log n)$), não importando a variedade dos elementos do vetor.