Relatório Experimental

Algoritmos de Ordenação

- *Insertionsort*
- *Selectionsort*
- *Mergesort*
- *Heapsort*
- *Quicksort*

Sumário

Introdução e Metodologia

Neste relatório apresento os resultados de diversos experimentos realizados com o objetivo de testar os algoritmos de ordenação apresentados em aula e implementados no 3º Exercício Programa (EP).

Os experimentos foram realizados utilizando o seguinte hardware:

- CPU AMD Phenom X4 9500
- 2GB DDR2-800 RAM

E o seguinte sistema operacional:

- Linux Kubuntu 7.10 AMD64 (2.6.22-15)

Conforme mostram os relatórios do sistema operacional que constam nas referências.

Os tempos foram medidos em segundos (*s*), no intervalo de *1s* a *60*s, através da opção *-t* do programa *ordena*, que faz uso da biblioteca *time.h*. Os consumos de memória foram medidos através do utilitário *top*, o valor registrado foi o consumo máximo do processo *ordena* durante toda sua execução.

Os arquivos para os testes foram criados através do programa *rndgen*, cujo código-fonte está disponível nas referências ao final deste documento.

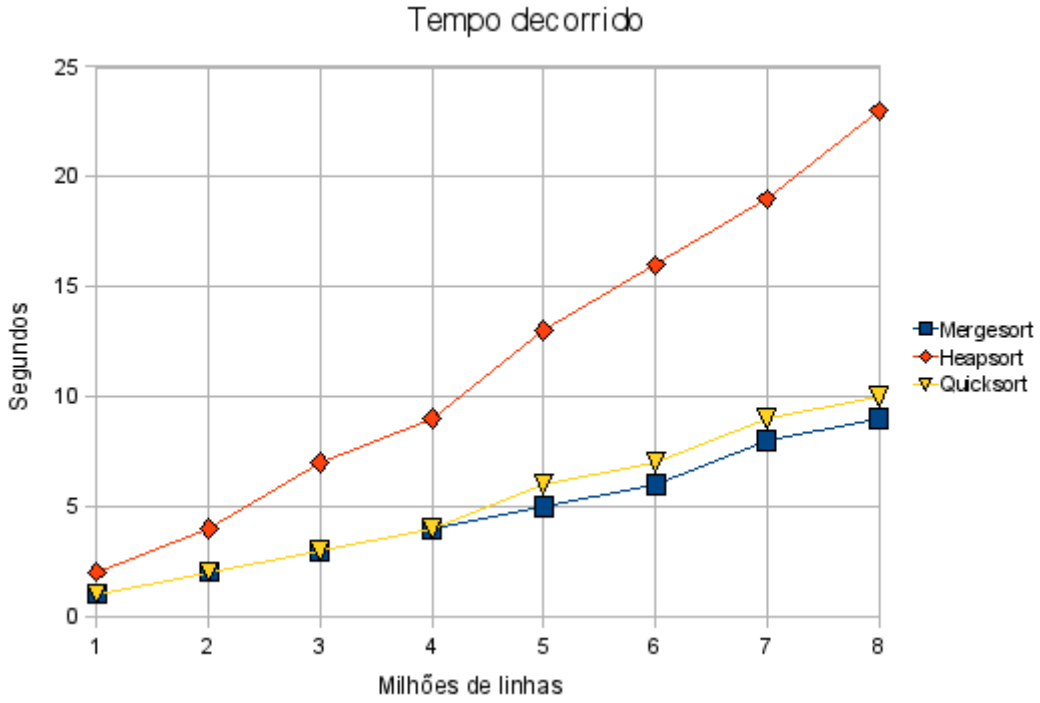
Experimento 1: arquivos embaralhados

Neste teste, utilizei o programa *ordena* com diferentes algoritmos de ordenação para ordenar arquivos sem nenhum padrão aparente. Para obter resultados palpáveis, foram utilizados arquivos com milhões de linhas. Os resultados podem ser observados na tabela seguir, que apresenta o tempo decorrido para cada algoritmo em função do número de milhões de linhas:

Algoritmo \ Milhões de linhas	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Insertionsort	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*
Selectionsort	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*	*> 60*
Mergesort	*1*	*2*	*3*	*4*	*5*	*6*	*8*	*9*
Heapsort	*2*	*4*	*7*	*9*	*13*	*16*	*19*	*23*
Quicksort	*1*	*2*	*3*	*4*	*6*	*7*	*9*	*10*

Os algoritmos elementares (*insertionsort* e *selectionsort*) não estão sequer no mesmo páreo que os algoritmos avançados (*mergesort*, *heapsort* e *quicksort*), pois demoraram mais do que 60s em todos os testes.

Dentre os algoritmos avançados, o *mergesort* se destaca como o mais rápido, seguido muito de perto pelo *quicksort* e deixando o *heapsort* em terceiro lugar. O gráfico abaixo demonstra bem essa situação:



Conforme aumenta o tamanho do arquivo, fica mais evidente a diferença entre os algoritmos. É interessante notar que o algoritmo *quicksort*, apesar de teoricamente apresentar o pior caso proporcional ao quadrado do tamanho da lista ($O(n^2)$), possui a importante característica de em média se comportar similarmente ao *mergesort*, que possui o pior caso proporcional ao produto do tamanho da lista pelo seu logaritmo na base 2 ($O(n \cdot \log_2 n)$). Curiosamente, o *quicksort* se sai melhor até do que o *heapsort*, que teoricamente tem pior caso $O(n \cdot \log_2 n)$, como o *mergesort*.

Experimento 2: arquivos em ordem inversa

Desta vez usei o programa para ordenar em ordem crescente arquivos em ordem decrescente. Os resultados estão na tabela abaixo:

Algoritmo \ Milhões de linhas	1	2	4	8
Insertionsort	*> 60*	*> 60*	*> 60*	*> 60*
Selectionsort	*> 60*	*> 60*	*> 60*	*> 60*
Mergesort	*< 1*	*< 1*	*1*	*2*
Heapsort	*< 1*	*1*	*2*	*5*
Quicksort	*> 60*	*> 60*	*> 60*	*> 60*

O *mergesort* e o *heapsort* se beneficiam com a ordem inversa da lista.

O mesmo não pode ser dito dos outros algoritmos. O *quicksort* passa a se comportar de forma quadrática, pois sua função de separação é incapaz de separar uma lista ordenada ao meio – fator crucial para o desempenho $O(n \cdot log_2 n)$ observado anteriormente.

Experimento 3: arquivos total ou parcialmente ordenados

Para este experimento, foram criados quatro arquivos. O primeiro estava totalmente ordenado. O segundo, o terceiro e o quarto consistiam, respectivamente, de dois, quatro e oito outros arquivos ordenados que foram concatenados. Todos os arquivos possuíam quatro milhões de linhas. Os resultados são apresentados a seguir:

Algoritmo \ Arquivo	**1º**	**2º**	**3º**	**4º**
Insertionsort	*< 1*	*> 60*	*> 60*	*> 60*
Selectionsort	*> 60*	*> 60*	*> 60*	*> 60*
Mergesort	*1*	*2*	*1*	*2*
Heapsort	*1*	*3*	*4*	*4*
Quicksort	*> 60*	*6*	*4*	*4*

Os resultados deste teste são fascinantes. O primeiro detalhe que salta aos olhos é o fato de o *insertionsort* se sair melhor do que qualquer outro algoritmo no caso de uma lista totalmente ordenada devido à simplicidade do algoritmo, que encara nesta situação o seu melhor caso.

O *quicksort*, por outro lado, enfrenta o seu pior caso: uma lista ordenada, onde novamente a função separa se mostra incapaz de separar a lista de forma satisfatória. Seu comportamento é curioso, no sentido de que, conforme menos ordenado estiver o arquivo, melhor seu desempenho.

O *mergesort* e o *heapsort* claramente se beneficiam com uma lista totalmente ordenada, apresentando tempos melhores do que no experimento 1. O *mergesort* apresenta no entanto uma leve vantagem.

Experimento 4: consumo de memória

Neste teste, o programa foi colocado para ordenar um arquivo com oito milhões de linhas e, enquanto ele executava a tarefa, o consumo de memória foi medido usando-se o utilitário *top*. O consumo de memória máximo registrado em *megabytes* (*MB*) é apresentado na tabela abaixo:

Algoritmo	Consumo máximo (MB)
Insertionsort	*221*
Selectionsort	*221*
Mergesort	*317*
Heapsort	*221*
Quicksort	*221*

Todos os algoritmos, salvo o *mergesort*, consomem a mesma quantidade de memória durante o procedimento de ordenação. O *mergesort* apresenta um consumo variado, pois a cada iteração ele armazena parte da lista em um local separado, que é alocado dinamicamente, para o procedimento da intercalação, com um pico de aproximadamente *317MB* de consumo.

O benefício de desempenho que o *mergesort* traz não é gratuito. Seu custo é um maior consumo de memória.

Experimento 5: insertionsort vs. selectionsort

Como nos demais experimentos os algoritmos elementares apresentaram um desempenho inexpressivo, decidi compará-los separadamente, num cenário em escala menor. Para isso, coloquei-os para ordenar arquivos com milhares de linhas. Os resultados podem ser observados na tabela a seguir:

Algoritmo \ Milhares de linhas	**20**	**40**	**60**	**80**
Insertionsort	*1*	*4*	*10*	*19*
Selectionsort	*2*	*7*	*18*	*33*

Os outros algoritmos levariam um tempo desprezível para ordenar estes arquivos pequenos (inferior a 1s). Mas podemos observar de perto a diferença entre o *insertionsort* e o *selectionsort*.

O *insertionsort* se mostra melhor em todos os casos, levando menos tempo do que o *selectionsort* para ordenar os arquivos.

Conclusões

Os experimentos realizados mostram como os algoritmos mostrados em aula desempenham em casos mais comuns e também em casos mais extremos.

Para muitos casos, o *mergesort* é o algoritmo ideal, apresentando o melhor desempenho dentre todos e com pior caso $O(n \cdot \log_2 n)$. Sua desvantagem, no entanto, é o consumo extra de memória, possivelmente tornando-o inutilizável em certos casos em que a memória é um recurso escasso.

O *quicksort* e o *heapsort* também desempenham muito bem. O *quicksort* geralmente é mais rápido do que o *heapsort*, porém este tem a vantagem de pior caso $O(n \cdot \log_2 n)$, contra $O(n^2)$ do *quicksort*, cuja desvantagem fica evidente quando o utilizamos para ordenar arquivos total ou parcialmente ordenados.

Os outros algoritmos são muito simples e fáceis de implementar, mas apresentam um desempenho muito pior. Todavia, o *insertionsort* se sai melhor do que o *selectionsort* e ainda conta com a vantagem de ser muito rápido no raro caso de uma lista totalmente ordenada.

Os testes confirmam as expectativas a respeito das complexidades teóricas dos algoritmos. Em nenhum caso o *mergesort* ou o *heapsort* cresceram quadraticamente, enquanto o *quicksort* desempenha em média muito bem, mas em alguns casos sua natureza quadrática fica evidente. Os outros algoritmos se comportam de forma quadrática praticamente o tempo todo. Os consumos de memória também conferem com o esperado.

Referências

Relatórios do sistema operacional:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 2
model name      : AMD Phenom(tm) 9500 Quad-Core Processor
stepping        : 2
cpu MHz         : 2200.150
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext
3dnow constant_tsc pni cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy altmovcr8
abm sse4a misalignsse 3dnowprefetch osvw
bogomips        : 4572.77
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate

processor       : 1
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 2
model name      : AMD Phenom(tm) 9500 Quad-Core Processor
stepping        : 2
cpu MHz         : 2200.150
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 1
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext
3dnow constant_tsc pni cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy altmovcr8
abm sse4a misalignsse 3dnowprefetch osvw
bogomips        : 4400.53
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate

processor       : 2
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 2
model name      : AMD Phenom(tm) 9500 Quad-Core Processor
stepping        : 2
cpu MHz         : 2200.150
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 2
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
```

```
wp              : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext
3dnow constant_tsc pni cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy altmovcr8
abm sse4a misalignsse 3dnowprefetch osvw
bogomips        : 4400.37
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate

processor       : 3
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 2
model name      : AMD Phenom(tm) 9500 Quad-Core Processor
stepping        : 2
cpu MHz         : 2200.150
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 3
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext
3dnow constant_tsc pni cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy altmovcr8
abm sse4a misalignsse 3dnowprefetch osvw
bogomips        : 4400.37
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate

$ free -k
             total       used       free     shared    buffers     cached
Mem:       2063356    1795892     267464          0      44328    1201224
-/+ buffers/cache:     550340    1513016
Swap:       995988      38472     957516

$ uname -a
Linux  SmartWarthog  2.6.22-15-generic  #1  SMP  Wed  Aug  20  15:47:07  UTC  2008  x86_64
GNU/Linux
```

Programa rndgen:

```
#include<stdlib.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
        unsigned long int linhas = 1000;
        int rnd, inter, i, j;

        inter = 'z' - 'a' + 1;

        if (argc > 1)
                linhas = atoi(argv[1]);

        for (i = 0; i < linhas; i++) {
                for (j = 0; j < 20; j++) {
                        rnd = rand();
                        rnd %= inter;
                        rnd += 'a';
                        printf("%c", rnd);
                }
                puts("");
        }

        return 0;
}
```