Estruturas de Dados II
Primeiro projeto

Caroline Santos Corrêia, 9874735
Caio Oliveira da Silva, 9390301

20 de Junho de 2021

1 Introdução

Como proposto, implementamos 4 algoritmos de ordenação: bubble sort, quick sort, radix sort e heap sort. Pretendemos analisá-los do ponto de vista assintótico (observando o comportamento deles no melhor, pior e médio caso) e empírico (observando o comportamento deles quando passamos listas de tamanhos cada vez maiores para serem ordenadas). Além disso, para cada uma dessas listas, temos a intenção de estudar os algoritmos quando passamos listas aleatórias, ordenadas de forma crescente e ordenadas de forma decrescente. As análises são feitas a partir de gráficos e tabelas, seguindo o que foi ensinado nas aulas.

2 Comentários sobre a implementação dos algoritmos

Tendo em vista a melhor medição de tempo de modo a não tornar o processo muito demorado, cada algoritmo de ordenação foi executado 25 vezes para obter um desvio padrão e uma média mais acurada de tempo. Além disso, para obter mais pontos e tornar o gráfico mais preciso, fizemos uma pequena alteração no tamanho das listas a serem ordenadas. Começamos todos os algoritmos com 1.000 elementos e, até atingir 10.000, aumentamos a lista em 500 elementos a cada iteração, obtendo listas de tamanho 1.500, 2.000 etc. Em seguida, ao chegarmos na lista de tamanho 10.000, aumentamos de 5.000 em 5.000 até chegarmos em 100.000 elementos. Como a partir daqui ficou inviável de calcular o tempo do bubble sort e do bubble sort otimizado, consideramos apenas os outros três algoritmos: quick sort, radix sort e heap sort. Então até 1.000.000 nós aumentamos as listas de 50.000 em 50.000 e, finalmente, de 1.000.000 até 10.000.000 nós aumentamos de 500.000 em 500.000, obtendo, desse modo, 37 pontos a serem analisados para o bubble sort e sua versão otimizada, e 73 para os outros algoritmos.

O processo todo de medição levou aproximadamente 8 horas para ser finalizado e o resultado, com todos os tempos e desvio padrão, pode ser encontrado no arquivo anexado `sorting.txt`. Vale a pena mencionar também que o algoritmo foi executado após o computador ter sido reiniciado e nenhum outro programa foi usado pelos membros do grupo durante o processo todo. Além disso, ao executar o programa, ele sobrescreverá o arquivo `sorting.txt`. Para o desenvolvimento do projeto foi utilizado o editor de texto NetBeans, mas o algoritmo em si foi compilado em um terminal para eliminar a influência que o editor causaria na medição de tempo. E, para finalizar, a análise empírica através de gráficos foi feita usando o programa LibreOffice Calc.

3 Bubble sort

3.1 Análise assintótica

Os algoritmos bubble sort implementados em C foram os seguintes:

```
1  void bubblesort(list *l) {
2          elem aux;
3
4          for (int i = 0; i < l->size - 1; i++) {
5                  for (int j = 0; j < l->size - i - 1; j++) {
6                          if (l->vec[j] > l->vec[j+1]) {
7                                  aux = l->vec[j];
8                                  l->vec[j] = l->vec[j+1];
9                                  l->vec[j+1] = aux;
10                         }
11                 }
12         }
13
14         return;
15 }
```

```
1  void bubblesort_optimized(list *l) {
2          elem aux;
3          int sorted = 0;
4
5          for (int i = 0; i < l->size - 1; i++) {
6                  for (int j = 0; j < l->size - i - 1; j++) {
7                          if (l->vec[j] > l->vec[j+1]) {
8                                  aux = l->vec[j];
9                                  l->vec[j] = l->vec[j+1];
10                                 l->vec[j+1] = aux;
11                                 sorted = 1;
12                         }
13                 }
14                 if (sorted == 0) return;
15         }
16
17         return;
18 }
```

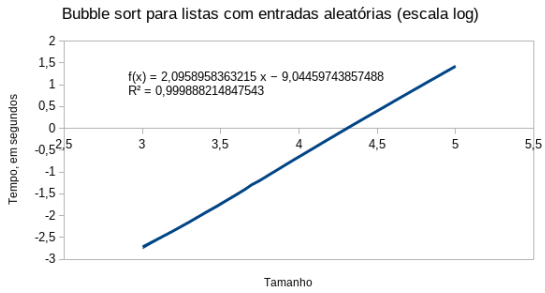
Como tal algoritmo é baseado em comparações, então para a análise levaremos em conta apenas tais operações. Na linha 4 da função `bubblesort` (imagem da esquerda), temos n comparações, sendo n o tamanho do vetor, todas vindas de `i < l->size-1`. No laço interno, quando $i = 0$, temos n comparações vindas de `j < l->size-i-1` e $n - 1$ vindas de `l->vec[j] < l->vec[j+1]` (linhas 5 e 6, respectivamente). Agora note que esse número diminui em 1 sempre que i aumenta. Logo, quando i for $n - 2$, que é o último caso em que o laço externo será executado, j poderá chegar no máximo até 1 e, portanto, teremos duas comparações de j e apenas uma de `l->vec[j] < l->vec[j+1]`. Ou seja, na linha 6 teremos $\sum_{i=1}^{n-1} i$ comparações ao todo e na linha 5 teremos $\sum_{i=2}^{n} i$, o que nos dá um custo total de $\frac{n(n-1)}{2}$ para a linha 6 e $\frac{n(n+1)}{2} - 1$ para a linha 5. Somando com as n comparações da linha 4, obtemos um custo total de $n^2 + n$. Ou seja, o bubble sort é $O(n^2)$. A versão otimizada do bubble sort (imagem da direita) essencialmente fará $n - 1$ comparações extras na linha 14 caso o vetor não esteja ordenado. Ou seja, o custo irá subir para $n^2 + 2n - 1$, o que ainda torna o algoritmo quadrático. No entanto, se o vetor estiver ordenado, o laço externo será executado apenas uma vez, o que implica que teremos apenas uma comparação na linha 5 (`i < l->size-1`), n na linha 6 (`j < l->size-i-1`), $n - 1$ na linha 7 e uma na linha 14, totalizando um custo total de $2n + 1$ operações e, portanto, o algoritmo será $O(n)$ no caso de listas ordenadas. Uma outra otimização que poderia ser feita é colocar `sorted = 0` dentro do laço em i (entre as linhas 5 e 6), pois assim sairíamos do laço logo que o vetor estivesse ordenado, fato que pode ocorrer antes de percorrermos todas as posições possíveis. Em resumo, os casos médio e pior são $O(n^2)$ em ambas as versões do algoritmo e o melhor caso continua sendo $O(n^2)$ na versão padrão, mas $O(n)$ na versão otimizada.

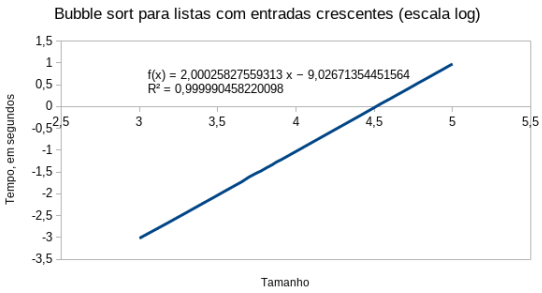
3.2 Análise empírica

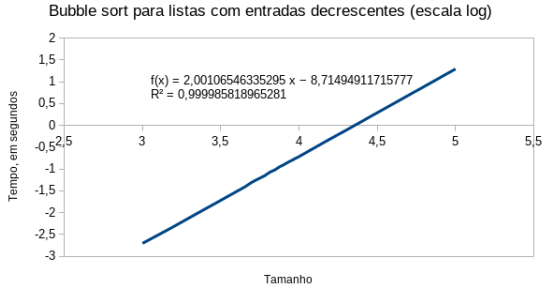
Para a tabela não ficar muito grande, vamos destacar a variação de tamanho das listas em apenas 3 magnitudes, mas no gráfico colocaremos todos os valores calculados e a tabela completa pode ser encontrada no arquivo `sorting.txt`:

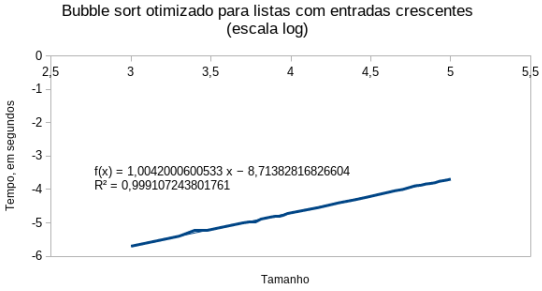
Tamanho	Aleatória (em s)	Desvio aleatória (em s)	Crescente (em s)	Desvio crescente (em s)	Decrescente (em s)	Desvio decrescente (em s)
1000	0.001921	0.000118	0.000965	0.000049	0.001985	0.000135
10000	0.222955	0.000635	0.093591	0.000145	0.192115	0.000172
100000	26.338546	0.053688	9.446916	0.029414	19.516320	0.024172

Como pode ser visto nos três primeiros gráficos, a função é de fato quadrática e a aproximação da reta foi muito boa em todos os casos. Também é possível ver no último gráfico que a versão otimizada é linear para listas com entradas crescentes.









Como a versão otimizada do Bubble sort não realiza muitas outras operações significativas nos casos de listas aleatórias e decrescentes, mostramos apenas o gráfico e a tabela do caso em que a lista possui entradas crescentes, que, como previsto, é linear e foi o mais rápido de todos os algoritmos a "ordenar" listas de 100.000 elementos, com uma diferença significativa de duas casas decimais com relação a todos os outros algoritmos.

Tamanho	Crescente (em s)	Desvio crescente (em s)
1000	0.000002	0.000000
10000	0.000020	0.000003
100000	0.000204	0.000010

4 Quick sort

4.1 Análise assintótica

O algoritmo quick sort implementado em C foi o seguinte:

```
1  void quicksort_rec(list *l, const int begin, const int end) {
2          int pivot;
3
4          if (begin < end) {
5                  pivot = random_partition(l, begin, end);
6                  quicksort_rec(l, begin, pivot - 1);
7                  quicksort_rec(l, pivot + 1, end);
8          }
9
10         return;
11 }
```

```
1  int random_partition(list *l, int begin, int end) {
2          srand(time(NULL));
3          int k = begin + (rand() % (end - begin + 1));
4          elem aux = l->vec[k];
5          l->vec[k] = l->vec[end];
6          l->vec[end] = aux;
7
8          return partition(l, begin, end);
9  }
```

```
1  int partition(list *l, int begin, int end) {
2          int i = begin - 1;
3          elem aux, pivot = l->vec[end];
4
5          for (int j = begin; j < end; j++) {
6                  if (l->vec[j] < pivot) {
7                          i++;
8                          aux = l->vec[i];
9                          l->vec[i] = l->vec[j];
10                         l->vec[j] = aux;
11                 }
12         }
13
14         aux = l->vec[i+1];
15         l->vec[i+1] = l->vec[end];
16         l->vec[end] = aux;
17
18         return i + 1;
19 }
```

Como o quick sort também é um algoritmo de comparação, esta será a operação analisada. Comecemos pela função `partition` (imagem da direita). Nela temos duas linhas com comparação: 5 e 6. A linha 5 fará sempre `end - begin + 1` comparações com a variável `j` e a linha seguinte fará sempre `end - begin` comparações entre o pivô e o restante da lista. Então, dada uma lista de tamanho n, tal função realizará $2n + 1$ operações de comparação.

A função `random_partition` não realiza comparações (a menos que exista alguma por trás da função rand) e, portanto, podemos "desconsiderá-la" da contagem. Note que ela funciona apenas como um intermediário para dizer qual será o tamanho da lista passada para a função `partition` na próxima iteração. E, por fim, a função `quicksort_rec` irá fazer uma comparação na linha 4 e, ao chamar a função `random_partition` pela primeira vez, o custo da linha 5 será de $2n + 1$ comparações, como já vimos.

No entanto, as linhas seguintes são um pouco mais delicadas de analisar e é aqui que entram as diferentes complexidades do quick sort. Se déssemos o azar de o pivô ser sempre o último elemento ou o primeiro, então passaríamos pra uma das chamadas recursivas uma lista de tamanho 1 e outra de tamanho $n-1$. Ou seja, em ambos os casos cairíamos na recorrência $Q(n) = 2n + 2 + Q(1) + Q(n-1)$. Como $Q(1) = 1$, ficamos com custo $Q(n) = 2n + 3 + Q(n-1)$. Resolvamos tal recorrência. Observe que na k-ésima chamada da recursão teríamos um custo de $Q(n) = (2n+3)k + Q(n-k)$. Logo, como iremos parar apenas quando $n-k = 1$, isso significa que $k = n-1$ e, portanto, o custo total do quick sort se torna $Q(n) = (2n+3)(n-1)+1 = 2n^2+n-2$, que é $O(n^2)$. O melhor caso do quick sorte é quando a lista é sempre dividida ao meio, pois nesse caso a recorrência se torna $Q(n) = 2n+2+Q(n/2)+Q(n/2) = 2(n+1+Q(n/2))$. Como na k-ésima chamada temos um custo de $Q(n) = 2k(n+1+Q(n/(2^k))$ e a recursão acaba quando $n/2^k = 1$, então, usando propriedades de exponencial e logaritmo, sabemos que isso ocorre quando $k = \log_2(n) = \log(n)$. Ou seja, nesse caso o custo do quick sort é $Q(n) = 2n\log(n) + 2\log(n) + 1$ e, portanto, $O(n\log(n))$. Mas, como estamos considerando o caso em que os pivôs são escolhidos aleatoriamente, a análise complica um pouco.

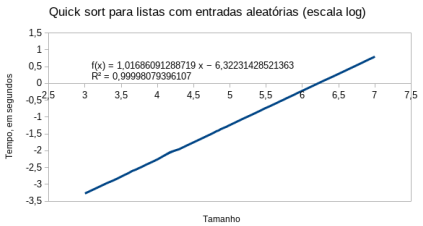
Primeiramente observe que, se na primeira chamada obtivermos um pivô $k_1 > 0$, então o custo da primeira chamada da função `quicksor_trec` será $Q(n) = 2n + 2 + Q(k_1) + Q(n-k_1-1)$. Ou seja, cada chamada irá depender do pivô obtido, que nesse caso é aleatório. Mas, se assumirmos que todos os números de 1 até $n-1$ são escolhidos com mesma probabilidade, podemos tirar a média sobre todos os números e fazer a seguinte estimativa para o custo: $Q(n) = 2n+2+\frac{1}{n}\sum_{i=1}^{n-1}(Q(i)+Q(n-i-1)) = 2n+2+\frac{2}{n}\sum_{i=1}^{n-1}Q(i)$. Agora observe que podemos eliminar esse somatório ao subtrairmos $Q(n-1)$ de $Q(n)$, obtendo a seguinte equação de recorrência: $nQ(n) = (n+1)Q(n-1) + 2n - 2$. Dividindo essa equação por $n(n+1)$ e ignorando o termo com -2 no numerador, já que estamos fazendo uma análise de pior caso, ficamos com a seguinte desigualdade: $\frac{Q(n)}{n+1} \leq \frac{Q(n-1)}{n} + \frac{2}{n+1}$. Agora note que, se aplicarmos o mesmo raciocínio para $n-1$, iremos obter a desigualdade $\frac{Q(n-1)}{n} \leq \frac{Q(n-2)}{n-1} + \frac{2}{n}$. Logo, se continuarmos repetindo esse processo de usar as desigualdades para chamadas menores, o seguinte padrão emerge: $\frac{Q(n)}{n+1} \leq \frac{Q(1)}{2} + \sum_{i=2}^{n}\frac{2}{i+1}$. Agora é só fazer uma estimativa clássica e perceber que $\sum_{i=2}^{n}\frac{2}{i+1} \leq 2\sum_{i=2}^{n}\frac{1}{i+1} \leq 2\int_1^n \frac{1}{x}dx = 2\ln(n)$. Ou seja, voltando pra desigualdade $\frac{Q(n)}{n+1} \leq \frac{Q(1)}{2} + \sum_{i=2}^{n}\frac{2}{i+1}$, temos que $Q(n) \leq (n+1)(\frac{1}{2} + 2\ln(n))$, o que também nos dá $O(n\ln(n))$ no pior caso da versão com pivôs aleatórios.

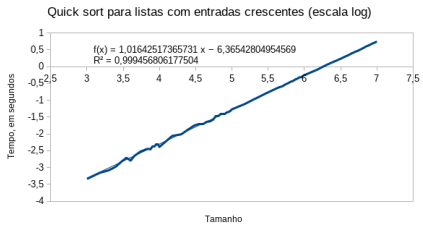
4.2 Análise empírica

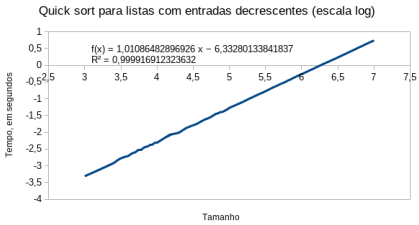
Novamente para que a tabela não fique muito grande, vamos destacar a variação de tamanho das listas em apenas 5 magnitudes, mas no gráfico colocaremos todos os valores calculados.

Tamanho	Aleatória (em s)	Desvio aleatória (em s)	Crescente (em s)	Desvio crescente (em s)	Decrescente (em s)	Desvio decrescente (em s)
1000	0.000538	0.000011	0.000459	0.000008	0.000491	0.000000
10000	0.005516	0.000032	0.004096	0.000022	0.004939	0.000026
100000	0.057942	0.000237	0.053728	0.002871	0.053658	0.024172
1000000	0.604773	0.011520	0.557524	0.037652	0.542777	0.020716
10000000	6.245036	0.037458	5.531738	0.122921	5.506720	0.088779

É possível ver que em todos os casos obtivemos funções muito parecidas, sendo que nenhuma delas é linear, mas ainda assim com custo muito baixo. Também tivemos aproximações muito boas para todos os casos, como pode ser visto pelo R^2.







5 Radix sort

5.1 Análise Assintótica

O algoritmo radix sort implementado em C foi o seguinte:

```
1  void radixsort(list *l) {
2          elem max = l->vec[0];
3          int position = 1;
4
5          for (int i = 1; i < l->size; i++) {
6                  if (l->vec[i] > max) {
7                          max = l->vec[i];
8                  }
9          }
10
11         while (max / position > 0) {
12                 countingsort(l, l->size, position);
13                 position *= 10;
14         }
15
16         return;
17 }
```

```
1  void countingsort(list *l, const int size, const int position) {
2          int *B = (int *) calloc(10, sizeof(int));
3          int *C = (int *) calloc(size, sizeof(int));
4          int key;
5
6          for (int i = 0; i < size; i++) {
7                  key = l->vec[i] / position;
8                  key = key % 10;
9                  B[key]++;
10         }
11
12         for (int i = 1; i < 10; i++) {
13                 B[i] += B[i-1];
14         }
15
16         for (int i = (size - 1); i >= 0; i--) {
17                 key = l->vec[i] / position;
18                 key = key % 10;
19                 B[key]--;
20                 C[B[key]] = l->vec[i];
21         }
22
23         for (int i = 0; i < size; i++) {
24                 l->vec[i] = C[i];
25         }
26
27         free(B);
28         free(C);
29         return;
30 }
```

Como tal algoritmo é baseado em atribuições ao invés de comparações, essa será a operação analisada em tal algoritmo. Comecemos pela função `countingsort` (imagem da direita). Deixando de lado a inicialização dos vetores usando `calloc`, pois tal inicialização provavelmente é mais eficiente do que uma atribuição normal, podemos começar a análise na linha 6. Nela teremos $n+1$ atribuições para a variável i, sendo n o tamanho do vetor, pois i começa em 0 e termina em n com ambos os valores inclusos. Nas três próximas linhas, temos um total de n atribuições em cada, totalizando um custo de $4n+1$ atribuições para esse primeiro laço. O segundo laço é um pouco menos custoso e tem valor constante de 19 atribuições, sendo 10 delas para a variável i e 9 para o vetor B (linha 13). O terceiro laço terá $n+1$ atribuições para a variável i na linha 16, pois i irá de $n-1$ até -1 com ambos os valores inclusos, e as próximas quatro linhas tem uma atribuição em cada, totalizando um custo de $5n+1$ atribuições para o terceiro laço. E, finalmente, o último laço tem custo total de $2n+1$ atribuições, sendo $n+1$ da variável i na linha 23 e n da linha 24. Ou seja, o counting sort tem custo total de $11n+22$, o que o torna um algoritmo linear.

Passemos agora para a função `radixsort`. Temos duas atribuições nas linhas 2 e 3. Em seguida teremos n atribuições para a variável i na linha 5 e, no pior caso, $n-1$ atribuições para a variável `max` na linha 7. Em ambos os casos continuamos com custo linear, o que não interfere na análise assintótica. E, finalmente, se denotarmos por d o número de dígitos da variável `max`, o laço definido pelo `while` será executado d vezes e, portanto, teremos um custo total de $d(11n+22)$ atribuições para a linha 12 e d para a linha 13, totalizando um custo total de $2+O(n)+d+d(11n+22)$ atribuições, o que torna essa implementação do radix sort um algoritmo $O(dn)$. No entanto, poderíamos ter usado outras bases dentro do counting sort ao invés da base decimal, o que faria com que o custo do segundo laço da imagem da direita fosse $2b-1$ ao invés de constante 19 (caso em que $b=10$). Logo, uma análise mais precisa torna o algoritmo $O(d(n+b))$, sendo d o número de dígitos do maior valor da lista a ser ordenada, n o tamanho da lista e b a base com que faremos o counting sort. Tal algoritmo possui o mesmo custo assintótico em todos os casos.

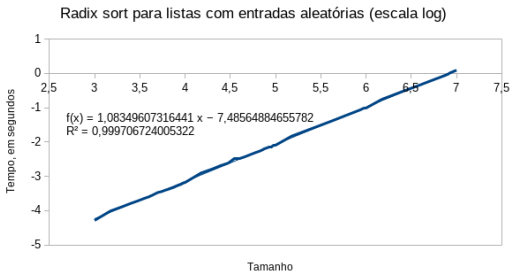
5.2 Análise empírica

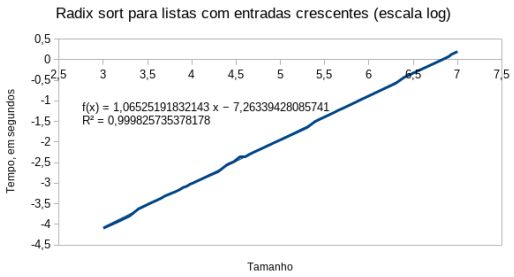
Vamos destacar, novamente, a variação da sequência de números em 5 magnitudes:

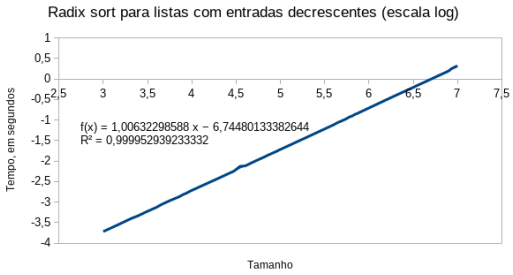
É possível observar mais uma vez que em nenhum dos casos tivemos um custo linear, mas todos foram

Tamanho	Aleatória (em s)	Desvio aleatória (em s)	Crescente (em s)	Desvio crescente (em s)	Decrescente (em s)	Desvio decrescente (em s)
1000	0.000052	0.000009	0.000081	0.000004	0.000192	0.000011
10000	0.000660	0.000015	0.000982	0.000023	0.001930	0.000030
100000	0.008238	0.000050	0.011284	0.000049	0.019140	0.000066
1000000	0.100186	0.000146	0.132185	0.000261	0.196382	0.000391
10000000	1.223104	0.015136	1.570243	0.015094	2.079507	0.016726

bem baixos se comparados com o restante dos algoritmos e todos estão bem aproximados.







6 Heap sort

6.1 Análise assintótica

O algoritmo heap sort implementado em C foi o seguinte:

```c
void heapsort(list *l) {
        elem aux;

        for (int i = (l->size / 2 - 1); i >= 0; i--) {
                heapify(l, l->size, i);
        }

        for (int i = (l->size - 1); i >= 1; i--) {
                aux = l->vec[0];
                l->vec[0] = l->vec[i];
                l->vec[i] = aux;
                heapify(l, i, 0);
        }

        return;
}
```

```c
void heapify(list *l, const int size, const int i) {
        int max = i, left = 2 * i + 1, right = 2 * i + 2;
        elem aux;

        if (left < size && l->vec[left] > l->vec[max]) {
                max = left;
        }

        if (right < size && l->vec[right] > l->vec[max]) {
                max = right;
        }

        if (max > i) {
                aux = l->vec[i];
                l->vec[i] = l->vec[max];
                l->vec[max] = aux;
                heapify(l, size, max);
        }

        return;
}
```

Como o heap sort também é um algoritmo baseado em comparações, essa será a operação analisada. Comecemos pela função `heapify` (imagem da direita). Note que sempre teremos pelo menos 4 comparações sendo feitas, pois pode ser que a comparação `right < size` falhe e em C isso significa que a outra condição não será verificada, já que temos um && no `if`. Agora observe que o objetivo dessa função é olhar para a lista como se ela fosse uma árvore binária quase cheia. Logo, caso a chamada recursiva seja feita, no pior dos casos ela será executada com 2/3 dos nós da árvore, que é o caso em que temos exatamente metade das folhas e caímos na subárvore que está cheia. Ou seja, podemos analisar a função `heapify` usando a seguinte recorrência: $H(n) \leq c + H(\frac{2n}{3})$, sendo c uma constante, que pode ser 4 ou 5, mas que no caso

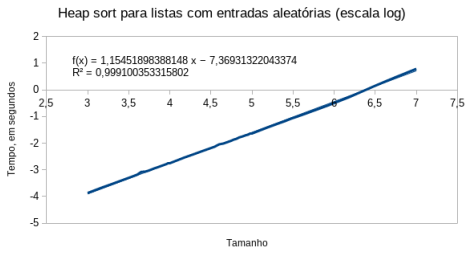
assintótico não fará diferença. Após k chamadas da função heapify, obtemos $H(n) \leq kc + H(\frac{2^k n}{3^k})$. Como no pior dos casos tal função irá parar quando $\left(\frac{2}{3}\right)^k n = 1$ e podemos assumir que $H(1)$ tem custo c, então note que isso irá ocorrer quando $k = \log_{3/2}(n)$ e, portanto, $H(n) \leq c \log_{3/2}(n) + H(1) = c(\log_{3/2}(n) + 1)$. Ou seja, se n é o tamanho da lista, no pior dos casos o custo da função `heapify` será $O(\log_{3/2}(n))$.

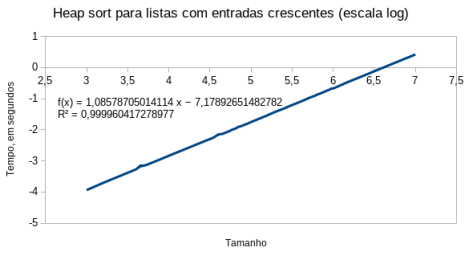
Analisemos agora a função `heapsort`. Note que na linha 4 teremos $\frac{n}{2} + 1$ comparações da variável i e, na linha 5, chamaremos a função `heapify` variando a altura da árvore em cada chamada, começando da metade da árvore até a raiz, o que nos daria um pior caso de $\sum_{i=\frac{n}{2}}^{n} \log_{3/2}(i)$ comparações na linha 5. Lembrando que $\log(a) + \log(b) = \log(ab)$, tal somatório acaba virando quase que o logaritmo de $n!$ e, pela aproximação de Stirling, $\log(n!)$ tem comportamento assintótico $O(n \log_{3/2}(n))$. Ou seja, como no nosso caso $\sum_{i=\frac{n}{2}}^{n} \log_{3/2}(i) \leq \log(n!)$, podemos majorar nossa função por $O(n \log_{3/2}(n))$. Além disso, como $n < n \log_{3/2}(n)$, podemos desconsiderar as $\frac{n}{2} + 1$ comparações da linha 4 e com isso o custo do primeiro laço é $O(n \log_{3/2}(n))$. E, finalmente, teremos n comparações na linha 8, o que também poderá ser ignorado do ponto de vista assintótico, e na linha 12 chamaremos a função heapify sempre começando da raiz da árvore, mas diminuindo o tamanho em 1 em cada chamada, o que resulta em um custo $\sum_{i=1}^{n-1} \log_{3/2}(i)$, que é $O(n \log_{3/2}(n))$, pois nesse caso temos exatamente $\log_{3/2}((n-1)!)$ (novamente pela aproximação de Stirling). Ou seja, o custo total do heap sort será também $O(n \log_{3/2}(n))$. Vale a pena observar que, caso o vetor esteja ordenado de forma decrescente, então a linha 5 da função `heapsort` terá sempre custo constante, já que nenhuma condição será satisfeita dentro da função `heapify`. Logo, na verdade o custo dessa primeira chamada será linear, mas isso é desbalanceado na linha 12, que essencialmente terá que trocar todas as posições, pois agora a lista fere todas as condições. E o oposto acontece caso o vetor esteja ordenado de modo crescente, isto é, todo o trabalho é feito logo no primeiro laço e o segundo realizará apenas as comparações. Observe também que isso se refletiu na análise empírica, tendo uma diferença de aproximadamente 3 segundos e meio para ordenar uma lista com entradas aleatórias, que fará boa parte das comparações em ambos os laços, e listas com entradas crescentes e decrescentes, que levaram aproximadamente o mesmo tempo.

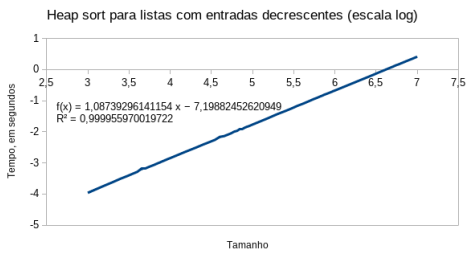
6.2 Análise empírica

Mais uma vez, temos o tempo e desvio padrão da variação da sequência de números em 5 magnitudes:

Tamanho	Aleatória (em s)	Desvio aleatória (em s)	Crescente (em s)	Desvio crescente (em s)	Decrescente (em s)	Desvio decrescente (em s)
1000	0.000135	0.000001	0.000115	0.000001	0.000109	0.000001
10000	0.001822	0.000029	0.001462	0.000017	0.001424	0.000016
100000	0.023642	0.000091	0.017695	0.000107	0.017048	0.000115
1000000	0.318544	0.015072	0.215969	0.000424	0.211536	0.001215
10000000	6.095673	0.024137	2.625677	0.023684	2.572726	0.016101







7 Conclusões

As análises teóricas foram baseadas no livro [1] e o que se pode concluir da análise empírica é que, de todos os algoritmos implementados, o que mais se destacou foi o radix sort, levando praticamente menos de 2 segundos para ordenar listas de tamanho 10.000.000 em todos os casos. O caso em que a lista é decrescente levou um pouco mais de 2 segundos por conta do algoritmo que gera a lista decrescente ter sido implementado considerando-se o maior valor possível que cabe em uma variável do tipo inteiro, o que acaba sendo um número com muitos dígitos e isso influencia bastante o algoritmo, já que ele leva em consideração o número de dígitos do maior valor da lista para fazer a ordenação a partir dele. Vale a pena também observar que o quick sort não caiu no pior caso, o que coincide com a análise assintótica do algoritmo que gera pivôs aleatórios.

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L Rivest, and S. Clifford. *Introduction to Algorithms.* The MIT Press, 2009.