RELATÓRIO DA DISCIPLINA ESTRUTURA DE DADOS

ALUNO: HUGO RAFAEL DE MEDEIROS FERNANDES

PROFESSOR: JOÃO PAULO DE SOUZA MEDEIROS

LABORATÓRIO ALAN TURING – PRÉDIO BSI

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE – CERES/CAICÓ

Email: hugo.fernandes@live.com

RESUNO:

Utilizando os conhecimentos estudados em sala sobre algoritmo e complexidade, foi desenvolvido em laboratório e em sala de aula, uma tarefa prática utilizando algoritmos de ordenação e outros, para verificar seus comportamentos e determinar a ordem de seus tempos de execução.

1. INTRODUÇÃO

O objetivo deste relatório é relatar um experimento realizado em laboratório e em sala de aula pela disciplina *Estrutura de Dados*, com o objetivo de verificar, desenvolver e entender o funcionamento dos algoritmos em questão, medindo e comparando seus tempos de execução.

2. ALGORITMOS UTILIZADOS

Os algoritmos utilizados foram:

- Algoritmo de Busca;
- Algoritmo de Busca Binária;
- Algoritmo de Fibonacci Iterativo;
- Algoritmo de Fibonacci Recursivo;
- Algoritmo *Insertion–Sort*;
- Algoritmo *Merge–Sort*;
- Algoritmo *Quick–Sort*;
- Algoritmo *Distribution–Sort*.

3. FERRAMENTAS UTILIZADAS

As ferramentas utilizadas foram:

- Papel e Caneta;
- Computador;
- Sistema Operacional *Debian GNU/Linux*;

- *Software Gnuplot*;
- Processador de texto *Gedit*;
- Linguagem de programação *C*.

4. MÉTODO DE MEDIÇÃO E COMPARAÇÃO DOS RESULTADOS

O método de medição utilizado para capturar o tempo de execução nos algoritmos foi através da função *gettimeofday* presente na biblioteca *sys/time.h* da linguagem *C*, onde seus resultados seram inseridos num arquivo de texto, por onde é gerado o gráfico do tempo de execução do algoritmo através da ferramenta *Gnuplot*.

Outro método para medir o tempo de execução do algoritmo é calculando matematicamente sua complexidade utilizando o método da substituição. Dessa forma podendo determinar aproximadamente seu tempo de execução.

Por último, compara-se os resultados dos dois métodos para verificar se os dados são precisos e assim avaliar qual algoritmo é mais eficiente.

5. REFERÊCIAS

1. Algoritmos: teoria e prática / Thomas H. Comen... [et al.]; tradução da segunda edição [americana] Vandenberg D. de Souza – Rio de Janeiro: Elsevier, 2002 – 6ª reimpressão.
2. http://www.dicas-l.com.br/arquivo/usando_gnuplot_para_gerar_bons_graficos.php#.U0TiHvldWcU – acessado entre março e abril de 2014.
3. http://fig.if.usp.br/~esdobay/c/gcc.html - acessado entre março e abril de 2014.
4. http://professor.ufabc.edu.br/~daniel.martin/ED/tempo.html - acessado entre março e abril de 2014.
5. Wikipédia – acessado entre março e abril de 2014.

6. ANEXOS

Abaixo estão páginas escaneadas dos pseudo-algoritmos com uma prevê descrição e os cálculos de sua complexidade, bem como os gráficos gerados a partir dos códigos desenvolvidos em laboratório.

Algoritmo de busca

```
algoritmo busca(A, n, K)
    for i = 1 to n do
        if A[i] == K then
            return i
    return 0
```

• Onde, A é o vetor, n é o tamanho de A e K o elemento a ser encontrado.

Descrição

O algoritmo busca um elemento que esteja dentro de um vetor. Ele percorre todo o vetor comparando cada elemento com o elemento procurado. Se depois disto ele não tiver sucesso, retorna falso.

Complexidade

Esse algoritmo possui três casos de complexidade onde seu tempo de execução é diferente, são esses:

Melhor caso

Quando o elemento buscado está na primeira posição do vetor.

$$T_b(n) = c_1 + c_2 + c_3 \Rightarrow T_b(n) = \alpha$$

Ele executa as primeira, segunda e terceira linhas do algoritmo, encontrando o elemento e terminando.

complexidade constante: $\Omega(\alpha)$

Pior Caso

Quando o elemento buscado não se encontra em nenhuma posição do vetor.

$$T_w(n) = c_1(n+1) + c_2 n + c_4 \Rightarrow T_w(n) = \alpha n + \beta$$

Ele percorre todo o vetor, executando as primeira e segunda linhas o máximo de vezes possível, como a decisão da segunda linha não é satisfeita, ele não executa a terceira linha. Quando o laço da primeira linha termina, só basta executar a quarta linha e terminar a execução.

Complexidade linear: $O(n)$

Medio Caso

Quando o elemento buscado pode está em qualquer posição no vetor, inclusive não está.

Probabilidade de está no vetor = P
Probabilidade de não está no vetor = (1-P)

$$P + (1-P) = 1$$

$$T_a(n) = p_1\frac{P}{n} + p_2\frac{P}{n} + p_3\frac{P}{n} + \dots + p_n\frac{P}{n} + (1-P)(n+1)$$

$$= \frac{P}{n}\left(p_1 + p_2 + p_3 + \dots + p_n\right) + (1-P)(n+1)$$

$$= \frac{P}{n}\sum_{i=1}^{n} i + (1-P)(n+1)$$

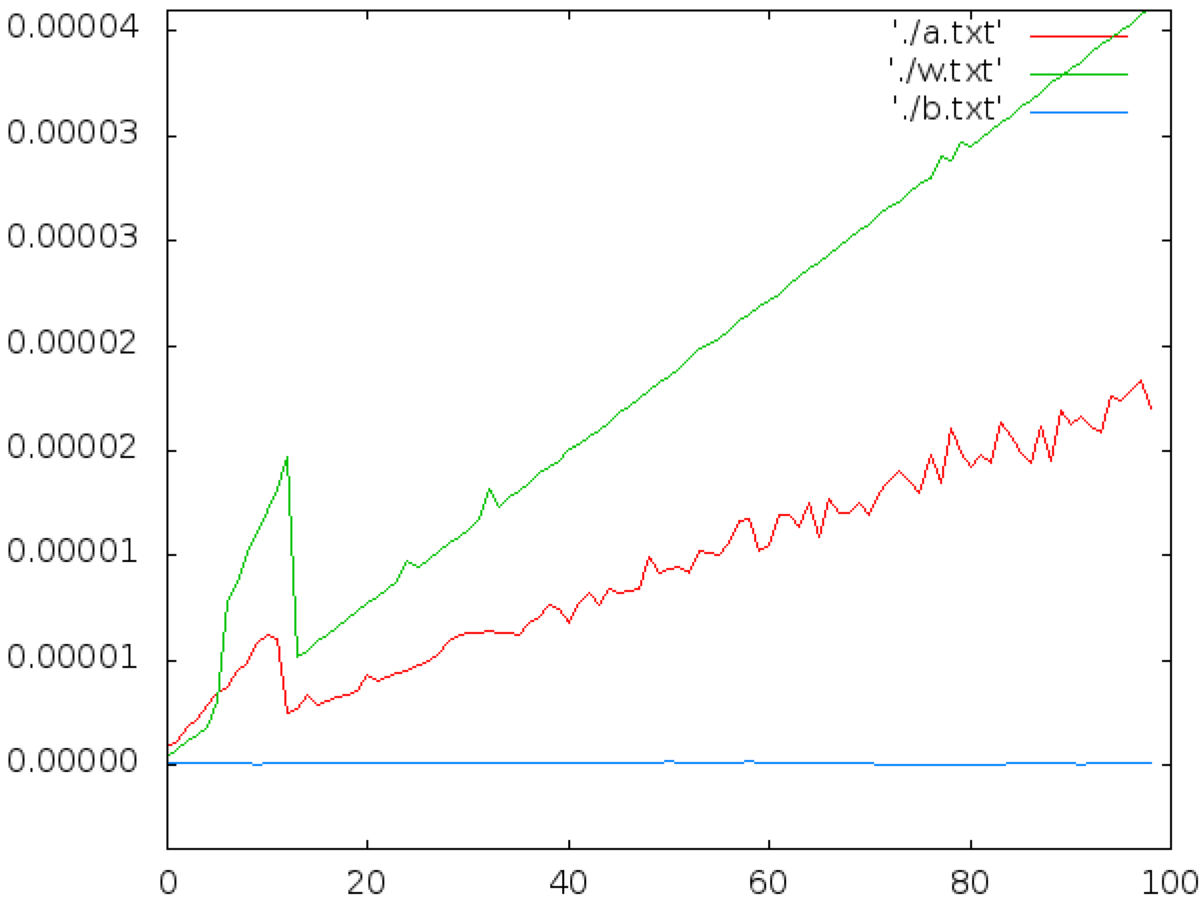
$$= \frac{P}{n}\left(\frac{n}{2}(n+1)\right) + (1-P)(n+1)$$

$$= \frac{P(n+1)}{2} + (1-P)(n+1)$$

$$= (n+1)\left(\frac{P}{2} + (1-P)\right) = (n+1)\frac{(2-P)}{2} = (n+1)\left(1-\frac{P}{2}\right)$$

$$T_a(n) = (n+1)\left(1-\frac{P}{2}\right)$$

Complexidade linear: $\Theta(n)$


'./a.txt'
'./w.txt'
'./b.txt'
0.00004
0.00003
0.00003
0.00002
0.00002
0.00001
0.00001
0.00000
0
20
40
60
80
100

Algoritmo de Busca Binária

```
algoritmo busca(V, s, e, K)
   i = ⌊(s+e)/2⌋
   if V[i] == K
   |    return i
   if s == e
   |    return -1
   if V[i] < K
   |    busca(V, i+1, e, K)
   else
   |    busca(V, s, i-1, K)
```

Onde, V é o vetor, s é a primeira posição do vetor, e é a última posição do vetor e K o elemento buscado.

Descrição

O algoritmo busca um elemento num vetor, sendo que este começa na busca da metade do vetor, tendendo a procurar para a direita se K > V[i] ou para a esquerda se K < V[i].

Complexidade

Este algoritmo possui três casos onde o tempo de execução muda dependendo da situação, são eles:

Melhor Caso

Quando o elemento buscado está no meio do vetor.

$$T_b(n) = c_1 + c_2 + c_3 \Rightarrow T_b(n) = \alpha$$

Complexidade constante: $\Omega(\alpha)$

Pior Caso

Quando o elemento buscado não está no vetor.

Caso base (quando $s = e$)

$$T_w(1) = \underbrace{c_1 + c_2 + c_4 + c_5}_{b} \Rightarrow T_w(1) = b$$

Equação de recorrência

$$T_w(n) = \underbrace{c_1 + c_2 + c_4 + c_6 + c_0}_{a} + T_w(n/2)$$

$$T_w(n) = a + T_w(n/2)$$

Onde, c_0 é a execução a linha 7 ou linhas 8 e 9.

$T_w(n/2) = a + T_w(n/2^2)$

$T_w(n) = a + (a + T_w(n/2^2))$

$= 2a + T_w(n/2^2)$

$T_w(n/2^2) = a + T_w(n/2^3)$

$T_w(n) = 2a + [a + T_w(n/2^3)]$

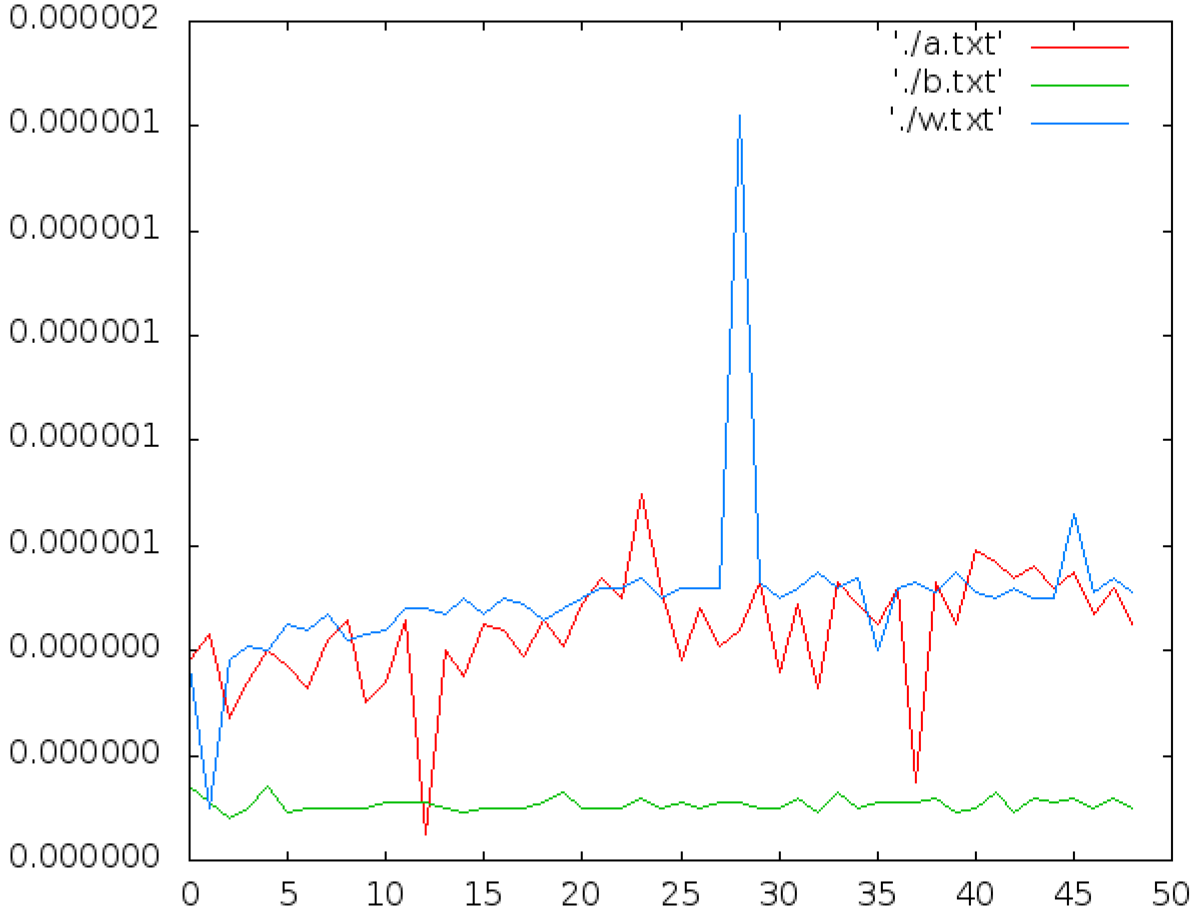
$= 3a + T_w(n/2^3)$

$T_w(n) = xa + T_w(n/2^x) \leftarrow$ Para atingir o caso base $2^x = n$, então $\underline{\log_2 n = x}$

$T_w(n) = \log_2 n a + T_w(1)$

$= \log_2 n a + b$

Complexidade logaritmica : $\Theta(\log_2 n)$


'./a.txt'
'./b.txt'
'./w.txt'
0.000002
0.000001
0.000001
0.000001
0.000001
0.000001
0.000000
0.000000
0.000000
0
5
10
15
20
25
30
35
40
45
50

Algoritmo Fibonacci Iterativo

```
algoritmo fib(n)
    i = 1
    j = 0
    if n ≤ 1 then
        return j
    for k = 3 to n do
        t = i + j
        j = i
        i = t
    return i
```

Descrição

Este algoritmo retorna o n-ésimo elemento da sequência de fibonacci. Ele utiliza uma estratégia iterativa calculando elemento por elemento até chegar ao elemento procurado.

Complexidade

Este algoritmo possui apenas dois casos, sendo que executa sempre as mesmas instruções. Sua complexidade é:

Quando $n > 1$,

$$T(n) = c_1 + c_2 + c_3 + c_5(n-1) + c_6(n-2) + c_9$$

$$T(n) = \alpha + \beta(2n-3) \qquad \Theta(n)$$

Quando $n \leq 1$,

$$T(n) = \overbrace{c_1 + c_2 + c_3 + c_4}^{\alpha}$$

$$T(n) = \alpha \qquad \Theta(\alpha)$$

Complexidade linear: $\Theta(n)$

Algoritmo Fibonacci Recursivo

```
algoritmo fib(n)
1.   if n < 3 then
2.        return n - 1
3.   else
4.        return fib(n - 1) + fib(n - 2)
```

Descrição

O algoritmo retorna o n-ésimo elemento da sequência de fibonacci. Ele utiliza uma estratégia recursiva.

Complexidade

Caso base (quando $n < 3$)

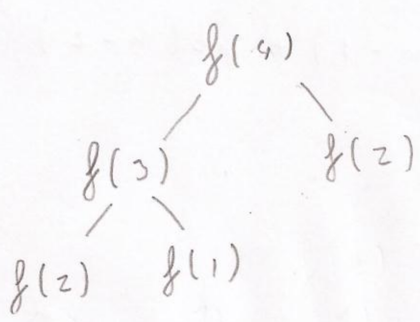
$$T(n) = c_1 + c_2 = \alpha$$

Equação de recorrência (quando ocorre recorrência)

$$T(n) = c_1 + c_3 + c_4 + T(n-1) + T(n-2)$$

Para resolver a complexidade deste algoritmo o método da substituição não é o indicado. Neste caso o método mais aconselhado é o método da árvore de recorrência.

Assumindo que $n = 4$:



$1 + 0 + 1 = 2$

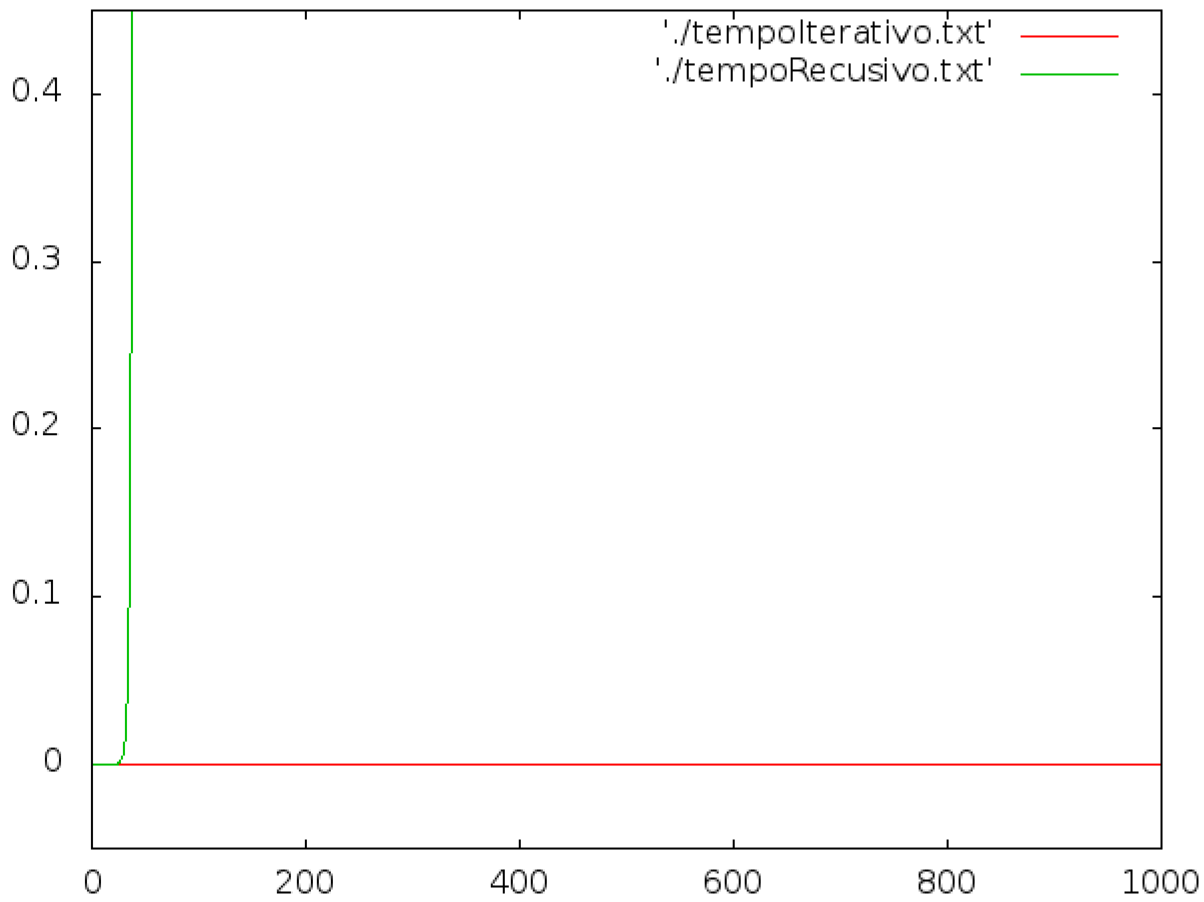
Equação do número de nós da árvore

$n^{\underline{o}}$ de nós $= 2^n - 2^{n-1} - 1$

Então, $T(n) \in O(2^n)$

Como $fib(n)$ realiza $2^n - 2^{n-1} - 1$ chamadas, substituindo:

Então, $T(n) \in \Theta(fib(n))$



'./tempoIterativo.txt'
'./tempoRecusivo.txt'
0.4
0.3
0.2
0.1
0
0
200
400
600
800
1000

Algoritmo Merge-sort

```
algoritmo mergeSort
    if s < e then
        m = ⌊(s + e)/2⌋
        mergeSort(A, s, m)
        mergeSort(A, m + 1, e)
        merge(A, s, m, e)

algoritmo merge(A, s, m, e)
    k = 1
    i = s
    j = m + 1
    while k ≤ e - s + 1 do
        if (A[i] < A[j] and i ≤ m) or (j > e) then
            B[k] = A[i]
            i = i + 1
        else
            B[k] = A[j]
            j = j + 1
        k = k + 1
    for k = 1 to e - s + 1 do
        A[s + k - 1] = B[k]
```

Descrição

O algoritmo ordena um vetor utilizando a estratégia "Dividir para Conquistar". Ele divide o vetor até ficar somente vetores unitários, nesse momento ele para de dividir e começar a reconstruir o vetor ordenando os elementos.

Complexidade

Este algoritmo só possui um caso, tendo em vista que, realiza sempre as mesmas operações de dividir e reconstruir o vetor. Sua complexidade é:

Função merge

O merge pode-se afirmar logo a primeira vista que ele é linear, pois nas linhas 4 e 12 os laços executam o número de vezes do tamanho do vetor mais uma para verificação. Já as instruções no seu interior executam no mais uma vez menos.

Complexidade linear: $O(n)$

mergeSort

Caso base (quando $s \geq e$)

$T(1) = c_1$

Equação de recorrência

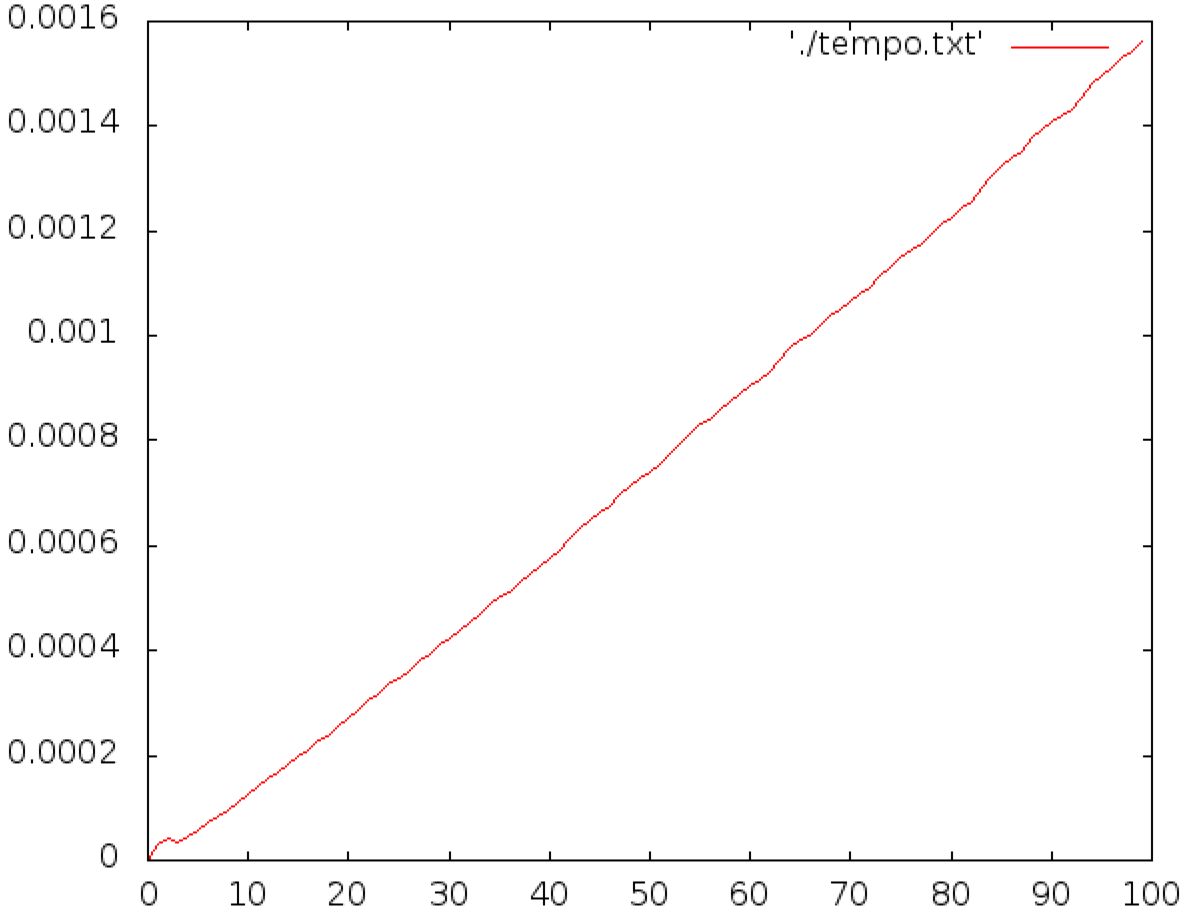
$$T(n) = \underbrace{b}_{\text{constante}} + \underbrace{T_m(n)}_{\text{tempo da função merge}} + \underbrace{2T(n/2)}_{\text{tempo das duas chamadas ao mergeSort. Como o mergeSort divide o vetor "n/2".}}$$

$$T(n) = b + T_m(n) + 2T(n/2)$$
$$= b + cn + 2T(n/2)$$

$$T(n/2) = b + c\frac{n}{2} + 2T(n/2^2)$$
$$T(n) = b + cn + 2\left[b + cn/2 + 2T(n/2^2)\right]$$
$$= 3b + 2cn + 2^2T(n/2^2)$$

$$T(n/2^2) = b + c\frac{n}{2^2} + 2T(n/2^3)$$
$$T(n) = 3b + 2cn + 2^2\left[b + c\frac{n}{2^2} + 2T(n/2^3)\right]$$
$$= 7b + 3cn + 2^3T(n/2^3)$$
$$= (2^x - 1)b + xcn + 2^xT(n/2^x)$$
$$= (2^{\log_2 n} - 1)b + (\log_2 n)cn + 2^{\log_2 n} c_1$$
$$= (n-1)b + cn\log_2 n + c_1 n$$

Complexidade linear: $O(n\log_2 n)$



0.0016
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
0
10
20
30
40
50
60
70
80
90
100
'./tempo.txt'

Algoritmo Distribution Sort

```
algoritmo distributionSort(A, n)
    a = minimo(A, n)
    b = maximo(A, n)
    B = zeros(b - a + 1)
    for i = 1 to n do
    |    B[A[i] - a + 1]++
    for i = 2 to b - a + 1 do
    |    B[i] = B[i] + B[i - 1]
    for i = 1 to n do
    |    g = B[A[i] - a + 1]
    |    C[g] = A[i]
    |    B[A[i] - a + 1]--
```

Descrição

Este algoritmo ordena um vetor utilizando vetores auxiliares sendo um deles um vetor de indices, ou seja, um vetor que armazena a posição que o elemento deve ficar no final do algoritmo.

Complexidade

Este algoritmo só possui um caso tendo em vista que sempre executa as mesmas instruções. Sua complexidade é:

A função minimo() busca o menor elemento em um vetor, para isso ela percorre todo o vetor comparando cada elemento. Seu tempo de execução é linear.

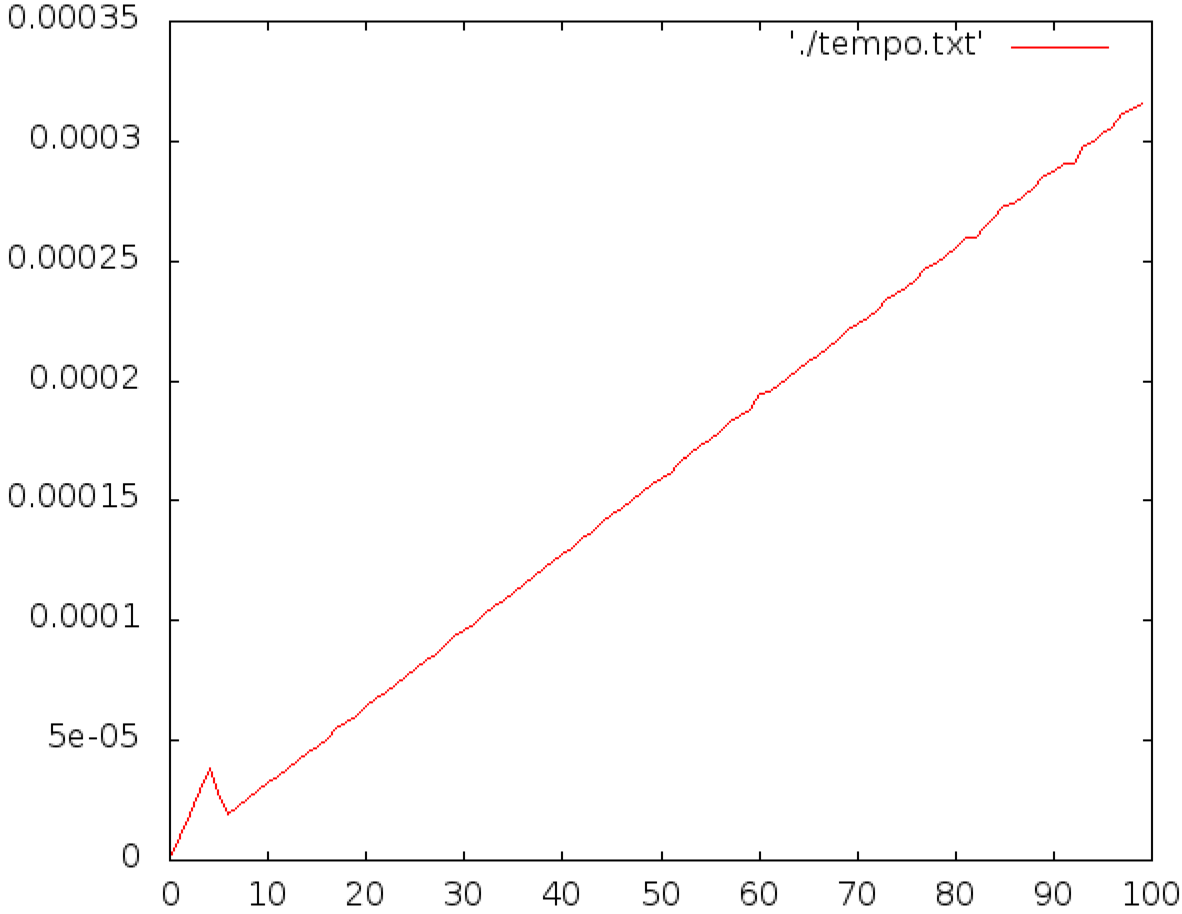
A função maximo() é parecida com a função minimo() sendo a única diferença que ela busca pelo maior elemento no vetor. Seu tempo de execução também é linear.

A função zero() preenche o vetor com zeros em todas as suas posições. Como percorre todo o vetor para isso, sua complexidade é linear. O tamanho do vetor que é passado para essa função é definido pela expressão, (maximo() - minimo() + 1) este tamanho irei chamar de K.

Nas linhas 4, 5, 8, 9, 10, 11 executam no máximo o número de vezes que o tamanho do vetor, já nas linha 6 e 7 executam no máximo K vezes.

Concluindo, todas as instruções deste algoritmo são de ordem linear.

Complexidade linear: $\Theta(n + K)$


0.00035
0.0003
0.00025
0.0002
0.00015
0.0001
5e-05
0
0 10 20 30 40 50 60 70 80 90 100
'./tempo.txt'

Algoritmo Insertion-sort

```
algoritmo insertionSort (A, n)
    for i=2 to n do
        k = A[i]
        j = i-1
        while j>0 and A[j] > k do
            A[j+1] = A[j]
            j = j-1
        A[j+1] = k
```

Onde, A é o vetor e n o tamanho do vetor.

Descrição

O algoritmo ordena um vetor utilizando a estratégia "Diminuir para conquistar". Ele ordena comparando de duas em duas posições, colocando os maiores elementos para a direita. Ele sempre percorre todo o vetor, se na primeira vez ele não atingir o objetivo de ordenar todo o vetor, ele pode retornar e realizar novas comparações até atingir seu objetivo.

Complexidade

O tempo despendido por esse algoritmo depende da entrada e de como os valores estão ordenados. O algoritmo possui três casos de complexidade que são:

Melhor caso: (quando o vetor já está ordenado)

$$T(n) = c_1 n + \overbrace{(c_2 + c_3 + c_4 + c_7)}^{\alpha}(n-1)$$

$$= (c_1 + \alpha)n + \alpha$$

Complexidade linear: $\Theta(n)$

Pior caso: (quando o vetor está ordenado inversamente)

$$T(n) = c_1 n + (c_2 + c_3)(n-1) + c_4\left(\frac{n(n-1)}{2} - 1\right) +$$

$$(c_5 + c_6)\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \overbrace{(c_4 + c_5 + c_6)}^{\beta} n^2 + \overbrace{\left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7\right)}^{\gamma} n$$

$$- \underbrace{(c_2 + c_3 + c_4 + c_7)}_{\alpha}$$

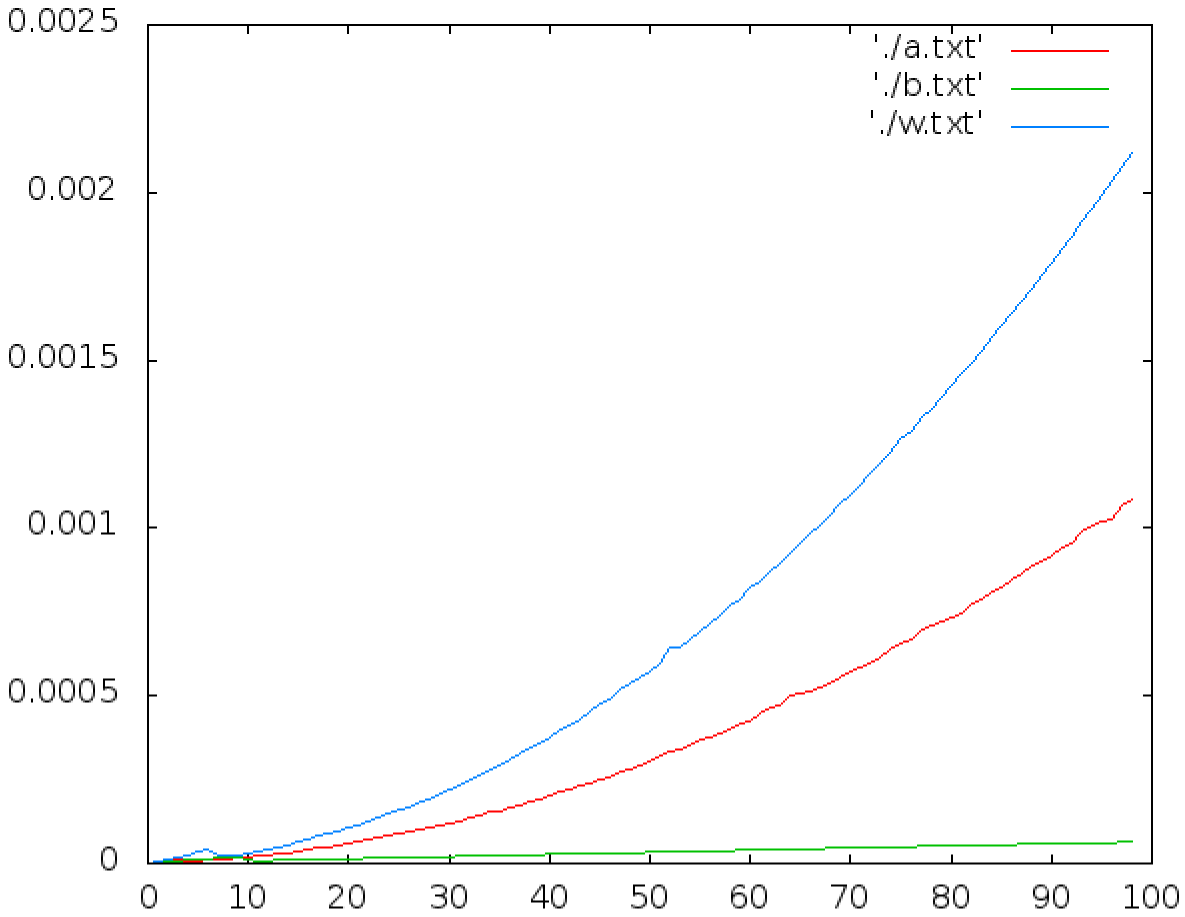
$$= \beta n^2 + \gamma n - \alpha$$

Complexidade Quadrada: $\Theta(n^2)$

Medio caso (quando o vetor está ordenado aleatoriamente)

Muitas vezes, o "caso médio" é quase tão ruim quanto o pior caso. Suponha um vetor com n elementos, ordenados aleatoriamente e que se queira descobrir o lugar no subarranjo $A[1 \dots j-1]$ em que o elemento $A[a]$ deva ser inserido. Em média, metade dos elementos de A são maiores que $A[j]$ e metade menores. Assim, em média, o tempo seria $t_a = n/2$. Isso continuará sendo uma função quadrática do tamanho da entrada, exatamente como no pior caso.

Complexidade Quadrada: $\Theta(n^2)$


0.0025
0.002
0.0015
0.001
0.0005
0
0
10
20
30
40
50
60
70
80
90
100
'./a.txt'
'./b.txt'
'./w.txt'

Algoritmo QuickSort

```
algoritmo quickSort(A, s, e)
    if e > s
        p = partition(A, s, e)
        quickSort(A, s, p-1)
        quickSort(A, p+1, e)
```

```
algoritmo partition(A, s, e)
    p = s
    j = s + 1
    for i = s+1 to e do
        if A[p] > A[i]
            j = j+1
            A[j-1] <=> A[i]
    A[p] <=> A[j-1]
    return j-1
```

Descrição

Este algoritmo ordena um vetor utilizando a estratégia "Dividir e conquistar". Ele ordena escolhendo um elemento chamado pivô onde divide o vetor a partir deste elemento.

Complexidade

Este algoritmo possui três casos de complexidade que se diferenciam pela forma que a função partition escolhe o pivô.

Este algoritmo é do tipo "in place" ou seja "no lugar" que não precisa de uma memória auxiliar para ordenar.

Partition

Esta função possui tempo de execução linear tendo de vista que uma instrução que executa mais vezes é a da linha 3, que no máximo irá executar o número de vezes do tamanho do vetor.

Complexidade linear: $\Theta(n)$

QuickSort

Médio Caso (quando o partition escolhe o pivô aleatoriamente)

Esse caso possui uma complexidade mais próxima do melhor caso, principalmente quando o vetor tem um tamanho elevado. Na medida que o vetor aumenta, fica mais difícil o pior caso ocorrer, já que a probabilidade de escolher o primeiro ou o último elemento do vetor, diminui a medida que ele aumenta.

Melhor caso (quando o partition escolhe o pivô na metade do vetor.)

$$T_b(n) = \underbrace{T_p(n)}_{cn} + 2\left(T_b\left(n/2\right)\right)$$

$$= cn + 2T_b(n/2)$$

$$T_b(n) = cn + 2c\left(n/2\right) + 4T\left(n/4\right)$$

$$T_b(n) = 2c\left(\frac{n}{2}\right) + 4c\left(\frac{n}{4}\right) + 8T\left(\frac{n}{8}\right)$$

$$T_b(n) = (\log_2 n)\, cn + 2^{\log_2 n} + T(1)$$

complexidade linear: $\Theta(n \log_2 n)$

Pior Caso (quando o partition escolhe o pivô no final do vetor.)

$$T_w(n) = \overbrace{T_p(n)}^{cn} + \overbrace{T_w(0)}^{0} + T_w(n-1)$$

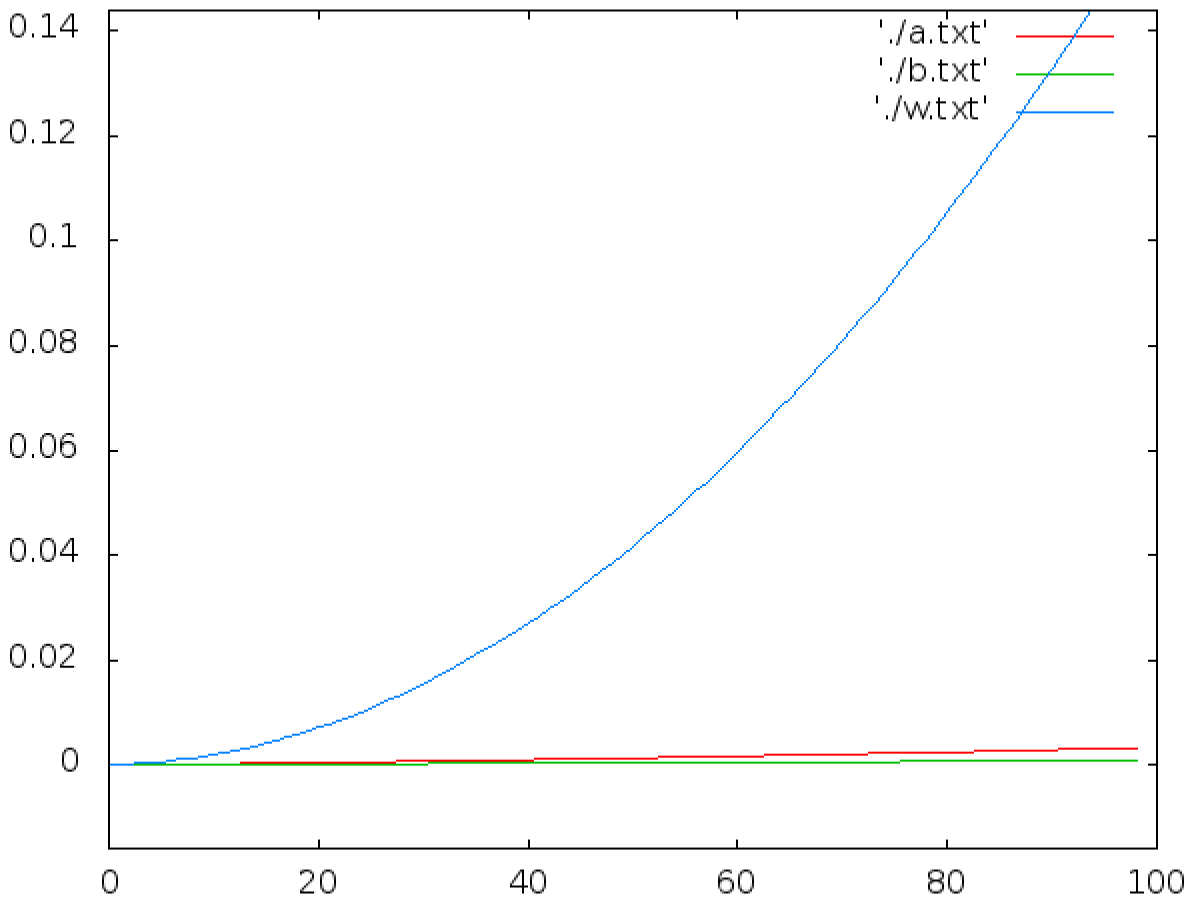
$$= cn + T_w(n-1)$$

$$T_w(n) = cn + c(n-1) + T_w(n-2)$$

$$T_w(n) = \sum_{i=1}^{n} c(n-1) + cn$$

$$T_w(n) = cn\frac{(n-1)}{2} + cn$$

Complexidade Quadrada: $\Theta(n^2)$


'./a.txt'
'./b.txt'
'./w.txt'
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80
100