Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

Primeiro trabalho prático – disponível em: http://www.dcc.ufmg.br/~alvaro/tp1alvaro.ps Código fonte disponível em: http://www.dcc.ufmg.br/~alvaro/tp1alvaro.c

Álvaro Rodrigues Pereira Júnior

Professor - Nivio Ziviani

Belo Horizonte
12 de abril de 2004

Sumário

1 Avaliação de somas

Este exercício consite na análise de um conjunto de somas, como segue.

1. $\sum_{i=1}^{n} a^i$

   De acordo com [8], podemos considerar S_n a soma da série:

   $$S_n = \sum_{i=1}^{n} a^i$$

   através da expansão da série:

   $$S_n = a + a^2 + a^3 + ... + a^{n-1} + a^n$$

   multiplicando cada lado por a:

   $$aS_n = a^2 + a^3 + ... + a^n + a^{n+1}$$

   pela subtração $aS_n - S_n$:

   $$aS_n - S_n = a^{n+1} - a$$

   colocando S_n em evidência:

   $$S_n(a-1) = a(a^n - 1)$$

   $$S_n = a\frac{a^n - 1}{a - 1}, \quad para\ a \neq 1$$

   no caso de $a = 1$, o valor de S_n é obtido através da soma do valor 1 n vezes:

   $$S_n = n$$

   para $n \to \infty$ a série converge:

   $$S_n = \frac{a}{1-a}$$

   para

   $$-1 < a < 1$$

2. $\sum_{i=1}^{n} i^2$

   De acordo com [4], vamos resolver o somatório através de expansão e contração. A técnica se baseia na idéia de escrever o somatório de maneira simples com forma fechada conhecida.

   devemos inicialmente escrever $\sum_{i=1}^{n} i^2$ através de um somatório duplo de i:

   $$\sum_{i=1}^{n} i^2 = \sum_{j=1}^{n} \sum_{i=j}^{n} i$$

por [2] sabemos que a forma fechada para o somatório $\sum_{x=1}^{m} x$ é:

$$\sum_{x=1}^{m} x = \frac{m(m+1)}{2}$$

usaremos S_n para denotar a forma fechada do somatório. Aplicando a relação anterior e considerando o deslocamento de limites [5], temos:

$$S_n = \sum_{j=1}^{n} \frac{n(n+1)}{2} - \frac{j(j-1)}{2}$$

$$S_n = \sum_{j=1}^{n} \frac{n^2 + n - j^2 + j}{2}$$

$$S_n = \frac{1}{2} \sum_{j=1}^{n} (n^2 + n - j^2 + j)$$

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

$$\frac{3}{2} S_n = \frac{1}{2}(n^3 + n^2) + \frac{1}{4} n(n+1)$$

$$6S_n = 2n^3 + 2n^2 + n^2 + n$$

$$S_n = \frac{2n^3 + 3n^2 + n}{6}$$

3. $\sum_{i=1}^{k} 2^{k-i} i^2$

Podemos reescrever o somatório da seguinte forma:

$$\sum_{i=1}^{k} 2^{k-i} i^2 = \sum_{i=1}^{k} \frac{2^k}{2^i} i^2$$

o termo constante 2^k pode ser retirado:

$$2^k \sum_{i=1}^{k} \frac{i^2}{2^i}$$

de acordo com [2], a razão entre os termos sucessivos $\sum_{i=1}^{\infty} \frac{i^2}{2^i}$ é:

$$\frac{(k+1)^2 / 2^{k+1}}{k^2 / 2^k} = \frac{(k+2)^2}{2k^2} \leq \frac{8}{9}$$

para $k \geq 3$.

portanto o somatório pode também ser dividido em:

$$\sum_{i=1}^{\infty} \frac{i^2}{2^i} = \sum_{k=0}^{2} \frac{i^2}{2^i} + \sum_{k=3}^{\infty} \frac{i^2}{2^i} \geq \sum_{k=0}^{2} \frac{i^2}{2^i} + \frac{8}{9} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k = O(1)$$

voltando à análise do somatório, como $\sum_{i=1}^{k} \frac{i^2}{2^i}$ é $O(1)$, podemos concluir que $2^k \sum_{i=1}^{k} \frac{i^2}{2^i}$ é $O(2^k)$.

4. $\sum_{i=1}^{n} \frac{1}{i}$

Para inteiros positivos n, o n-ésimo número harmônico é:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}$$

através da técnica de aproximação por integrais, podemos chegar a uma estimativa restrita para o n-ésimo número harmônico [2]. Para uma função monotonicamente decrescente, usamos:

$$\int_{m-1}^{n+1} f(x)dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)dx$$

como limite inferior temos:

$$\sum_{i=1}^{n} \frac{1}{i} \geq \int_{1}^{n+1} \frac{dx}{x}$$

$$\sum_{i=1}^{n} \frac{1}{i} \geq ln(n+1)$$

para o limite superior temos:

$$\sum_{i=2}^{n} \frac{1}{i} \leq \int_{1}^{n} \frac{dx}{x}$$

$$\sum_{i=2}^{n} \frac{1}{i} \leq ln(n)$$

$$\sum_{i=1}^{n} \frac{1}{i} \leq ln(n) + 1$$

portanto:

$$\sum_{i=1}^{n} \frac{1}{i} = ln(n) + O(1)$$

5. $\sum_{i=1}^{n} log(i)$

Considerando S_n a soma da série a ser avaliada:

$$S_n = \sum_{i=1}^{n} log(i)$$

$$S_n = log(1) + log(2) + log(3) + ... + log(n)$$

Usando a propriedade:

$$log(a) + log(b) = log(a * b)$$

temos:

$$S_n = log(1 * 2 * 3 * ... * n)$$

podemos observar então que:

$$S_n = log(n!)$$

pela aproximação de Stirling [2]:

$$log(n!) = \Theta(nlog(n))$$

Assim:

$$S_n = \Theta(nlog(n))$$

2 Complexidade de tempo

Este exercício consiste na análise de complexidade de tempo para alguns procedimentos, como segue.

1.
```
PROCEDURE FazAlgo ( n : integer);
VAR i, j, k, x : integer;
BEGIN
  x := 0;
  FOR i := 1 TO n - 2 DO
    FOR j := i + 1 TO n - 1 DO
      FOR k := 1 TO j DO x := x + 1;
END;
```

   Para o procedimento acima, temos três somatórios, como segue:

$$T(n) = \sum_{i=1}^{n-2} \left(\sum_{j=i+1}^{n-1} \left(\sum_{k=1}^{j} 1 \right) \right)$$

$$T(n) = \sum_{i=1}^{n-2} \left(\sum_{j=i+1}^{n-1} j \right)$$

   Fazendo:

$$T'(n) = \sum_{j=i+1}^{n-1} j$$

$$T'(n) = (i+1) + (i+2) + ... + (n-1)$$

   De [2], sabemos que:

$$\sum_{x=1}^{m} x = \frac{m(m+1)}{2}$$

   Logo,

$$T'(n) = \frac{(n-1)(n-1+1) - i(i+1)}{2}$$

$$T'(n) = \frac{n(n-1) - i(i+1)}{2}$$

$$T'(n) = \frac{n^2 - n}{2} - \frac{i^2}{2} - \frac{i}{2}$$

Substituindo T'(n) em T(n):

$$T(n) = \left(\frac{n^2 - n}{2}\right)\left(\sum_{i=1}^{n-2} 1\right) - \frac{1}{2}\left(\sum_{i=1}^{n-2} i^2\right) - \frac{1}{2}\left(\sum_{i=1}^{n-2} i\right)$$

De [6] sabemos que:

$$\sum_{x=1}^{m} x^2 = \frac{m(m+1)(2m+1)}{6}, para\ m \geq 0$$

Logo,

$$T(n) = \left(\frac{n^2 - n}{2}\right)(n-2) -$$

$$\frac{1}{2}\left(\frac{(n-2)((n-2)+1)(2(n-2)+1)}{6}\right) - \frac{1}{2}\left(\frac{(n-2)((n-2)+1)}{2}\right)$$

$$T(n) = \left(\frac{n^3 - 2n^2 - n^2 + 2n}{2}\right) -$$

$$\left(\frac{(n^2 - n - 2n + 2)(2n-3)}{12}\right) - \left(\frac{n^2 - n - 2n + 2}{4}\right)$$

$$T(n) = \left(\frac{n^3 - 3n^2 + 2n}{2}\right) -$$

$$\left(\frac{2n^3 - 2n^2 - 4n^2 + 4n - 3n^2 + 3n + 6n - 6}{12}\right) - \left(\frac{n^2 - 3n + 2}{4}\right)$$

$$T(n) = \left(\frac{n^3 - 3n^2 + 2n}{2}\right) -$$

$$\left(\frac{2n^3 - 9n^2 + 13n - 6}{12}\right) - \left(\frac{n^2 - 3n + 2}{4}\right)$$

$$T(n) = \frac{6n^3 - 18n^2 + 12n - 2n^3 + 9n^2 - 13n + 6 - 3n^2 + 9n - 6}{12}$$

$$T(n) = \frac{4n^3 - 12n^2 + 8n}{12}$$

$$T(n) = \frac{n^3 - 3n^2 + 2n}{3}$$

$$T(n) = \frac{n^3}{3} - n^2 + \frac{2n}{3}$$

$$T(n) = O(n^3)$$

2.
```
Pesquisa(n);
IF  n <= 1
THEN termine
ELSE BEGIN
     ordena os n elementos;
     { de alguma forma isto permite descartar 1/3 dos   }
     { elementos e fazer uma chamada recursiva no resto }
     Pesquisa (2n/3);
     END;
```
   Vamos considerar que a ordenação de n elementos utiliza um algoritmo de complexidade $O(n^2)$.

   Assim, a equação de recorrência para o procedimento acima é:

$$\begin{cases} T(1) = 0 & , \ \forall\ n = 1 \\ T(n) = n^2 + T(2n/3) & , \ \forall\ n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= \left(n^2\right) + T\left(\frac{2n}{3}\right) \\
T\left(\frac{2n}{3}\right) &= \left(\frac{2n}{3}\right)^2 + T\left((\frac{2}{3})^2 n\right) \\
T\left((\frac{2}{3})^2 n\right) &= \left((\frac{2}{3})^2 n\right)^2 + T\left((\frac{2}{3})^3 n\right) \\
&\vdots \\
T\left((\frac{2}{3})^{k-1} n\right) &= \left((\frac{2}{3})^{k-1} n\right)^2 + T\left((\frac{2}{3})^k n\right)
\end{aligned}$$

   Substituindo os termos temos:

$$T(n) = n^2 + (\frac{2n}{3})^2 + \left((\frac{2}{3})^2 n\right)^2 + \cdots + \left((\frac{2}{3})^{k-1} n\right)^2 + T\left((\frac{2}{3})^k n\right)$$

O que representa o seguinte somatório:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{k-1} \left((\frac{2}{3})^i n \right)^2 \\
T(n) &= n^2 \sum_{i=0}^{k-1} \left(\frac{4}{9} \right)^i
\end{aligned}
$$

Usando a seguinte fórmula [2]:

$$\sum_{x=0}^{m} a^x = \left(\frac{1 - a^{m+1}}{1 - a} \right)$$

Temos:

$$
\begin{aligned}
T(n) &= n^2 \left(\frac{1 - (\frac{4}{9})^{k-1+1}}{1 - \frac{4}{9}} \right) \\
T(n) &= \frac{9n^2}{5} - \frac{9n^2}{5} (\frac{2}{3})^{2k}
\end{aligned}
$$

Como $(\frac{2}{3})^k n = 1$, temos que $(\frac{2}{3})^k = \frac{1}{n}$. Logo, substituindo:

$$
\begin{aligned}
T(n) &= \frac{9n^2}{5} - \frac{9n^2}{5} \left(\frac{1}{n^2} \right) \\
T(n) &= \frac{9n^2}{5} - \frac{9}{5} = O(n^2)
\end{aligned}
$$

3.
```
PROCEDURE Sort (VAR A: ARRAY[1..n] OF integer; i, j: integer);
{ n uma potencia de 3 }
BEGIN
  IF i < j
  THEN BEGIN
       m := (i + j - 1)/3;
       Sort (A,i,m);
       Sort (A,m+1,j);
       Merge (A,i,m,j);
       { Merge intercala A[i..m] e A[m+1..j] em A[i..j] a um custo n}
       END;
END;
```

A equação de recorrência para o procedimento acima é:

$$
\begin{cases}
T(1) = 1 & , \forall\ n = 1 \\
T(n) = n + T(n/3) + T(2n/3) & , \forall\ n > 1
\end{cases}
$$

Vamos utilizar o método da árvore de recursão, conforme apresentado em [2]. A figura 1 mostra a árvore de recursão para T(n) = T(n/3) + T(2n/3) + n, com algumas derivações. Quando somamos os valores em todos os níveis da árvore de recursão, obtemos um valor n para cada nível. O caminho mais longo da raiz até uma folha n é:

$$n \longrightarrow (2/3)n \longrightarrow (2/3)^2 n \longrightarrow ... \longrightarrow 1$$

Tendo em vista que $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, a altura da árvore é $\log_{3/2} n$.

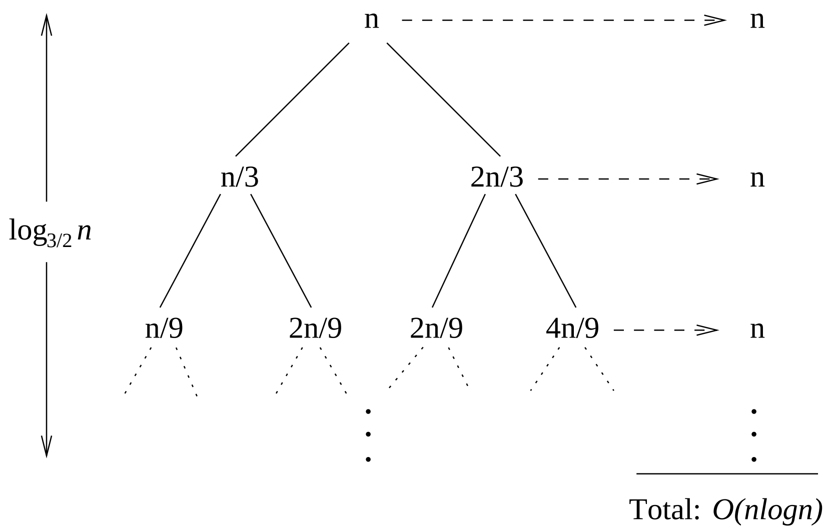

Figura 1: Árvore de recursão para a recorrência T(n) = T(n/3) + T(2n/3) + n.

Intuitivamente, esperamos que a recursão para a recorrência seja no máximo o número de níveis vezes o custo de cada nível, ou $O(nlog_{3/2}n) = O(nlogn)$. O custo total é distribuído de modo uniforme ao longo dos níveis da árvore de recursão. Ainda é necessário considerar o custo das folhas. Se essa árvore de recursão fosse uma árvore binária completa de altuar $\log_{3/2} n$, haveria $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ folhas. Como o custo de cada folha é uma constante, o custo total de todas as folhas seria então $\theta(n^{\log_{3/2} 2})$, que é $\omega(nlogn)$. No entanto, Essa árvore de recursão não é uma árvore binária completa, e assim ela tem menos que $n^{\log_{3/2} 2}$ folhas. Além disso, a medida que descemos a partir da raiz, mais e mais nós internos estão ausentes. Consequentemente, nem todos os níveis contribuem com um custo exatamente igual a n; os níveis em direção à parte inferior contribuem menos. Estas diferenças são toleráveis, pois estamos tentando que a suposição $O(nlogn)$ para o limite superior é correta.

Podemos usar o método de substituição para verificar que $O(nlogn)$ é um limite superior para a solução da recorrência. Mostramos que $T(n) \leq dnlogn$, onde d é uma constante positiva apropriada. Temos:

$$
\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + n \\
&\leq d(n/3)log(n/3) + d(2n/3)log(2n/3) + n \\
&= (d(n/3)logn - d(n/3)log3) + (d(2n/3)logn - d(2n/3)log(3/2)) + n \\
&= dnlogn - d((n/3)log3 + (2n/3)log(3/2)) + n \\
&= dnlogn - d((n/3)log3 + (2n/3)log3 - (2n/3)log2) + n \\
&= dnlogn - dn(log3 - 2n/3) + n \\
&\leq dnlogn,
\end{aligned}
$$

Desde que $d \geq 1/(log3 - (2/3))$. Desse modo, não tivemos que executar uma contabilidade de custos mais precisa na árvore de recursão.

4.
```
PROCEDURE D ( VAR L );
BEGIN
  IF |L| > 1
  THEN BEGIN
       Divide L em 3 partes iguais L1, L2, L3 com custo
       de |L| * |L| * |L| passos; {|L| ao cubo}
       D(L1);   D(L2);   D(L3);
       END;
END;
```

A equação de recorrência para o procedimento acima é:

$$
\begin{cases}
T(1) = 0 & , \ \forall \ n = 1 \\
T(n) = n^3 + 3T(n/3) & , \ \forall \ n > 1
\end{cases}
$$

Desta forma, temos:

$$
\begin{aligned}
T(n) &= n^3 + 3T\left(\frac{n}{3}\right) \\
3T\left(\frac{n}{3}\right) &= 3\left(\frac{n^3}{3^3}\right) + 3^2T\left(\frac{n}{3^2}\right) \\
3^2T\left(\frac{n}{3^2}\right) &= 3^2\left(\frac{n^3}{3^6}\right) + 3^3T\left(\frac{n}{3^3}\right) \\
&\vdots \\
3^{k-1}T\left(\frac{n}{3^{k-1}}\right) &= 3^{k-1}\left(\frac{n^3}{3^{3(k-1)}}\right) + 3^kT\left(\frac{n}{3^k}\right)
\end{aligned}
$$

Substituindo os termos temos:

$$
T(n) = n^3 + 3\left(\frac{n^3}{3^3}\right) + 3^2\left(\frac{n^3}{3^6}\right) + ... + 3^{k-1}\left(\frac{n^3}{3^{3(k-1)}}\right) + 3^kT\left(\frac{n}{3^k}\right)
$$

O que representa o seguinte somatório:

$$
\begin{aligned}
T(n) &= n^3 + \sum_{i=1}^{k-1} 3^i \left(\frac{1}{3}\right)^{3i} n^3 \\
T(n) &= n^3 + n^3 \sum_{i=1}^{k-1} 3^i . 3^{-3i} \\
T(n) &= n^3 + n^3 \sum_{i=1}^{k-1} \left(\frac{1}{3}\right)^{2i} \\
T(n) &= n^3 + n^3 \sum_{i=1}^{k-1} \left(\frac{1}{9}\right)^{i} \\
T(n) &= n^3 \sum_{i=0}^{k-1} \left(\frac{1}{9}\right)^{i}
\end{aligned}
$$

Usando a seguinte fórmula [2]:

$$
\sum_{x=0}^{m} a^x = \left(\frac{1-a^{m+1}}{1-a}\right)
$$

Temos:

$$
\begin{aligned}
T(n) &= n^3 \left(\frac{1-(\frac{1}{9})^{k-1+1}}{1-\frac{1}{9}}\right) \\
T(n) &= \frac{9n^3}{8} - \frac{9n^3}{8}(\frac{1}{9})^k \\
T(n) &= \frac{9n^3}{8} - \frac{9n^3}{8}(\frac{1}{3^{2k}})
\end{aligned}
$$

Como $(\frac{1}{3})^k n = 1$, temos que $(\frac{1}{3})^k = \frac{1}{n}$. Logo, substituindo:

$$
\begin{aligned}
T(n) &= \frac{9n^3}{8} - \frac{9n^3}{8}(\frac{1}{n^2}) \\
T(n) &= \frac{9n^3}{8} - \frac{9n}{8} = O(n^3)
\end{aligned}
$$

3 Limite inferior

1. **Apresente um algoritmo para encontrar o maior e o segundo maior elemento de um conjunto.**

   O algoritmo que será apresentado consite na comparação dois a dois dos elementos da lista e em chamadas recursivas. A figura 2 apresenta um exemplo das comparações que são realizadas. Para este exemplo, o primeiro elemento é comparado com o segundo, o terceiro com o quarto, e assim por diante. Na primeira comparação, como 17 é maior do que 3, 17 é selecionado e 3 é o perdedor. Na segunda comparação, 22 é ganhador e é selecionado. Em seguida, todos os elementos ganhadores compõem uma nova lista e são comparados dois a dois. Este processo ocorre até que a nova lista tenha apenas um elemento. Este elemento é necessariamente o maior elemento daquela lista. Para o exemplo da figura 2, o maior elemento é 87.

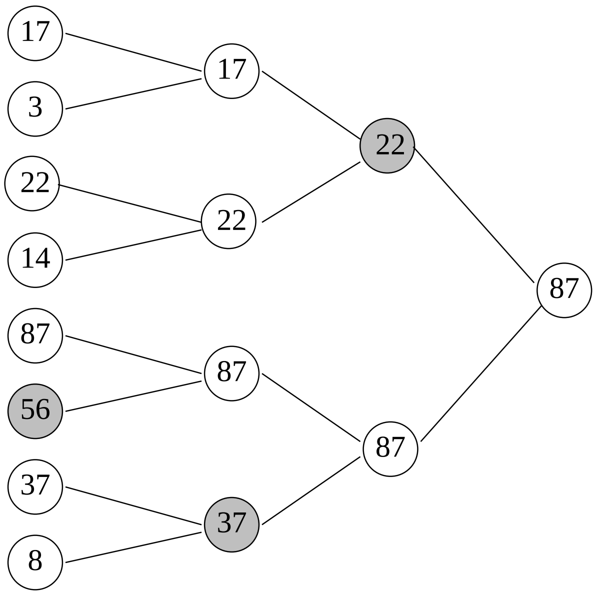

Figura 2: Exemplo de comparações realizadas para o algoritmo apresentado.

   O método apresentado acima é para a retirada do maior elemento da lista. Ainda é necessário retirar o segundo maior elemento. Necessariamente, o segundo maior elemento da lista foi, em algum momento, comparado ao maior elemento. Este fator é considerado e assim é possível recuperar o segundo maior elemento aplicando o mesmo processo descrito acima, para retirada do maior elemento. Neste momento a nova lista é composta por todos os elementos da lista principal que perderam para o maior elemento da lista principal. Assim, o maior elemento desta nova lista é recuperado, que é o segundo maior elemento da lista principal. O algoritmo para retirada do maior e segundo maior elementos de uma lista é apresentado na figura 3 e será detalhado a seguir.

   O algoritmo da figura 3 é apresentado utilizando duas funções: **Maior**, que utiliza o processo descrito acima para retirada do maior elemento da lista e **Maiores**, que chama a função **Maior** duas vezes – inicialmente, passando a lista completa de elementos, para retirada do maior elemento e posteriormente com uma nova lista, composta de todos os elementos que perderam para o ganhador da primeira chamada.

A função **Maior** funciona da seguinte maneira: inicialmente, os elementos da lista são comparados dois a dois, sempre armazenando as comparações que foram realizadas. É feita uma chamada recursiva à função **Maior**, com todos os elementos ganhadores. Desta forma, sempre a metade dos elementos da lista são novamente comparados. Quando a lista possuir somente um elemento, este é o maior elemento e a função retorna. A função deve retornar não só o maior elemento da lista como também uma nova lista contendo os elementos que perderam em comparações para o maior elemento. Esta nova lista é usada em uma nova chamada à função **Maior**, o que ocorre na função **Maiores**. O retorno desta segunda chamada é o maior elemento desta nova lista, que é o segundo maior elemento da lista principal. Também é retornado uma nova lista de perdedores, que não é utilizada.

```
FUNTION Maior ( VAR Lista(n) );
BEGIN
  IF n > 1
  THEN BEGIN
       Compare dois a dois os elementos da lista. Se
       n é ímpar, não compare o último elemento;
       Com a lista dos números maiores (ganhadores),
       chame recursivamente a função Maiores. Se n é
       ímpar, inclua o último elemento na nova lista;
       Mantenha a informação de qual número foi
       comparado com qual.
       END;
  ELSE BEGIN
       Retorne o ganhador (que é o único elemento da
       lista) e ainda a nova lista com os números que
       perderam do ganhador.
       END;
END;

FUNTION Maiores ( VAR Lista(n) );
BEGIN
       maiorElemento = Maior(Lista(n));
       segundoMaior = Maior(Lista($log_2n$));
END;
```

Figura 3: Algoritmo para retirada do maior e do segundo maior elementos de uma lista

O algoritmo é apresentado em alto nível e algumas situações específicas não são detalhadas. um exemplo é a situação onde a lista possui um número ímpar de elementos. Neste caso, o último elemento não é comparado e é inserido na nova lista que será enviada para a chamada recursiva. Outra situação não considerada no algoritmo apresentado é o armazenamento dos elementos que perderam nas comparações. O elemento perdedor de uma determinada comparação pode precisar ser recuperado caso o elemento na qual ele perdeu venha a ser o vencedor. Neste caso o elemento perdedor passa a compor a nova lista na nova chamada da função **Maior**.

2. **Seja $f(n)$ o número de comparações realizadas pelo seu algoritmo. Apresente a análise da complexidade de tempo do seu algoritmo.**

   Inicialmente será feita a análise de complexidade para a primeira chamada da função **Maior**. Conforme apresentado na questão anterior, como a função **Maior** é chamada pela segunda vez com os elementos perdedores do ganhador, ela é chamada com $log_2 n$ elementos. A equação de recorrência para este algoritmo recursivo é apresentada a seguir.

$$\begin{cases} T(1) = 0 & , \forall\ n = 1 \\ T(n) = n/2 + T(n/2) & , \forall\ n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= \left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) \\
T\left(\frac{n}{2}\right) &= \left(\frac{n}{2^2}\right) + T\left((\frac{n}{2^2})\right) \\
T\left(\frac{n}{2^2}\right) &= \left(\frac{n}{2^3}\right) + T\left((\frac{n}{2^3})\right) \\
&\vdots \\
T\left(\frac{n}{2^{k-1}}\right) &= \left(\frac{n}{2^k}\right) + T\left((\frac{n}{2^k})\right)
\end{aligned}$$

$$\begin{aligned}
T(n) &= \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k} + T\left(\frac{n}{2^k}\right) \\
T(n) &= \sum_{i=1}^{k} \left(\frac{n}{2}\right)^i \\
T(n) &= n\sum_{i=1}^{k} \left(\frac{1}{2}\right)^i
\end{aligned}$$

Usando a seguinte fórmula [2, 8]:

$$S_n = a\frac{a^n - 1}{a - 1}$$

Temos que:

$$T(n) = n\left(\frac{\frac{1}{2}\left(\frac{1}{2}^k - 1\right)}{\frac{1}{2} - 1}\right)$$

$$T(n) = n\left(1 - \left(\frac{1}{2}\right)^k\right)$$

Pela condição de parada $T(1) = 0$, temos que:

$$\left(\frac{1}{2}\right)^k = \frac{1}{n}$$

Substituindo, temos:

$$T(n) = n\left(1 - \frac{1}{n}\right)$$

$$T(n) = n - 1 = O(n)$$

Este é o valor para a primeira chamada da função **maior**. A segunda chamada da função é feita com $log_2 n$ elementos, que são os elementos que perderam para o vencedor. Logo, a análise de complexidade para todo o algoritmo é:

$$T(n) = (n-1) + (log_2 n - 1) = n + log_2 n - 2 = O(n)$$

3. **Determine o limite inferior para o problema de encontrar o maior e o segundo maior elemento de um conjunto.**

   As demonstrações realizadas para este exercício são baseadas em [3].

   Vamos considerar o **teorema**: "Nenhum algoritmo que encontre tanto o maior quanto o segundo maior elemento de um conjunto desordenado pode garantir uma performance de pior caso de c comparações, para $c < n + \lceil logn \rceil - 2$."

   Prova, usando a técnica do adversário:

   Vamos assumir que A é o maior elemento, B o segundo maior, e $(n-2)$ elementos menores que ele. Para qualquer algoritmo conhecer esta informação, ele certamente tem que saber que A é maior que B, senão poderíamos inverter os valores de A e B sem que o algoritmo soubesse, o que identificaria incorretamente o segundo maior elemento. Da mesma forma, o algoritmo precisaria ter identificado o grupo de $(n-2)$ elementos que são menores que B. Para verificar que todos estes elementos realmente estão abaixo de B, cada elemento deste conjunto tem que ter perdido uma comparação ou para B ou senão para algum outro membro do grupo (se isto não acontecesse, ele poderia ser o maior ou segundo maior elemento do grupo, o que não seria identificado e geraria um resultado incorreto). Isto significa que para qualquer entrada, um algoritmo tem que fazer pelo menos $n-2$ comparações na qual não envolva o maior elemento. Agora será mostrado que um adversário pode forçar qualquer algoritmo a fazer pelo menos $\lceil logn \rceil$ comparações que envolva o maior elemento e assim tal algoritmo deve executar pelo menos $n + \lceil logn \rceil - 2$ comparações no total.

Para fazer isto, vamos considerar o seguinte adversário. Dado um conjunto de respostas de comparações anteriores, vamos dizer se uma resposta a uma comparação é "conhecida" se aquela resposta é a mesma para todas as outras nas quais são consistentes com comparações anteriores. Nosso adversário irá manter, para cada elemento x, a lista $L(x)$ de elementos que são "conhecidos" por serem menor ou igual a x baseado em comparações anteriores. Inicialmente, $L(x) = \{x\}$ para todos os elementos, uma vez que a única coisa que sabemos é que x é menor ou igual a ele mesmo. Agora, se perguntado "a é menor ou igual a b?", se esta pergunta é conhecida, o adversário vai dar a resposta conhecida. Por outro lado, o adversário vai dizer que a vence a comparação exatamente quando $|L(a)| \geq |L(b)|$. Respondendo questões desta forma, podemos ter certeza de que existirá sempre pelo menos uma resposta que é consistente com todas as respostas do nosso usuário.

Usando este adversário, observamos que se um elemento x em particular estiver envolvido em exatamente k comparações, então $|L(x)| \leq 2^k$. Podemos provar isto por indução. Como um caso base, sabemos que quando $k = 0$, $|L(x)| = 1 = 2^0$. Pelo passo indutivo, assuma que $|L(x)| \leq 2^{k-1}$ depois de $k-1$ comparações envolvendo x e considermos a k-ésima comparação entre x e algum y. No caso em que x perde na comparação ou se já era "conhecido" que x era pelo menos tão grande quanto y, então nehum novo elemento é adicionado a $L(x)$, assim $|L(x)| \leq 2^{k-1} \leq 2^k$. Se a resposta não era previamente conhecida e o adversário responde que x ganha a comparação, então ele tem que ter sido o caso em que $|L(y)| \leq |L(x)|$. Se $L'(x) = L(x) \cup L(y)$. Mas estes são os únicos elementos que podem ser adicionadas a $L(x)$. Por outro lado, $|L'(x)| \leq |L(x)| + |L(y)|$ (alguns elementos de $L(y)$ podem já estar em $L(x)$). No entanto, por indução sabemos que $|L(x)| \leq 2^{k-1}$ e pelas regras do adversário, sabemos que $|L(y)| \leq |L(x)| \leq 2^{k-1}$ e que $|L'(x)| \leq 2^{k-1} + 2^{k-1} = 2^k$.

Finalmente, estamos prontos para mostrar que qualquer algoritmo tem que fazer pelo menos $\lceil logn \rceil$ comparações na qual envolva o maior elemento. Note que uma vez que o algortmos eventualmente saiba o maior elemento A, então devemos ter $L(A) = n$. Se A está envolvido em k comparações, sabemos que $n = L(A) \leq 2^k$ e então, rearranjando, temos $k \geq logn$. Como k deve ser integral, vemos que $k \geq \lceil logn \rceil$ e assim, o maior elemento está envolvido em pelo menos $\lceil logn \rceil$ comparações. Basendo no primeiro parágrafo, a prova está completa.

4. **O algoritmo apresentado no item (a) é ótimo?**

   Sim. A função de complexidade do algoritmo proposto é igual ao limite inferior do problema, como verificado nas duas questões acima.

   $f(n)$ **é um limite assintótico firme? Justifique sua resposta.**

   Para o algoritmo apresentado, $f(n) = n + log_2 n - 2$. $f(n)$ será um limite assintótico firme se $g(n) = \theta(f(n))$, onde $g(n) = n$. Isto irá ocorrer quando existirem constantes c_1, c_2 e m tais que, para todo $n \geq m$, a função $g(n)$ é igual a $g(n)$ a menos de uma constante. Assim, temos que verificar se $g(n) = \theta(f(n))$. Para isso, temos que obter constantes c_1, c_2 e m tais que:

   $$c_1 n \leq n + log_2 n - 2 \leq c_2 n$$

   para todo $n \geq m$. Dividindo por n leva a

   $$c_1 \leq 1 + \frac{log_2 n}{n} - \frac{2}{n} \leq c_2$$

   O lado esquerdo da desigualdade é válido para qualquer valor de $n \geq 4$ quando escolhemos $c_1 \leq 1$. Da mesma forma, escolhendo $c_2 \geq 2$, o lado direito da desigualdade é válido para qualquer $n \geq 4$. Como existem as constantes $c_1 = 1$, $c_2 = 2$ e $m = 4$, $f(n)$ é um limite assintótico firme.

4 Shellsort

O objetivo deste trabalho é realizar um estudo do algoritmo Shellsort de ordenação interna [8].

4.1 Introdução

Esta seção foi baseada em [8].

O Shellsort foi proposto por Shell [7] em 1959. É uma extensão do método da inserção. Enquanto este troca itens adjacentes quando está procurando o ponto de inserção na sequência destino, o Shellsort permite trocas de registros que estão distantes um do outro. Os itens que estão separados por h posições são rearranjados de forma que todo h-ésimo item leva a uma sequência ordenada. Várias sequências para h têm sido experimentadas. Uma sequência que já apresentou resultados satisfatórios em experimentos é a seguinte:

$$\begin{array}{ll} h(s) = 3h(s-1)+1, & para\ s > 1 \\ h(s) = 1, & para\ s = 1. \end{array}$$

A análise de complexidade do algoritmo ainda é um problema em aberto. Para a sequência apresentada acima existem duas conjeturas[1] tendo o número de comparação como medida de complexidade: $C(n) = O(n^{1,25})$ e $C(n) = O(n(\ln n)^2)$.

4.2 Algoritmo e implementação

Foi utilizado o algoritmo implementado em [8] para ordenação usando o método Shellsort. O algoritmo Shelsort utiliza a sequência apresentada na seção 4.1 e é mostrado na figura 4, em linguagem C. Por estar utilizando parte do código da literatura, não foi interessante a composição de um projeto contendo arquivos cabeçalhos específicos, *makefile* e rotinas bem divididas em arquivos específicos. Desta forma, todo o código encontra-se em somente um arquivo, que é apresentado no anexo 1. Programas em *batch* fazem chamadas aos executáveis, para simular sequências de execução.

Dois contadores estão sendo utilizados na função **Shellsort** da figura 4: **comparações** é usado para contar o número de comparações entre valores chaves são efetuadas em cada execução e **trocas** é usado para contar o número de trocas de registro que são efetuadas em cada execução. Além da função **Shellsort**, as funções **rand0a1** e **Permut** também foram importantes na execução dos experimentos. Elas têm a função de gerar uma lista de números aleatórios para serem ordenados nos experimentos.

[1] de acordo com o dicionário Aurélio da língua portuguesa, conjetura é 1. Opinião sem fundamento preciso. 2. Suposição.

```
typedef struct Item {
  ChaveTipo Chave;
} Item;

void Shellsort(Item *A, Indice *n)
{
  int i, j;
  int h = 1;
  Item x;
  do h = h * 3 + 1;  while (h < *n);
  do
    {
      h /= 3;
      for (i = h + 1; i <= *n; i++)
        {
          x = A[i];
          j = i;
          while ((comparacoes++) && (A[j - h].Chave > x.Chave))
            {
              A[j] = A[j - h];
              trocas++;
              j -= h;
              if (j <= h)
                  goto L999;
            }
          L999: {
                  A[j] = x;
                  trocas++;
                 }
        }
    } while (h != 1);
  comparacoes--;
}
```

Figura 4: Algoritmo Shellsort utilizado.

4.3 Número de comparações e movimentos de registros

Determine experimentalmente o número esperado de (i) comparações e (ii) movimento-de-registros. Utilize arquivos de diferentes tamanhos com chaves geradas randomicamente. Use o programa *Permut.c* para obter uma permutação randômica de n valores ([8] pag. 427). Repita cada experimento algumas vezes e obtenha a média para cada medida de complexidade.

Os experimentos foram realizados da seguinte forma: oito tamanhos diferentes de listas de números aleatoriamente desordenados foram utilizados: contendo 50, 100, 500, 1.000, 10.000, 100.000, 1.000.000 (um milhão) e 10.000.000 (dez milhões) de elementos. Para cada um destes tamanhos de conjuntos foram realizadas *cem* execuções. Em todas as execuções eram computados o número total de comparações entre chaves e número total de trocas entre registros, conforme contadores da função da figura 4.

Os tempos de execução (tempo de relógio) foram bem pequenos para as cem execuções do tamanhos de conjuntos até 100.000 elementos. Até este tamanho, todas as cem execuções de cada um dos seis tamanhos levaram, juntas, cerca de 50 segundos. Para as cem execuções para o tamanho de conjunto de um milhão de elementos, cerca de 8 minutos e para as cem execuções para o tamanho de conjunto sendo dez milhões, cerca de três horas. Os experimentos foram realizados em um computador com processador K6II, com 500MHz de frequência e 256KB de memória principal. A tabela 1 apresenta as médias e os desvios padrão para os oito tamanhos diferentes experimentados, para as duas medidas de complexidade citadas.

Tabela 1: Diferentes tamanhos de conjuntos experimentados

Tamanho	**Comparações**		**Trocas de registros**	
	média	**desvio**	**média**	**desvio**
50	284,99	6,18	303,41	6,01
100	721,73	4,59	763,54	4,47
500	5878,70	3,16	6081,83	3,07
1.000	13879,64	2,95	14314,41	2,86
10.000	235092,32	2,91	239436,58	2,85
100.000	3879139,83	3,62	3920492,37	3,59
1.000.000	64600360,85	3,26	65009645,67	3,24
10.000.000	1091897712,54	3,61	1096040016,42	3,59

Verificamos pelo gráfico da figura 1 que o número médio de comparações é sempre menor que o número médio de trocas realizadas. Esta relação será melhor avaliada na próxima seção.

4.4 Comparação dos resultados

Dê a sua interpretação dos dados obtidos, comparando-os com as conjeturas relatadas em ([8] pag. 102).

O gráfico da figura 5 apresenta os valores dos desvios padrão para todos os tamanhos de conjuntos experimentados, tanto para o número de comparações quanto para o número de trocas de registros. O desvio padrão é uma função da distância entre os valores da amostra. Um desvio padrão maior significa que mais execuções ficaram distantes da média. Para as duas medidas de complexidade analisadas, os menores desvios ficaram para tamanhos nem tão grandes nem tão pequenos de conjuntos. O menor desvio para as duas medidas de complexidade avaliadas é para um conjunto de 10.000 elementos, como podemos verificar pela tabela 1 e pelo gráfico da figura 5.

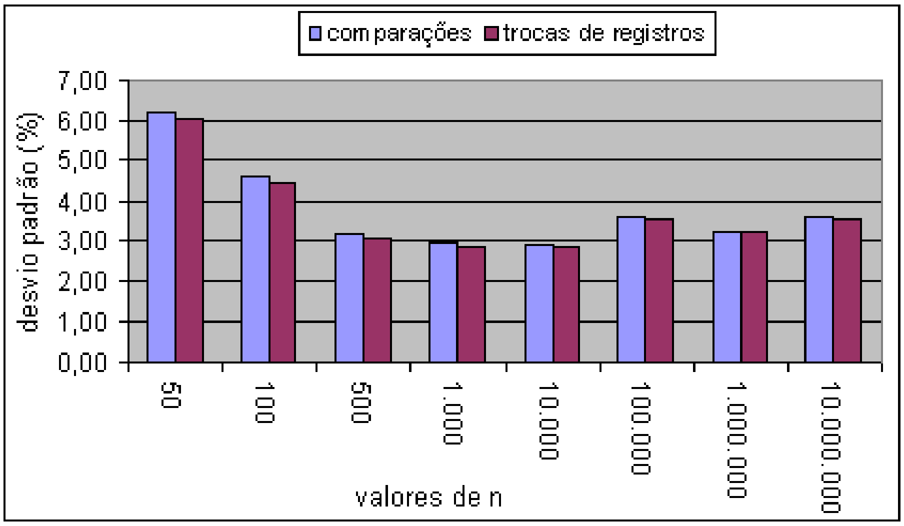

Figura 5: Valores dos desvios padrão para comparações e números de registros.

O gráfico da figura 6 mostra a diferença relativa entre a média do número de comparações e de trocas de registros, em relação ao número de comparações. Assim, o gráfico apresenta o quanto cada valor médio de trocas de registros está acima do valor médio de comparações. Analisando o gráfico, verificamos que a diferença relativa vem diminuindo a medida em que o tamanho do conjunto aumenta. Desta forma, os números de comparações e trocas de registros tendem a se aproximarem a medida em que o conjunto aumenta. Como a diferença relativa é sempre baixa e não varia muito quando o conjunto aumenta, verificamos que ambas as medidas de complexidade devem ter funções de complexidade similares.

O gráfico da figura 7 mostra a razão absoluta entre o número de comparações e o tamanho do conjunto e entre o número de trocas de registros e o tamanho do conjunto. Verificamos que as duas razões analisadas ficam bem próximas. Isto nos permite inferir que ambas as medidas de complexidade crescem na mesma proporção, com o crescimento do tamanho do conjunto de elementos utilizados. Portanto, as medidas de complexidade avaliadas, número de comparações e número de trocas de registros possuem características semelhantes. Isto reforça a idéia de que as funções de complexidade para as duas medidas são bem similares.

Conforme apresentado na seção 4.1, duas conjeturas estão presentes em [8] para a complexidade do número de comparações: $C_1(n) = O(n^{1,25})$ e $C_2(n) = O(n(\ln n)^2)$. Iremos agora comparar as conjeturas com os valores experimentais encontrados. Iremos considerar as funções de complexidade $f_1(n) = n^{1,25}$ e $f_2(n) = n(\ln n)^2$ para

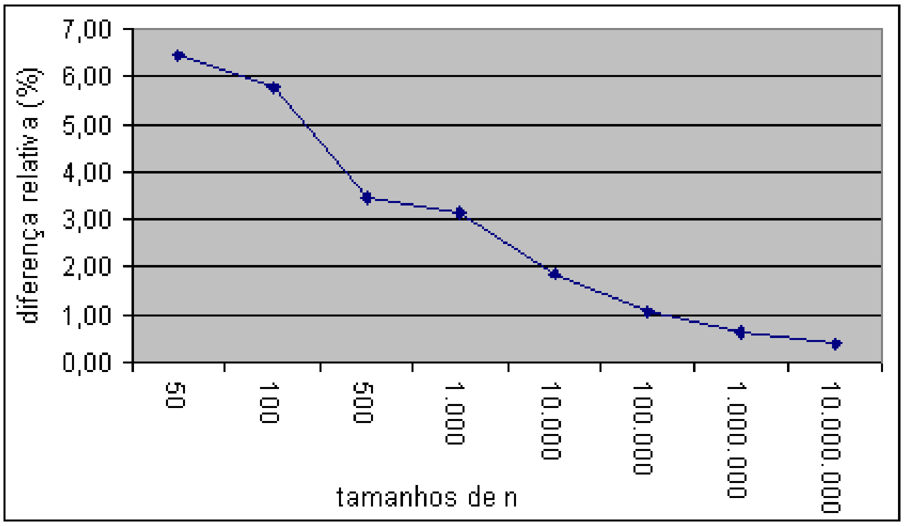

Figura 6: Diferença relativa entre o número de comparações e trocas de registros.

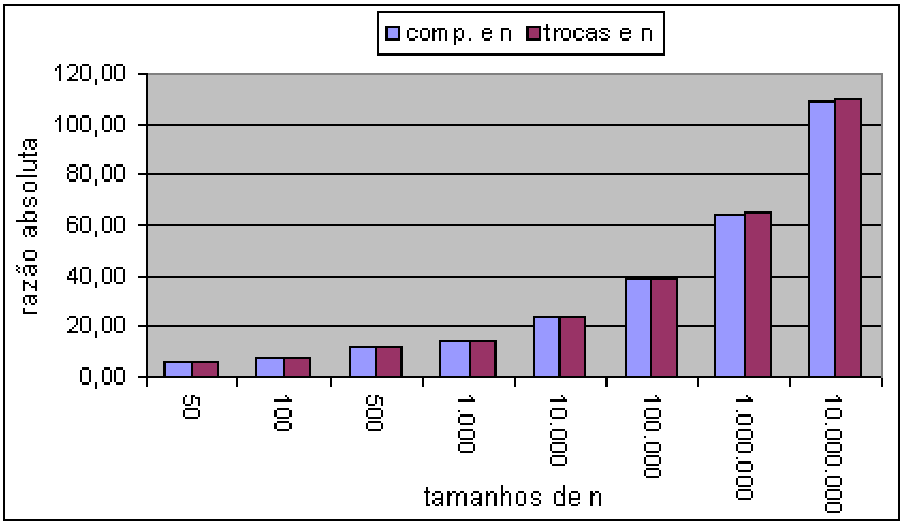

Figura 7: Razão entre o número de comparações e trocas de registros em relação ao tamanho do conjunto.

as duas conjeturas apresentadas. A tabela 2 apresenta os valores de $f_1(n)$ e $f_2(n)$ para os diferentes tamanhos de conjuntos (n). Como pode ser verificado, os valores estão relacionados com as médias encontradas nos experimentos realizados, o que será analisado.

Tabela 2: Número de comparações para as diferentes conjeturas.

Tamanho	$f_1(n) = n^{1,25}$	$f_2(n) = n(\ln n)^2$
50	132,96	765,20
100	316,23	2120,76
500	2364,35	19310,68
1.000	5623,41	47717,08
10.000	100000,00	848303,70
100.000	1778279,41	13254745,28
1.000.000	31622776,60	190868331,98
10.000.000	562341325,19	2597930074,13

O gráfico da figura 8 apresenta a razão absoluta entre a função da conjetura C_1 e o número médio de comparações efetuadas para o algoritmo experimentado e ainda a razão absoluta entre o número médio de comparações e a função da conjetura C_2. Neste caso, o que queremos verificar é qual função de cada conjetura está mais relacionada com as médias dos diferentes tamanhos de conjuntos experimentados.

De acordo com o gráfico verificamos que a conjetura C_1 está mais próxima da complexidade do algoritmo utilizado, para a sequência h utilizada. Isto é verificado porque a linha relacionada a C_1 varia menos no eixo y do que C_2. Ou seja, variando o tamanho do conjunto, as médias obtidas pelo algoritmo continuam proporcionais à função C_1. De fato, para a função representando a conjetura C_2, esta variação não é grande, de forma que também é uma boa conjetura para a complexidade do algoritmo, para a sequência h utilizada.

Uma outra forma de verificar que C_1 é uma conjetura mais próxima do algoritmo utilizado do que C_2 é analisar o desvio padrão dos valores que foram plotados no gráfico. O desvio padrão representa a distância dos valores, que deveriam estar mais próximos como explicado anteriormente. O desvio padrão para os valores da razão de f_1 mostrada no gráfico da figura 8 é $0,20$ e para f_2 é $0,42$, reforçando a teoria apresentada no parágrafo anterior.

Como f_1 e f_2 são funções que representam bem os resultados encontrados no algoritmo para diferentes tamanhos de conjuntos para a medida de comparação, seria possível encontrar constantes c_1 e c_2 que, multiplicadas pelos valores encontrados para as funções f_1 e f_2, respectivamente, sejam muito próximas do número real de comparações encontrados pelo algoritmo, para cada uma das execuções realizadas.

4.5 Outros trabalhos

Procure trabalhos relacionados na literatura com resultados analíticos (se houver) e descreva sucintamente esses trabalhos. Como os resultados experimentais que você obteve comparam com os resultados analíticos encontrados na literatura?

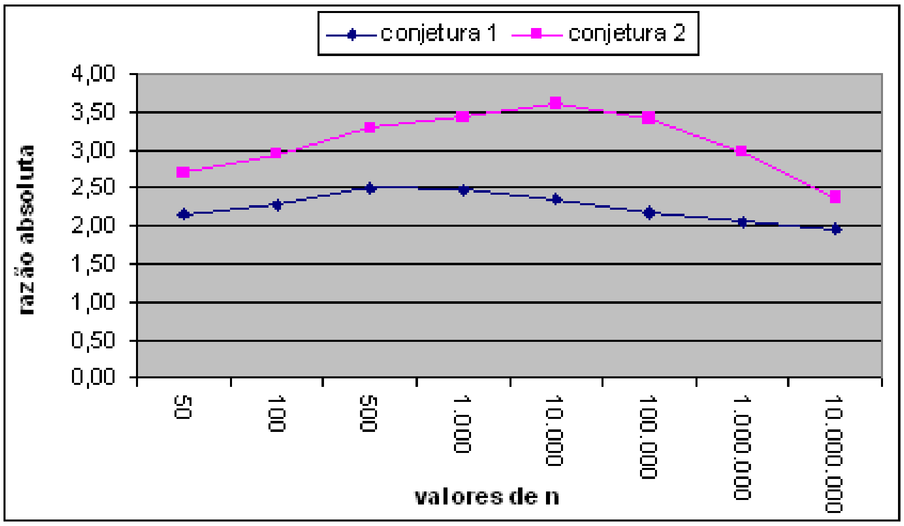

Figura 8:

Como já apresentado neste documento, o algoritmo **Shellsort** foi apresentado por Shell [7], em 1959. A partir daí, muitos trabalhos já foram realizados no sentido de modificar o algoritmo, propor novas sequências de incremento e ainda medir e avaliar a complexidade do algoritmo, problema em aberto até o momento. A seguir dois trabalhos serão apresentados de forma sucinta.

Tao Jiang, Ming Li e Paul Vitányi demonstraram um limite inferior para o caso médio do algoritmo Shellsort. Foi o primeiro limite inferior não trivial geral para o caso médio do Shellsort, quase 40 anos depois de Shell apresentar o algoritmo [7]. O trabalho consiste de demonstrações e não possui resultados analíticos.

Marcin Ciura [1] apresenta os resultados de estudos baseados em análises sequenciais para encontrar sequências de incrementos buscando minimizar o tempo médio de execução do algoritmo Shellsort, para conjuntos de até milhares de elementos. A sequência proposta obteve uma performance 3% melhor do que a melhor conhecida até então. Ele apresenta evidências (e não provas) de que a nova sequência proposta é ótima. Este foi o único trabalho encontrado que apresenta o número de comparações para a série de Knuth [6], a mesma utilizada no algoritmo implementado, e citada em [8]. Desta forma, iremos comparar o resultado do experimento de Marcin [1] com os resultados retornados pelo algoritmo utilizado, e esperamos obter valores aproximados. Os tamanhos de conjuntos utilizados em [1] foram de 40, 400 e 4.000 elementos. Por este motivo o algoritmo foi novamente executado para estes novos valores, cem vezes cada, para obtermos a média.

A tabela 3 apresenta as médias para a sequência de Knuth [6] (1, 4, 13, 40, 121, 364, 1093) utilizada neste trabalho (sequência 1), utilizadas por Marcin em [1] (sequência 2) e ainda para a sequência (1, 4, 10, 23, 57, 132, 301, 701, 1750) utilizada por Martin em [1] (sequência 3).

Analisando a tabela 3, verificamos que, como era de se esperar, os valores para as sequências 1 e 2 estão bem próximos. Isto é explicado pelo fato de que utilizam a mesma sequência [6]. No entanto, percebemos que os resultados médios para a nova sequência (saequência 3) estão próximos da sequência utilizada neste trabalho mas um pouco melhor. A sequência 3 obteve melhores resultados do que a outra

Tabela 3: Diferentes tamanhos de conjuntos experimentados

Tamanho	1, 4, 13, 40, 121, 364, 1093		1, 4, 10, 23, 57, 132, 301, 701, 1750
	sequência 1	sequência 2	sequência 3
40	205,7	208.8	211.5
400	4416	4413	4274
4.000	76749	74620	66370

sequência para um conjunto de 400 e 4.000 elementos, somente perdendo para o tamanho mais baixo avaliado. Assim, verificamos um exemplo de uma sequência com performance melhor que a utilizada neste trabalho.

Referências

[1] Marcin Ciura. Best increments for the average case of shellsort. In *13th International Symposium on Fundamentals of Computation Theory*, pages 106–117, 2001.

[2] Thomas H. Cormen, Charles L. Leiserson, Ronald L. Rivest, and Clifford Stein. *Algoritmos: Teoria e Prática*. Campus, 2002.

[3] Michael Goldwasser. Design and analysis of computer algorithms. http://www.cs.luc.edu/mhg/courses/loyola/comp363/2003_Spring/handouts/selectionproofs.pdf, março 2004.

[4] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Matemática Concreta*. LTC - Livros Técnicos e Científicos, 1995.

[5] Paul Walton Purdom Jr. *The Analysis of Algorithms*. CBS College Publishing, 1984.

[6] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 3rd edition, 1997.

[7] D. L. Shell. A highspeed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[8] Nivio Ziviani. *Projeto de Algoritmos com Implementações em Pascal e C*. Pioneira Thomson, second edition, 2004.

Anexo 1 – Código fonte do programa utilizado

```
#include   <limits.h>
#include   <sys/time.h>
#include   <stdlib.h>
#define MaxTam          1000

typedef long ChaveTipo;
typedef struct Item {
  ChaveTipo Chave;
} Item;

typedef int Indice;
typedef Item Vetor[MaxTam + 1];
Vetor A;
Indice i, n;

long int comparacoes = 1;
long int trocas = 0;

void Shellsort(Item *A, Indice *n)
{
  int i, j;
  int h = 1;
  Item x;
  do h = h * 3 + 1;  while (h < *n);
  do
    {
      h /= 3;
      for (i = h + 1; i <= *n; i++)
        {
          x = A[i];
          j = i;
          while ((comparacoes++) && (A[j - h].Chave > x.Chave))
            {
              A[j] = A[j - h];
              trocas++;
              j -= h;
              if (j <= h)
                  goto L999;
            }
          L999: {
                  A[j] = x;
                  trocas++;
                }
        }
```

```
    } while (h != 1);
  comparacoes--;
}

void Copia(Item *Fonte, Item *Destino, Indice *n)
{ for (i = 1; i <= *n; i++)
    Destino[i] = Fonte[i];
}

void Testa(Item *V, Indice *n)
{ for (i = 2; i <= *n; i++) {
    if (V[i].Chave < V[i-1].Chave) {
      printf("ERRO: ");
      //Imprime(V, n);
      return;
    }
  }
}

double rand0a1() {
  double resultado=  (double) rand()/ INT_MAX; /* Dividir pelo maior inteiro */
  if(resultado>1.0) resultado= 1.0;
  return resultado;
}

void Permut( Item *A, int n) {
 int i,j;
 Item b;

 for(i = n-1; i>0; i --) {
   j = (i * rand0a1()) +1 ;
   b = A[i];
   A[i] = A[j];
   A[j] = b;
 }
}

int main(int argc, char *argv[])
{
  Vetor B;
  n = MaxTam;   /*Tamanho do arranjo a ser ordenado*/

  struct timeval sem;
  gettimeofday(&sem,NULL);
  srand((int)(sem.tv_sec + 1000000*sem.tv_usec));

  for (i = 1; i <= n; i++)
```

```
    A[i].Chave = i;

  Permut (A,n);
  Copia (A,B,&n);

  Shellsort(B, &n);
  Testa(B, &n);
  Copia(A, B, &n);
  printf("%8ld\t%8ld\n", comparacoes, trocas);

  return 0;
}
```