Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

Primeiro Trabalho: disponível em: http://www.dcc.ufmg.br/~jnacif/paa

José Augusto Miranda Nacif

Professor - Nivio Ziviani

Belo Horizonte
7 de abril de 2004

Sumário

1 Avaliar as somas

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

   Segundo [10], pode-se considerar S_n a soma desta 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-se cada lado por a:

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

   Através da subtração $aS_n - S_n$:

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

   Colocando-se S_n em evidência:

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

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

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

   Em [4] pode-se encontrar várias formas de se resolver este somatório. Uma forma interessante é a resolução através de expansão e contração. Esta técnica de resolução de somatórios se baseia na idéia de escrever o somatório de maneira simples com forma fechada conhecida.

   Primeiramente deve-se escrever $\sum_{i=1}^{n} i^2$ através de um somatório duplo de i:

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

A forma fechada para o somatório $\sum_{x=1}^{m} x$ é amplamente conhecida [1]:

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

A notação S_n será usada para denotar a forma fechada do somatório. Aplicando-se a relação anterior e levando-se em consideração o deslocamento de limites [6], tem-se:

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

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

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

Removendo o segundo somatório:

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

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

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

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

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

Este somatório pode ser reescrito 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 então ser retirado:

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

A razão entre os termos sucessivos $\sum_{i=1}^{\infty} \frac{i^2}{2^i}$ é [1]:

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

se $k \geq 3$.

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 $O(1)$, pode-se 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, pode-se chegar a uma estimativa restrita para o n-ésimo número harmônico[1]. Para uma função monotonicamente decrescente, usa-se:

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

   Como limite inferior tem-se:

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

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

   Para o limite superior tem-se:

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

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

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

   Portanto:

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

5. $\sum_{i=1}^{n} log(i)$
Considere-se 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-se a propriedade:

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

Tem-se:

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

Pode-se observar então que:

$$S_n = log(n!)$$

Pela aproximação de Stirling [1]:

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

Portanto:

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

2 Apresente a complexidade de tempo para os procedimentos

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

Os três laços do trecho de código acima podem ser representados pelos seguintes somatórios:

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

Observe que o limite superior do primeiro laço deve ser $n - 2$ e não n como está no trecho de código pois o limite superior do segundo laço é $n - 1$.

Retirando o terceiro somatório da expressão tem-se:

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

O segundo somatório deve ser resolvido:

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

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

A forma fechada para este somatório é [10, 1]:

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

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

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

Voltando à equação principal:

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

Outra forma fechada conhecida é [6]:

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

para $m \geq 0$.

Substituindo, tem-se:

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

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

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

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

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

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

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

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;
```

Como não foi especificado qual é o algoritmo de ordenação usado, será arbitrado que este algoritmo é $O(n^2)$. O procedimento acima pode ser representado pela seguinte equação de recorrência:

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

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

Através da substituição dos termos:

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

$$T(n) = \sum_{i=0}^{k-1} \left(\left(\frac{2}{3}\right)^i n\right)^2$$

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

Sabe-se que [1]:

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

$$T(n) = n^2 \left(\frac{1 - (\frac{4}{9})^k}{1 - \frac{4}{9}}\right)$$

$$T(n) = \frac{9n^2}{5} - \frac{9n^2}{5}\left(\frac{2}{3}\right)^{2k}$$

Através da substituição de $(\frac{2}{3})^k = \frac{1}{n}$ tem-se:

$$T(n) = \frac{9n^2}{5} - \frac{9n^2}{5}\left(\frac{1}{n^2}\right)$$

$$T(n) = \frac{9n^2}{5} - \frac{9}{5}$$

$$T(n) = O(n^2)$$

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 custo n}
  END;
END;
```

O procedimento acima apresenta a seguinte equação de recorrência:

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

A melhor forma de resolver este tipo de equação de recorrência é através do método da árvore de recursão [1]. A Figura 1 apresenta a árvore de recursão para a recorrência acima. Após a soma dos valores em todos os níveis da árvore de recursão, tem-se um valor n para cada nível. Pode-se afirmar que o caminho mais longo da raiz até uma folha n é:

$$n, (2/3)n, (2/3)^2 n, ..., 1$$

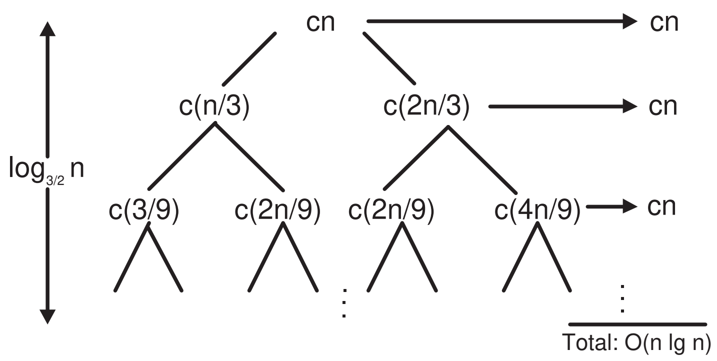

Figura 1: Árvore de recursão para a recorrência $T(n) = T(n/3) + T(2n/3) + cn$

A altura da árvore é $\log_{3/2} n$ pois $(2/3)^k n = 1$ quando $k = \log_{3/2} n$.

A recursão para esta recorrência deve ser de no máximo o número de níveis $*$ custo de cada nível, ou seja:

$$O(nlog_{3/2}n) = O(nlogn)$$

Deve-se ainda considerar o custo associado às folhas. Considerando que esta árvore de recursão seja uma árvore binária completa com altura $\log_{3/2} n$, tem-se $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ folhas. Como o custo de cada folha é constante, o custo total de folhas é então $\theta(n^{\log_{3/2} 2})$, que é $\omega(nlogn)$. Mas, esta árvore de recursão não é uma árvore binária completa, portanto ela possui menos do que $n^{\log_{3/2} 2}$ folhas. Deve-se observar também que a partir que descemos da raiz, mais nós internos estão ausentes e portanto nem todos os níveis contribuem com um custo igual a n. Para mostrar que $O(nlogn)$ é um limite superior para esta equação de recorrência estas diferenças são toleráveis.

Usando o método da substituição pode-se mostrar que $O(nlogn)$ é um limite superior para esta recorrência. Para isso deve-se mostrar que $T(n) \leq dnlogn$ (d = constante positiva).

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

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;
```

O procedimento acima apresenta a seguinte equação de recorrência:

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

Tem-se então:

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

Substituindo:

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

Que pode também ser escrito na forma do somatório:

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

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

Este somatório apresenta seguinte forma fechada [1, 10]:

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

$$T(n) = n^3 \left(\frac{1-(\frac{1}{9})^{k-1+1}}{1-\frac{1}{9}}\right)$$

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

Através da substituição de $(\frac{1}{3})^k = \frac{1}{n}$ tem-se:

$$T(n) = \frac{9n^3}{8} - \frac{9n^3}{8}\left(\frac{1}{n^2}\right)$$

$$T(n) = \frac{9n^3}{8} - \frac{9n}{8}$$

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

3 Limite inferior

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

```
procedure Maior_2Maior (var A: Vetor; var Max, 2Max: integer);
  var B: Vetor;
  var i: integer;
  n := Número de elementos de A
  while n > 1 do
  begin
    n := n / 2
    for i := 1 to n do
    begin
      if A[2i] > A[2i-1] then
        begin
          A[i] := A[2i]
          Armazena dados do perdedor( B );
        end;
      else
        begin
```

```
          A[i] := A[2i-1]
          Armazena dados do perdedor( B );
        end;
          if (n mod 2 > 0 and i = n) do A[i+1] = A[2i+1]
    end;
    Max := A[1]
  end;
  A := Logn elementos que perderam para o ganhador ( B )
  n := Número de elementos de A

  while n > 1 do
    begin
      n := n / 2
      for i := 1 to n do
      begin
        if A[2i] > A[2i-1] then
          begin
            A[i] := A[2i]
          end;
        else
          begin
            A[i] := A[2i-1]
          end;
            if (n mod 2 > 0 and i = n) do A[i+1] = A[2i+1]
      end;
      2Max := A[1]
    end;
end;
```

Um problema similar foi descrito pela primeira vez por [2], que consistia em determinar o campeão e o vice-campeão em um torneio de tênis. Por isso este algoritmo é conhecido na literatura como algoritmo do torneio. O primeiro laço (while) calcula o maior elemento do vetor de forma recursiva. O segundo laço (while) calcula o maior elemento do vetor constituído pelos logn elementos que perderam para o maior elemento, também de forma recursiva.

A Figura 2 ilustra o funcionamento do algoritmo para uma entrada de 8 elementos. A Figura 2 *a*) apresenta o funcionamento do primeiro laço (while). Todos os elementos do vetor A são comparados aos pares. Os perdedores são eliminados até que o vetor contenha apenas um elemento. Este elemento é o maior do conjunto. No caso do exemplo da Figura 2, este número é o 12. É importante ressaltar que este algoritmo precisa armazenar todos os números que perderam para o vencedor (maior número).

A Figura 2 *b*) ilustra o funcionamento do segundo laço (while). O objetivo deste laço é encontrar o segundo maior número do conjunto. É sabido que o segundo maior elemento necessariamente perdeu uma comparação para o maior elemento do conjunto. Portanto deve-se procurar o maior elemento dentro do conjunto dos números que perderam para o vencedor. No exemplo da Figura 2 o segundo maior número é o 10. Observe o algoritmo apresentado trata também o caso do vetor de entrada ter um tamanho ímpar. Neste caso o último número só é comparado em uma iteração onde se tenha outro número ímpar de elementos chegando assim a um número par.

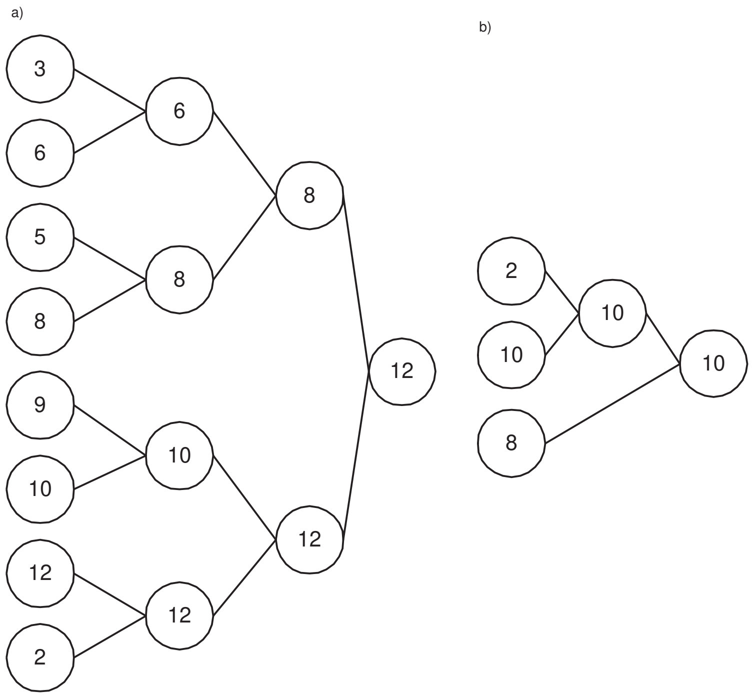

Figura 2: Funcionamento do algoritmo para uma entrada 8 elementos

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

   Para a análise de complexidade do algoritmo apresentado na seção anterior considera-se como operação relevante as comparações para determinar qual o maior elemento de cada dupla.

   O primeiro laço (while) pode ser representado pela seguinte equação de recorrência:

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

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

$$\vdots$$

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

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

Através de [1, 10]:

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

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

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

Através da condição de parada $T(1) = 0$, tem-se:

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

Substituindo:

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

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

Portanto, são necessárias $n - 1$ comparações para determinar o maior número no primeiro laço (while).

A análise de complexidade do segundo laço (while) é idêntica à análise do primeiro. O segundo laço (while) é responsável por achar o segundo maior elemento do vetor dentro do conjunto $m = log(n)$ números que foram derrotados pelo maior número. Pode-se concluir então que a complexidade do segundo laço (while) é de $log(n) - 1$.

Somando-se as complexidades dos dois laços (while), tem-se:

$$n - 1 + log(n) - 1 = n + log(n) - 2$$

É importante ressaltar que esta entrada independe da ordem dos elementos no vetor.

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

   Teorema. Nenhum algoritmo que encontre simultaneamente o maior e o segundo maior elemento de um conjunto pode garantir um desempenho, no pior caso, de c comparações para $c < n + \lceil log(n) \rceil - 2$.

   Prova[3]. Assume-se que A seja o maior elemento, que B seja o segundo maior e os outros $(n-2)$, os menores elementos. Qualquer algoritmo deve definir A como maior que B pois do contrário, pode-se inverter os valores de A e B e o algoritmo iria determinar erroneamente o segundo maior elemento. O algoritmo deve também identificar o grupo de $(n-2)$ elementos menores que B. Para verificar quais elementos são menores que B, cada elemento deste conjunto deve ter perdido uma comparação para B ou para outro elemento deste grupo. Isto implica que para qualquer entrada, um algoritmo deve fazer pelo menos $(n-2)$ comparações que não envolvam o maior elemento. Será mostrada agora uma forma de forçar o algoritmo a realizar pelo menos $\lceil log(n) \rceil$ comparações que envolvam o maior elemento e este algoritmo realizará pelo menos $n + \lceil log(n) \rceil - 2$ comparações no total.

   Para tanto, considere o seguinte adversário. Dado um conjunto de respostas para comparações pode-se dizer que uma resposta para novas comparações são conhecidas se a resposta é coerente com as comparações anteriores. Nosso adversário irá manter para cada elemento x, o conjunto de $L(x)$ de elementos conhecidos como menores ou iguais a x baseado em comparações anteriores. Inicialmente, $L(x) = x$ para todos elementos, como se a única coisa de que se pode ter certeza é que x é menor ou igual a ele mesmo. Agora, se resposta para pergunta $A \leq B$? for conhecida, o adversário dará a resposta correta. Do contrário, o adversário dirá que A ganhará a comparação exatamente quando $|L(a) \geq |L(b)||$. Respondendo as perguntas desta maneira, pode-se ter certeza que haverá sempre pelo menos uma coerência com todas as respostas do adversário.

   Através deste adversário, pode-se afirmar que se um elemento particular x esteve envolvido em exatamente k comparações, então $|L(x)| \leq 2^k$. Esta afirmação pode ser provada por indução. Como um caso base, foi visto que $k = 0, |L(x)| = 1 = 2^0$. Para o passo indutivo, assuma que $|L(x) \leq 2^{k-1}|$ após $k-1$ comparações envolvendo x, pode ser considerado a k-ésima comparação comparação entre x e y. No caso de x perder a comparação ou se a resposta fosse conhecida como x sendo pelo menos do mesmo tamanho de y, então nenhum elemento é adicionado a $L(x)$ e $L(x) \leq 2^{k-1} \leq 2^k$. Se a resposta não é conhecida, e o adversário responde que x ganha a comparação, então tem-se o caso em que $|L(y)| \leq |L(x)|$. Se $L'(x)$ é igual ao conjunto resultante após k-ésimas comparações então $L'(x) = L(x) \bigcup L(y)$. Mas estes são os únicos elementos que podem ser adicionados a $L(x)$. Então, $|L'(x)| \leq |L(x)| + |L(y)|$

(Alguns elementos de $L(y)$ podem já estar em $L(x)$. Apesar disso, por indução, sabe-se que $|L(x) \leq 2^{k-1}|$ e pelas regras do adversário, sabe-se que $|L(y)| \leq |L(x) \leq 2^{k-1}|$ e $|L'(x)| \leq 2^{k-1} + 2^{k-1} = 2^k$.

Finalmente pode-se mostrar que qualquer algoritmo teria feito $\lceil log(n) \rceil$ comparações que envolvam o maior elemento. Observe que desde que qualquer algoritmo saiba qual é o maior elemento A, $L(A) = n$. Se A esteve envolvido em k comparações, sabe que $n = L(A) \leq 2^k$, e reorganizando, pode-se ver que $k \geq log(n)$. Já que k deve ser um inteiro, $k \geq \lceil log(n) \rceil$ e consequentemente o maior elemento esteve envolvido em pelo menos $\lceil log(n) \rceil$ comparações.

4. O algoritmo apresentado no item (a)é ótimo? $f(n)$ é um limite assintótico firme? Justifique sua resposta.

   Sim, conforme demonstrado no item anterior o algoritmo apresentado é ótimo pois realiza $n + log(n) - 2$ comparações para determinar o maior e o segundo maior elemento do vetor.

   Conforme [1, 10], ser assintoticamente firme significa a existência de constantes c_1, c_2 e m tais que, para todo $n \geq m$, o valor de $g(n)$ está sobre ou acima de $c_1 f(n)$ e sobre ou abaixo de $c_2 f(n)$. Portanto:

$$c_1 n \leq n + log(n) - 2 \leq c_2 n$$

$$c_1 \leq 1 + \frac{log(n)}{n} - \frac{2}{n} \leq c_2$$

   Pode-se observar que a partir de $m = 4$ a relação anterior é válida para $c_1 = 1$ e $c_2 = 2$. Portanto pode-se afirmar que o algoritmo apresentado é assintoticamente firme.

4 Shellsort

O objetivo deste trabalho é realizar um estudo do algoritmo Shellsort de ordenação interna [10].

1. 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 [10]. Repita cada experimento algumas vezes e obtenha a média para cada medida de complexidade.

   O shellsort foi originalmente proposto por [9]. Trata-se de uma extensão do algoritmo de ordenação por inserção. O shellsort permite trocas de registros que estão distantes um do outro. Os itens que estão h posições são rearranjados de tal forma que todo h-ésimo item leva uma seqüência ordenada. Tal seqüência é dita estar h-ordenada. Uma das seqüências de h mais usadas, foi proposta por [7] e consiste em:

Tabela 1: Resultados empíricos do número de comparações e movimentos de registro

Tamanho do vetor	Comparações	Movimentos de registros
50	288	306
100	737	778
200	1.810	1.895
400	4.391	4.552
600	7.396	7.658
800	10.554	10.908
1.000	13.853	14.274
2.000	32.619	33.511
4.000	77.439	79.046
6.000	126.115	128.827
8.000	180.191	183.618
10.000	234.795	238.984
20.000	546.342	555.585
40.000	1.281.704	1.297.790
60.000	2.067.545	2.095.288
80.000	2.952.975	2.986.995
100.000	3.886.287	3.926.764
200.000	9.047.208	9.135.055
400.000	21.173.745	21.341.360
600.000	34.656.373	34.919.880
800.000	49.078.264	49.415.546
1.000.000	64.389.939	64.791.380

$$\begin{cases} h(s) = 3h(s-1) + 1 & , s > 1 \\ h(s) = 1 & , s = 1 \end{cases}$$

A Tabela 1 apresenta os resultados obtidos para número esperado de comparações e movimento de registros. A implementação do algoritmo shellsort foi baseada em [10] e utiliza a seqüência de saltos de knuth [7]. Acrescentou-se alguns contadores de comparacões/movimento de registros e foi alterada uma parte do permut.c pois este não estava permutando a última posição do vetor. Foram realizados 50 experimentos para cada tamanho de vetor a ser ordenado. Os valores apresentados na Tabela 1 são a média destes 50 valores arredondados para cima, pois não há muito sentido em se falar em comparação fracionária.

Pode-se observar que o número de movimentos de registros está sempre ligeiramente acima do número de comparações. Portanto pode-se afirmar que o número de comparações e movimentos de registros varia aproximadamente proporcionalmente. A listagem do código usado para gerar os resultados empíricos pode ser encontrada no Apêndice A.

2. Dê a sua interpretação dos dados obtidos, comparando-os com as conjecturas relatadas em [10].

   Em [10], podem-se observar duas conjecturas:

$$C(n) = O(n^{1,25})$$
$$C(n) = O(n(lnn)^2)$$

   Para comparar estas conjecturas com os dados empíricos de número de comparações obtidos, foram gerados dois gráficos. A Figura 3 apresenta o gráfico dos resultados obtidos com a conjectura 1. Foi usada uma constante c1 de multiplicação igual a 2. Os resultados foram bastante próximos para a maioria dos pontos do gráfico. Para a conjectura 2, não foi possível encontrar uma constante de multiplicação c1 que apresentasse bons resultados para todos os pontos do gráfico. Na Figura 4 são comparados os resultados empíricos obtidos com 2 curvas para a conjectura 2, a primeira com c1 igual a 0,3 e a segunda com c1 igual a 0,4.

   Portanto, foi observado que para a seqüência de saltos utilizada nesta implementação a conjectura 1 apresentou resultados mais próximos do que a conjectura 2.

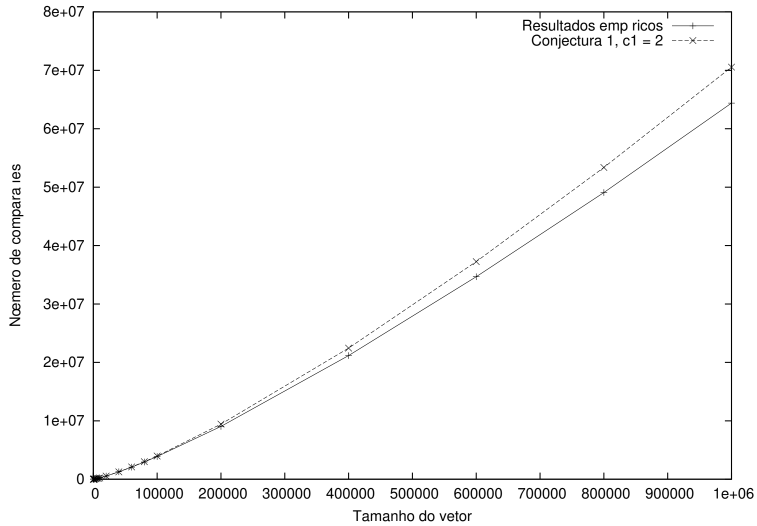

Figura 3: Comparação dos dados obtidos com a Conjectura 1

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

   A maioria dos resultados analíticos para o shellsort que são dependentes de seqüências específicas e confinados ao pior caso [8]. Em [5] é apresentado o primeiro limite inferior independente de seqüência para o caso médio do shellsort:

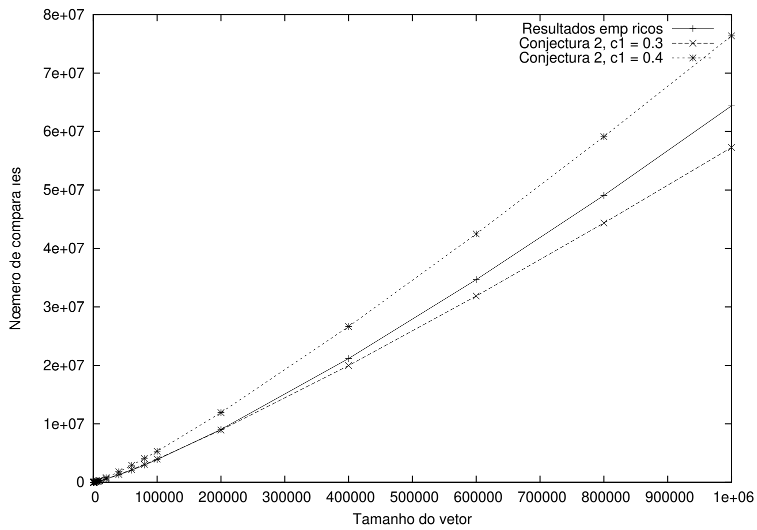

Figura 4: Comparação dos dados obtidos com a Conjectura 1

$$\Omega(pn^{1+\frac{1}{p}})$$

onde p é o número de passos necessários a finalização da ordenação. Este limite inferior é em relação a movimentos de registros. A Figura 5 apresentada uma comparação dos resultados empíricos de movimentos de registros com os resultados analíticos de [5]. Pode-se observar que os resultados empíricos diferem bastante dos resultados analíticos. Possivelmente isto acontece devido a generalidade deste limite inferior, pois o mesmo vale para qualquer seqüência de saltos e refere-se ao caso médio.

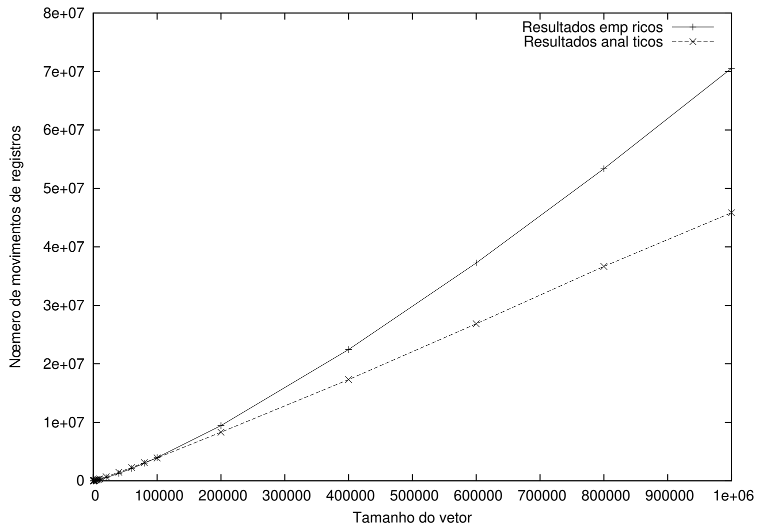

Figura 5: Comparação dos resultados empíricos com os analíticos [5]

Referências

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

[2] Rev. Charles Lutwidge Dodgson. Lawn tennis tournaments, 1883. Reprinted in The Works of Lewis Carroll, ed. Roger Lancelyn Green, Hamlyn Publishing Group Ltd., Middlesex, 1968.

[3] Michael Goldwasser. Sample proofs involving selection problems. Design and Analysis of Computer Algorithms, Handout,#5, Loyola University Chicago, February 2003.

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

[5] Tao Jiang, Ming Li, and Paul Vitányi. A lower bound on the average-case complexity of shellsort. *Journal of the ACM*, 47(5):905–911, September 2000.

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

[7] D.E. Knuth. *The Art of Computer Programming; Vol 3: Sorting and Searching*. Addison-Wesley, 1973.

[8] Robert Sedgewick. Analysis of shellsort and related algorithms. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, 1996.

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

[10] Nivio Ziviani. *Projeto de Algoritmos com Implementações PASCAL e C*. Pioneira Thomson Learning, 2004.

Apêndice A - Listagem do código usado para gerar os resultados do item 4

```
#include   <limits.h>
#include   <sys/time.h>
#include   <stdio.h>

#define Tamanho 1000000

typedef long ChaveTipo; typedef struct Item {
  ChaveTipo Chave;
  /* outros componentes */
} Item;

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

void Shellsort(Item *A, Indice *n) { int i, j;
  int h = 1;
  Item x;
  long unsigned int mov_registro = 0;
  long unsigned int comparacoes = 0;

  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) // Incrementa comparaçõe
      {
          A[j] = A[j - h];
          mov_registro++; // Incrementa movimentos de registro
          j -= h;
          if (j <= h) goto L999;
        }
          L999: A[j] = x;
      mov_registro++; // Incrementa movimentos de registro
    }
    } while (h != 1);

   printf("\nNumero de movimentos no registro: %d\n", mov_registro);
   printf("\nNumero de comparações: %d\n", comparacoes);
}
```

```
void Imprime(Item *V, Indice *n) { for (i = 1; i <= *n; i++)
    printf("%d ", V[i].Chave);  printf("\n");
} 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;
    }
  }
  printf("OK: ");
  Imprime(V, n);
} 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; i>=1; i --) {
   j = (int)(rand0a1() * (n-1) +2) ;
   b = A[i];
   A[i] = A[j];
   A[j] = b;
 }
} int main(int argc, char *argv[])

{ Vetor B;
  n = Tamanho;   /*Tamanho do arranjo a ser ordenado*/

  struct timeval seed;
  gettimeofday (&seed,NULL);
  srand((int)(seed.tv_sec + 1000*seed.tv_usec));

  for (i = 1; i <= n; i++)
    A[i].Chave = i;

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

  Shellsort(B, &n);

  return 0;
}
```