UNIVERSIDADE FEDERAL DE SÃO PAULO
CAMPUS SÃO JOSÉ DOS CAMPOS
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA (DCT)

Exercícios de Projeto e Análise de Algoritmos
Lista 2

UC: **Projeto e Análise de Algoritmos**

Aluno: **Thauany Moedano**

RA: **92486**

Professor: **Dr. Reginaldo Massanobu Kuroshu.**

Entrega: **12/10/2015**

Resumo

Resolução da lista 2 de exercícios da aula de Projeto e Análise de Algoritmos no 2º semestre de 2015.

1. Mostre pelo método de substituição que

a) $T(n) = 2T(n/2) + 1 = O(n)$

Resposta: Queremos provar que $T(n) \leq cn$ para uma escolha apropriada de uma constante $c \geq 0$

Assumindo que o limite vale para todo positivo m $\leq$ n

m $= \frac{n}{2}$

$T(\frac{n}{2}) \leq c.\frac{n}{2}$

Substituinto $T(\frac{n}{2}$ na relação de recorrência:

$T(n) \leq 2.c.\frac{n}{2} + 1$

$T(n) \leq cn + 1$

Portanto, escolhendo um $c \geq 1$ temos que T(n) = O(n)

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

Resposta: Queremos provar que $T(n) \leq cn$ para uma escolha apropriada de uma constante $c \geq 0$

Assumindo que o limite vale para todo $m \leq n$

m = n-1

$T(n-1) \leq c.(n-1)$

Substituindo T(n-1) na relação de recorrência:

$T(n) \leq c(n-1)+1$

$T(n) \leq cn - c + 1$

Portanto, escolhendo um $c \geq 1$ temos que T(n) = O(n)

c) $T(n) = T(n/2) + 1 = O(\log n)$

Resposta: Queremos provar que $T(n) \leq c \log n$ para uma escolha apropriada de uma constante $c \geq 0$

Assumindo que o limite vale para todo $m \leq n$

m = $\frac{n}{2}$

$T(\frac{n}{2}) \leq c.\log \frac{n}{2}$

Substituindo $T(\frac{n}{2})$ na relação de recorrência:

$T(n) \leq c \log \frac{n}{2} + 1$

$T(n) \leq c \log 2 \log n + 1$

Considerando a base 2 para facilitar os cálculos:

$T(n) \leq c \log n + 1$

Portanto, escolhendo um $c \geq 1$ temos que T(n) = O($\log n$)

2. Encontre um bom limite superior assintótico para a recorrência utilizando o método de árvore de recursão: $T(n) = T(n-1) + T(n-2) + \Theta(1)$

Esta recorrência refere-se justamente a resolução da série de Fibonnacci que é dada por uma árvore de recursão semelhante a esta abaixo: (PS: não fui eu que desenhei)

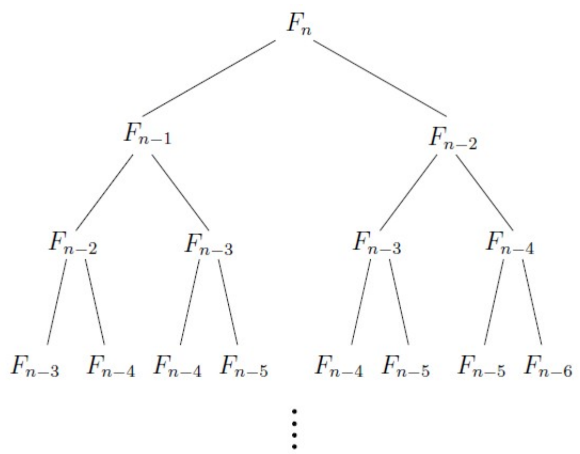

Figura 1: Árvore de Recursão da Série de Fibonnaci

A cada nível de recursão, o algoritmo executa 2^i. Portanto, o número total de execuções desse algoritmo é um somatório de 2^i com i variando de 1 a n. Portanto, podemos estabelecer que o limite superior para esta recorrência é de $O(2^n)$

3. Resolva as recorrências aplicando o Teorema Mestre.

a) $T(n) = 2T(n/2) + n$
$n^{\log 2} = n^1 = f(n)$
$T(n) = \Theta(n \log n)$

b) $T(n) = 4T(n/2) + n$
$n^{\log 4} = n^2 \geq f(n)$
$T(n) = \Theta \log(n^2)$

c) $T(n) = 4T(n/2) + n^2$
Similar ao exercicio anterior:
$n^2 = f(n) = n^2$
$T(n) = \Theta(n^2 \log n)$

d) $T(n) = 4T(n/2) + n^3$
Similar ao exercício anterior:
$n^2 \leq f(n) = n^3$
Também temos que mostrar que:
$af(\frac{n}{b}) \leq cf(n)$
$4.f(\frac{n}{2}) \leq cf(n)$
$4.((\frac{n}{2}))^3 \leq c.n^3$
Escolhendo $c \geq 4$ temos que a desigualdade é verdadeira e portanto:
$T(n) = \Theta(n^3)$

4. Sejam A e B dois vetores de números inteiros tais que o número total de inteiros nos dois vetores é n, e x um número inteiro. Projete um algoritmo de complexidade $O(n \log n)$ para o problema de determinar se existem índices i e j tais que $A[i] + B[j] = x$.

A ideia principal deste algoritmo é procurar complementares entre os dois vetores A e B de tal forma que sua soma seja o X. Ou seja, existe um único elemento que somado a cada elemento de A[i] que pode resultar em X. Queremos saber se este elemento está em B. Portanto devemos olhar para os menores elementos de A e ir somando com os maiores elementos de B.

Como A e B são ordenados de forma inversa utilizando o HeapSort, temos um gasto de $n \log n$. O while faz cerca de $O(n)$ comparações, portanto o algoritmo é de $O(n \log n)$

```
int Algoritmo(int A[],int B[], int x) {
  int i = 0,j = 0;

  HeapSort(A);
  DHeapSort(B); //A eh ordenado em ordem crescente e B eh ordenado em
     ordem descrecente utilizando o HeapSort

  while(i < n || j < n) {
    if(A[i] + B[j] > x)
      j++;
    else if (A[i] + B[j] < x)
      i++;
    else
      return(1);
  }
    return(0);
}
```

5. Seja $A[1 \dots n]$ um vetor com n números distintos. Se $i < j$ e $A[i] > A[j]$, então o par (i, j) é chamado de uma inversão de A. Projete um algoritmo por divisão e conquista que determina o número de inversões em qualquer permutação de n elementos em tempo $\Theta(n \log n)$ no pior caso.

A ideia principal neste algoritmo foi fazer pequenas modificações na função MergeSort. A parte de intercalação nada mais do que verifica inversões, portanto basta acrescentar um contador na função intercala de maneira que seja possivel contar inversões a cada chamada recursiva da função.

```
int Intercala (int vet[], int inicio, int meio, int fim) {

      int aux[fim-inicio];
      int i, j, k;
      i = 0;
      int c;

      for (k = inicio; k <= meio; k++) {
           aux[i] = vet[k];
           i++;
      }

      j = ((fim-inicio)/2)+1;
      for (k = fim; k > meio; k--) {
           aux[j] = vet[k];
           j++;
      }

      i = 0;
      j = fim-inicio;
```

```
        c= 0;


        for (k = inicio; k <= fim; k++) {

            if(aux[i] <= aux[j]) {
                vet[k] = aux[i];
                i++;
            }
            else {
                vet[k] = aux[j];
                j--;
                c = c + (meio-i+1); //O pulo do gato eh contar a cada vez que o
                    vetor eh intercalado o numero de inversoes existentes.
            }

        }
        return(c);
    }

int Mergesort (int inicio, int fim, int vet[]) {

        int meio;
        int c;
        //C eh um contador que vai sendo atualizado a cada passagem recursiva
            da funcao
        if (inicio < fim) {
            meio = (inicio+fim)/ 2;
            c =  Mergesort(inicio, meio, vet); //Como dividimos os vetores na
                metade chamamos mergesort para a primeira metade
            c = Mergesort(meio+1, fim, vet); //E depois para a segunda metade
            c = Intercala(vet, inicio, meio, fim);
            return(c);

        }
```

6. Um vetor $A[1 \ldots n]$ é unimodal se consiste de uma sequência crescente seguida de uma sequência decrescente, ou seja, se $A[1 \ldots n]$ é unimodal, então existe um índice $m \in \{1, 2, \ldots, n\}$ tal que

- $A[i] < A[i+1]$ para todo $1 \leq i < m$; e
- $A[i] > A[i+1]$ para todo $m \leq i < n$,

onde $A[m]$ é o maior elemento do vetor.

a) Projete um algoritmo de divisão e conquista e forneça um pseudo-código do algoritmo que determina o maior elemento de um vetor unimodal em tempo $O(\log n)$.

Devemos ter em mente que o maior elemento do vetor sera aquele contido em A[m] pois divide os vetores crescente e descrecente. O elemento em A[m] é caracterizado quando o elemento á esquerda é menor que A[m] e o elemento a direita também é menor que A[m].

```
int compara(int A[], int p, int r, int meio) {

   if(A[meio-1] < A[meio] && A[meio+1] < A[meio])
      return(1);
   else
      return(0);
}

int unimodal(int A[], int p, int r) {
   if(p < r+1) {
      q = (p+r)/2;
      unimodal(A,p,q-1);
      unimodal(A, q+1, r);
      if(compara(A, p,r, meio))
         return(1);
   }
   else
      return(0);
}
```

b) Mostre qual é o limite superior assintótico para o tempo de execução do algoritmo.

A função unimodal faz chamadas recursivas dividindo o problema em dois subproblemas menores. Desta forma, até se chegar no caso base, o tempo de execução requer cerca de $\log n$ operações para se chegar ao caso base.

A função compara faz exatamente uma comparação independente do tamanho do subproblema e portanto tem um tempo constante de execução $\Theta(1)$ para executar. Portanto o tempo de execução total deste algoritmo é de $O(\log n)$