PROVA 2008/2

Questão 1

$2^{n+1} = \Theta(2^n)$

Vou provar que $1 \cdot 2^n \leq 2^{n+1} \leq 4 \cdot 2^n$ para $n \geq 1$

Primeiro vou provar que $2^{n+1} \leq 4 \cdot 2^n$

$2^{n+1} = 2^1 \cdot 2^n$

$\leq 2^2 \cdot 2^n$

$= 4 \cdot 2^n$ //

Agora vou provar que $\frac{1}{2} \cdot 2^n \leq 2^{n+1}$ para $n \geq 1$

$\frac{1}{2} \cdot 2^n \leq 2 \cdot 2^n$

$= 2^{n+1}$ //

Assim $2^{n+1} = \Theta(2^n)$ é verdade.

→ Não é verdade que $3^n = \Theta(2^n)$

Vou provar por absurdo que $c_1 \cdot 2^n \leq 3^n \leq c_2 2^n$ para $n \geq n_0$ não é verdade.

$3^n \leq c_2 \cdot 2^n$

$c_2 \geq \frac{3^n}{2^n}$

$c_2 \geq \left(\frac{3}{2}\right)^n$ absurdo já que c_2 deve ser uma constante.

$c_1 \cdot 2^n \leq 3^n$

$c_1 \leq \frac{3^n}{2^n}$

$c_1 \leq \left(\frac{3}{2}\right)^n$ absurdo, já que c_1 deve ser uma constante.

$2^{2n} = \Theta(2^n)$

Vou provar que $c_1 \cdot 2^n \leq 2^{2n} \leq c_2 \cdot 2^n$ para $n \geq n_0$ não é verdade.

Lema 1: $c_1 2^n \leq 2^{2n}$ para $n \geq n_0$

Prova: $c_1 \cdot 2^n \leq 2^{2n}$

$c_1 \leq \frac{2^{2n}}{2^n}$

$c_1 \leq \left(\frac{4}{2}\right)^n$

$c_1 \leq 2^n$, absurdo pois c_1 deve ser uma constante.

Lema 2: $2^{2n} \leq c_2 \cdot 2^n$ para $n \geq n_0$

Prova: $2^{2n} \leq c_2 \cdot 2^n$

$\frac{2^{2n}}{2^n} \leq c_2$

$c_2 \geq \left(\frac{4}{2}\right)^n$

$c_2 \geq 2^n$, absurdo, pois c_2 deve ser uma constante.

Pelos Lemas 1 e 2 comprova-se que 2^{2n} não é $\Theta(2^n)$

Questão 2 errada!.

Questão 2 - Qual a diferença entre as afirmações 1 e 2?

A afirmação 1 significa que o algoritmo consome tempo $O(n^2)$ para alguma instância (pior caso) já a afirmação 2 significa que o algoritmo consome tempo $O(n^2)$ sempre.

A afirmação 2 significa que o algoritmo executa em $O(n^2)$ para uma instância específica (a de melhor caso). Já a afirmação 2 significa que o algoritmo

executa em no máximo $O(n^2)$ para uma instância não determinada

c[1,2] ← 3
c[1,3] ← 5
c[1,4] ← 10
c[2,3] ← 2
c[2,4] ← 1
c[3,4] ← 1

Questão 5-

$v[i,k]$ = custo mínimo de uma viagem de i a k.

Recorrência:

$v[i,k] \leftarrow 0$ se $i = k$

$v[i,k] \leftarrow \min_{i<j<k} \{ \min \{ v[i,j] + v[j,k] \}, c[i,k] \}$

	1	2	3	4
1	0	3	5	4
2		0	2	1
3			0	1
4				0

```
VERIFICA-PREÇO(c, n)
1  Para i ← 1 até n faça
2      v[i,i] ← 0
3  Para l ← 2 até n faça
4      Para i ← 1 até n-l+1
5          k ← i+l-1
6          v[i,k] ← c[i,k]
7          Para j ← i+1 até k-1 faça
8              se v[i,k] > v[i,j] + v[j,k]
9                  então v[i,k] ← v[i,j] + v[j,k]
10 devolva v[1,n]
```

Correção:

Propriedade da subestrutura ótima:

Teorema: Seja S_{ij} uma solução ótima para $\langle c, i, j \rangle$ então existe uma única j tal que S_{ij} e $S_{j+1,n}$ são soluções ótimas de $\langle c, i, j \rangle$ e $\langle c, j+1, n \rangle$.

Prova: Suponha que S_{ij} não seja solução ótima para $\langle c, i, j \rangle$ então existe uma outra solução S^*_{ij} melhor que S_{ij} que adicionada a solução ótima $S_{j+1,n}$ produz uma solução $S^*_{1,n}$ melhor que S_{1n} O que é uma contradição.

Agora suponha que $S_{j+1,n}$ não seja solução ótima para $\langle c, j+1, n \rangle$, então existe uma outra solução ótima $S^*_{j+1,n}$ melhor que $S_{j+1,n}$ que adicionada a solução ótima S_{ij} produz uma solução S^*_{1n} melhor que S_{1n} O que é uma contradição.

Análise de consumo de tempo:

Tamanho da instância: n

$t(n)$ = consumo de tempo no pior caso para VERIFICA-PREÇO.

Linha	
1-2	$O(n)$
3	$O(n)$
4-6	$O(n^2)$
7-9	$O(n^3)$
10	$O(1)$
$t(n)$	$= O(n^3)$

Questão 4-

a)
```
ARRANJO(S, n, i°)
1 MERGESORT(S, n)
2 Para j ← 1 até i° faça
3     B[j] ← S[j]
```

O algoritmo é ordenado pela rotina Mergesort não modificada que possui consumo $O(n \lg n)$ e em seguida são transferidos os i primeiros elementos de S para B com custo $O(i)$

Tamanho da instância: (n, i)

$t(n,i)$ = consumo no pior caso

$t(n,i) = O(n \lg n + i)$

b) HEAP-OPTION(S, n, i)

```
1  BuildHeap(S, n)
2  Para j ← 1 até i:
3      B[j] ← EXTRACTMIN(S)
```

O algoritmo HEAP-OPTION utiliza a rotina BuildHeap com modificação que consome tempo $O(n)$ para transformar S em um MinHeap e as linhas 2 e 3 extraem os i menores elementos do minHeap cada EXTRACTMIN consome $O(\lg n)$

Tamanho da instância: (n, i)

$t(n,i)$ consumo no pior caso da heapoption

Linha

1 $O(n)$

2- $O(i)$

3. $O(i \lg n)$

$t(n,i) = O(n + i \lg n)$

c) PARTIÇÃO-I(S, n, i)

```
1  q ← SELECTBFPRT(S, n, i)
2  PARTICIONA(S, 1, n, q)
3  Para j ← 1 até i
4      B[j] ← S[j]
```

O algoritmo PARTIÇÃO-I utiliza a rotina SELECTBFPRT que recebe um vetor com o tamanho e o índice do iésimo elemento a ser encontrado e devolve o índice do i-ésimo menor elemento

PARTIÇÃO-I também utiliza a rotina particiona presente tb no algoritmo QuickSort. Esta rotina recebe um vetor com o intervalo a ser particionado e um índice indicando o pivô que é uma modificação do original.

Esta rotina arranja o vetor da forma $S[1..q-1] \leq S[q] \leq S[q+1..n]$ dessa forma caso os i-menores elementos estarão nas posições $S[1..q]$, portanto as linhas 2 e 4 copiam os i-menores elementos para o vetor B.

Tamanho da instância: (n, i)

$t(n,i)$ consumo no pior caso:

Linha

1 $O(n)$

2 $O(n)$

3 $O(i)$

$t(n,i) = O(n+i)$

Questão 8 - Vou provar que o problema 1 que chamarei PARTIÇÃO é NP-Completo.

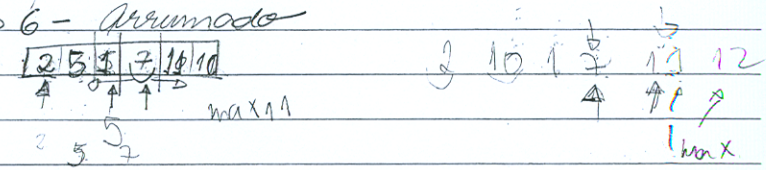
Primeiro provarei que PARTIÇÃO está em NP. Um certificado pode ser adquirido em tempo polinomial da seguinte forma: Seja (A,B) uma possível solução do problema da PARTIÇÃO. Soma-se todos os elementos de A e todos os elementos de B. Em seguida compara-se os dois valores. se forem iguais (A,B) é solução de PARTIÇÃO caso contrário não. Dessa forma está provado que partição está em NP.

Seja o problema é NP-completo e chamado de par Subset-Sum. Provarei que PARTIÇÃO é NP-Difícil. Mostrando que Subset-Sum $\leq_p$ PARTIÇÃO
Suponha I como uma instância de Subset-Sum podemos construir uma instância (I',J') de PARTIÇÃO da seguinte forma:
Seja $\Sigma_{i \in I} w_i = s$ e s metade do somatório total (instância criada em T.P.) então I' $\Sigma_{i \in I'} w_i = s = \Sigma_{j \in J'} w_j$.
Teorema: $\Sigma_{i \in I} w_i = s$ para s metade do somatório total se e somente se $\Sigma_{i \in I} w_i = \Sigma_{j \in J} w_j$
PROVA:
→ Suponha I uma instância de Subset-Sum tal que $\Sigma_{i \in I} w_i = s$, se s for metade do valor total então existe um conjunto J excludente tal que o somatório de seus elementos também tem valor s, assim existe uma partição (I,J) onde $\Sigma_{i \in I} w_i = \Sigma_{j \in J} w_j$
← Suponha $\Sigma_{i \in I} w_i = \Sigma_{j \in J} w_j$.

então $\Sigma_{i \in I} w_i - \Sigma_{j \in J} w_j = 0$ ou seja $\Sigma_{i \in I} w_i$ é igual a metade do valor total. se chamarmos s = metade do valor total então temos uma partição I tal que $\Sigma_{i \in I} w_i = s$.

Assim provo que Subset-Sum $\leq_p$ PARTIÇÃO e que PARTIÇÃO é NP-Completo.

Questão 6 - Arrumado




```
EH-ARRUMADO(A, n)
1  pivo ← 1        max ← A[1]   A[n+1] ← ∞   j ← 2
2  enquanto j ≤ n
3       se A[j] > max
4            então max ← A[j]
5       se A[j] < A[pivo]
6            enquanto max > A[j+1]
7                 j ← j+1
8            pivo ← j+1
9       j ← j+1
10 devolva pivo
```

O algoritmo EH-ARRUMADO recebe um vetor A[1..n] e devolve um inteiro i tal A[1..i-1] ≤ A[i] ≤ A[i+1..n] se for arrumado e n+1 caso não for.
tamanho da instância: n
T(n) = consumo do pior caso

O loop da linha 6 quando executado incrementa o contador do loop mais externo da linha 2 balanceando portanto a execução das linhas 2-10 em no máximo $O(n)$. Portanto trata-se de um algoritmo linear.

Corretude

invariante: No início da linha 2 pivo é um índice tal que $A[1..pivo] \leq A[pivo] \leq A[pivo+1..j-1]$

início: $j = 2$ e $pivo = 1$ então trivialmente $A[1..1] \leq A[1] \leq A[1..1]$.

manutenção: Vamos ver que a invariante do loop da linha 2 se mantém pois as linhas 3 e 4 atualizam o maior valor até o valor atual do pivô para se confirme através do segundo loop é procurado um valor maior que todos a esquerda do elemento $A[j]$ sendo incrementado na linha 7 caso não seja o maior e quando encontrado é incrementado na linha 9 restabelecendo o invariante.

término: $j = n+1$ ou seja $A[1..pivo] \leq A[pivo] \leq A[pivo..n]$

caso não se confirme que o vetor é ordenado então $A[pivo]$ retornará $n+1$

Questão 3 - $T(1) = 1$

$$T(n) = T(\lfloor \sqrt{n} \rfloor) + 1 \qquad n > 1$$

Mostre que $T(n) = \Theta(\lg \lg n)$.

Para simplificar, vamos restringir a recorrência para valores da forma 2^{2^i}.

$T(1) = 1$

$T(n) = T(\sqrt{n}) + 1 \qquad n > 2^{2^1}, 2^{2^2}, \ldots$

$= T(\sqrt{\sqrt{n}}) + 1 + 1$

$= T(\sqrt{\sqrt{\sqrt{n}}}) + 1 + 1 + 1$

$= T(n^{1/2^i}) + i$

$= T(2) + \log \log n$

$= 1 + \log \log n$

$n^{1/2^i} = 2$

$n = \sqrt[2^i]{2}$

$n = 2^{2^i}$

$\lg n = \lg 2^{2^i}$

$\lg n = 2^i$

$\lg \lg n = \lg 2^i$

$\lg \lg n = i$

Vou provar que a fórmula fechada está correta.

Prova por indução em n:

base: $n = 2$

$T(2) = 1 = 1 + \lg \lg 2 = 1 + \lg 1 = 1$ ok!

H.I. $= T(\sqrt{n}) = 1 + \lg \lg \sqrt{n}$

passo: $n > 2^{2^1}$

$T(n) = T(\sqrt{n}) + 1$

$\overset{H.I.}{=} 1 + \lg \lg \sqrt{n} + 1$

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

$= 1 + \lg \frac{1}{2} + \lg \lg n + 1$

$\leq 1 - 1 + \lg \lg n + 1$

$= \lg \lg n + 1$ //

Vou provar que $\frac{1}{2}\lg\lg n \leq \lg\lg n + 1 \leq 2\lg\lg n$ para $n \geq 2$

$\frac{1}{2}\lg\lg n \leq \lg\lg n$

$\leq \lg\lg n + 1$ //

e $\lg\lg n + 1 \leq \lg\lg n + \lg\lg n$

$= 2\lg\lg n$ //

Assim $A(n) = \Theta(\lg\lg n)$

FORA DO PRAZO!

Questão 4 - X - ~~a~~ x ~~a~~ b c ~~a~~ a e z - x b c a e

Subseq (z, X, n, m)
  i ← 1 j ← 1
  enquanto i ≤ m e j ≤ n
    se z[i] = X[j]
      então j ← j + 1
            i ← i + 1
      senão j ← j + 1
  se i > m
    então devolva Sim
    senão devolva Não

Corretude.

Invariante: A[1..i-1] é subsequencia de X[1..j-1].

Início: i = 1 e j = 1 trivialmente z vazio é subsequencia de uma sequencia vazia.

Manutenção: O invariante é atualizado caso z[i] = X[j] onde os contadores são incrementados e caso contrário, apenas o contador de X é incrementado a fim de buscar um novo elemento. Mantendo A[1..i-1] subsequência de X[1..j-1]

término: Se i = m+1 então z[1..m] é subsequencia de X[1..j-1] que pode ser X[1..n]

e se i < m e j = n+1 então significa que z[1..m] não é subseq de X[1..n]

Radix Sort e Counting Sort

1- Suponha que todos os elementos do vetor A[1..n] pertencem ao conjunto $\{1, b, \ldots n^3\}$. Escreva um algoritmo que arranje A[1..n] o vetor em ordem crescente em tempo $O(n)$.

Para esta solução vamos considerar a representação de cada elemento em A[1..n] na base n, que utiliza algarismos entre $0, 1, \ldots, n$. Desta forma para $i = 1, \ldots n$ temos:

$$A[i] = a_2 * n^2 + a_1 n + a_0$$

sendo que a_2, a_1 e a_0 são algarismos em $\{0, 1, \ldots, n\}$ a_0 é o algarismo menos significativo e a_2 é o mais significativo.

O algoritmo Ordene é uma modificação do algoritmo Radix-Sort

Ordene (A, n)
1- ordene A[1..n] usando como chave a_0
2- ordene A[1..n] usando como chave a_1
3- Ordene A[1..n] usando como chave a_2