UFF - Análise e Projeto de Algoritmos - Prova 1 - 1/2015

1. Falso ou verdadeiro? (justifique cuidadosamente)

   (a) (0.5) Seja L um vetor com n elementos, onde cada elemento de L vale a, b ou c. Então, é possível ordenar L com complexidade de pior caso $O(n)$.

   Verdadeiro. Podemos percorrer os vetor e separando elementos em três outros vetores de acordo com o seu valor. Depois é só juntar esses 3 vetores.

   (b) (0.5) Seja L um vetor com n elementos, onde cada elemento de L vale $v_1, \ldots, v_{n-1}$ ou v_n. Então, não possível ordenar L com complexidade de pior caso $O(n)$.

   Falso. Pode ser feito o mesmo que na questão anterior.

   (c) (0.5) Se as complexidades de melhor caso de um algoritmo é $\Theta(n^i)$ e o pior caso é de $\Theta(n^{i+2})$, então a complexidade de caso médio é $\Theta(n^{i+1})$.

   Falso, depende do algoritmo. Ex: Insertion-sor pior caso = $\Theta(n^2)$; melhor caso = $\Theta(n)$; caso médio $\Theta(n^2)$.

   (d) (0.5) Se a complexidade de melhor caso de um algoritmo A que resolve um problema P é $T(n) = \Omega(n \log n)$, então o limite superior assintótico $\ell(n)$ de P satisfaz $\ell(n) = O(n \log n)$.

   Falso. T(n) é um limite inferior para P. O limite superior não foi informado, mas sabemos que o limite superior é limitado inferiormente por $n \log n$.

   (e) (0.5) O algoritmo de ordenação Quicksort tem complexidade de pior caso $T(n) = \Theta(n^2)$ quando todos os n elementos do vetor de entrada são iguais.

   Verdade. O pior caso do Quicksort é quando o vetor já está ordenado. Como todos os elementos são iguais ele já está ordenado.

2. Considere o seguinte algoritmo:
   Algoritmo(L, i)
   Entrada: vetor $L[1 \ldots n]$ contendo n elementos e i valor inteiro (entre 1 e n)
   se $|L| \leq 1$ então retornar L
   senão
   x = primeiro elemento de L
   L_1 = vetor formado pelos elementos de L menores do que x
   L_2 = vetor formado pelos elementos de L iguais a x
   L_3 = vetor formado pelos elementos de L maiores do que x
   se ($|L_1| < i$ e $|L_1| + |L_2| \geq i$) então retornar x
   senão
   se $|L_1| \geq i$ então retornar Algoritmo(L_1, i)
   senão retornar Algoritmo(L_3, $i - |L_1| - |L_2|$)

   (a) (0.5) O que faz este algoritmo?

   Algoritmo para achar o menor i-essimo elemento do vetor.

   (b) (0.5) Mostre a árvore de recursão deste algoritmo para a entrada $L = [5, 2, 1, 7, 9, 3, 1, 6]$ e $i = 4$. Cada nó desta árvore é uma chamada recursiva. Ao definir os vetores L_1, L_2, L_3, respeitar a mesma ordem relativa que os elementos têm em L.

   Chamada - L_1 L_2 L_3
   (L, 4) - [2, 1, 3, 1] [5] [7, 9, 6]
   (L_1, 4) - [1, 1] [2] [3]
   (L_3, 1) - [] [3] []
   [3]

(c) (0.5) Escreva equações de recorrência do algoritmo acima.

$T(n) = 1$, se n= 1
$T(n) = T(p) + \Theta(n)$, se $n > 1$ e $p > i$
$T(n) = T(n - p) + \Theta(n)$, se $n > 1$ e $p < i$.

(d) (0.5) Analise a complexidade de pior caso deste algoritmo para um vetor L com n elementos.

No pior caso p=1 (ou p = n-1) em todas as iterações e a cada nível da recorrência temos $\Theta(n)$. Então ficamos com $\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$.

(e) (0.5) Mostre uma entrada com n = 8 e i = 6 que leva o algoritmo ao pior caso.

L = [1, 2, 3, 4, 5, 8, 7, 6]

3. (3.0) Sejam $[l_i, r_i]$, $1 \leq i \leq n$, intervalos fechados sobre a reta real. Desenvolva um algoritmo guloso que determine a menor quantidade de intervalos para cobrir completamente o intervalo $[0, M]$, para um dado $M \geq 0$, ou conclua que não é possível cobri-lo. Depois use as notações O, o, Ω e ω para a complexidade do algoritmo proposto.

Ordene os intervalos de forma não-decrescente em relação a l_i.
Para $i = 1 \ldots n - 1$
    $j = i + 1$
    enquanto $l_i = l_j$
        se $r_i < r_j$ então troca r_i com r_j
        se $j = n$ então break
        $j = j + 1$
    se $j > i + 1$ então $i = j$
fim = 0
j = 0
Para $i = 1 \ldots n$ (na ordem acima)
    se $l_i > fim$ então retorne "não é possível cobrir"
    se $r_i > fim$ então $j = j + 1$ e $fim = r_i$
returne j

O algoritmo é guloso, pois ele sempre escolhe o maior intervalo $[l_i, r_i]$ seguindo a ordem de r_i. Não leva sempre a melhor solução, mas garante que sempre encontra uma solução se ela existir. Poderíamos garantir a otimalidade mudando o ultimo Para para:
Para $i = 1 \ldots n$ (na ordem acima)
    se $l_i > fim$ então retorne "não é possível cobrir"
    se $r_i > fim$ então
        $max = i$
        Para $k = i + 1 \ldots n$
            se $l_k \leq fim$ então $i = i + 1$
            senão break
            se $r_k > r_{max}$ então $max = k$
        $j = j + 1$ e $fim = r_{\max}$

Ordenar tem melhor caso $\Theta(n)$ e pior caso $\Theta(n \log n)$. A segunda parte (Para e enquanto) tem complexidade $\Theta(n)$. A terceira parte (Para) tem melhor caso $\Theta(1)$ e pior caso $\Theta(n)$ (para as duas versões).

Com isso podemos dizer que $T(n) = O(n \log n)$, $T(n) = o(n^2)$, $T(n) = \Omega(n)$ e $T(n) = \omega(1)$.

4. (3.0) Seja $J = \{J_1, \ldots, J_n\}$ um conjunto de tarefas que devem ser executadas sequencialmente em um mesmo processador. Cada tarefa J_i consome uma unidade de tempo do processador e produz

um lucro $r_i > 0$ caso seja concluída até o tempo limite t_i, $i = 1, \ldots, n$. Se J_i for concluída após t_i , nenhum lucro é obtido desta tarefa. O lucro total de uma sequência de execução das tarefas é a soma dos lucros obtidos pelas tarefas que foram concluídas até o tempo limite. Descreva um algoritmo que determina uma sequência de execução das tarefas em que o lucro total é maximizado. Prove que o algoritmo está correto. Qual a sua complexidade?

Ordene as tarefas for t_i na ordem não-crescente.
$t = t_1$
Para $i = 1 \ldots n$
    $t = \min(t, t_i)$
    $max = i$
    $j = i + 1$
    enquanto $j \leq n$ e $t_j \geq t$
        se $r_j > r_{max}$ então $max = j$
        $j = j + 1$
    $A[n - i + 1] = max$
    $t = t - 1$
    se $t \leq 0$ então retorne $A[i \ldots n]$
    $j = max$
    enquanto $j \geq i + 1$
        $J_j = J_{j-1}$
        $j = j - 1$
retorne A

Antes de provar temos que notar que o algoritmo faz o maior número de tarefas possível. Isso não é difícil de verificar já que só tem o tempo t só fica sem tarefa se todas as tarefas com $t_i \geq t$ já foram feitas.

Prova por absurdo. Seja T uma solução ótima e A a solução do algoritmo. Para simplificar vamos supor que em A e em T não tem tempo vago. Isso não tem problema já que depois do intervalo todas as tarefas foram escolhidas e tem que estar na solução ótima ($r_i > 0, \forall i \in \{1 \ldots n\}$). Vamos assumir que $T \neq A$, se não A é ótima. Como o algoritmo é guloso, sabemos que as tarefas que não estão em A tem lucro menor que a feita no tempo 1. Como tem que ter pelo menso duas tarefas i e j tal que $J_i \in T$, $J_i \notin A$, $J_j \in A$ e $J_j \notin T$. Como dito anteriormente $r_j \geq r_i$, se trocarmos J_i por J_j levaria a uma solução com custo menor que a ótima (contradição) ou uma solução com mesmo custo que a ótima.

A complexidade do algoritmo é $O(n^2)$.

Versão mais simples:

$A = \emptyset$
Ordene as tarefas for r_i na ordem não-crescente.
Percorra as tarefas nessa ordem:
seja t' o tempo livre o mais perto de t_i
se $t' \geq 0$ adiciona a tarefa no tempo t' a A
retorne A

A complexidade do algoritmo é $O(n \log n + nt_{\max})$. Podemos vincular $t_{\max}$ com n já que mesmo $t_{\max}$ não tendo relação direta com n o número máximo de posições que temos que percorrer a partir de um tempo t_i é n. Sendo assim chegamos a $O(n \log n + n^2) = O(n^2)$.

A prova é simples, basta supor que uma tarefa i não foi escolhida pelo algoritmo e está na solução ótima. Como essa tarefa não foi escolhida foi porque todas as tarefas escolhidas no intervalo $[0, t_i]$ tem custo $r_j > r_i$, então trocar uma dessas tarefas por a i melhora a solução ótima (contradição).