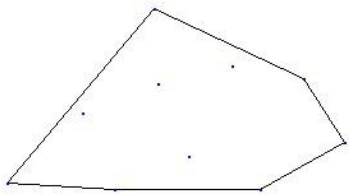
2ª Avaliação de Introdução à Análise de Algoritmos
Prof. Glauber Cintra

1) **(2 pontos)** O **fecho convexo** de um conjunto de pontos em duas dimensões é definido como o menor polígono convexo que contém todos os pontos. Cada ponto do conjunto ou é um vértice desse polígono (o fecho) ou está no seu interior. A figura apresenta o fecho convexo de um conjunto de 10 pontos no plano.



Escreva um algoritmo baseado em **divisão e conquista** que resolva o problema do fecho convexo. Determine a complexidade temporal e espacial do seu algoritmo.

```
Algoritmo Merge_Hull
    Entrada: S, um conjunto de pares ordenados (x,y), de tamanho n.
    Saída: C, fecho convexo de S.
    Se n <= 5
        Fecho_Bruto(S)
    Divida S em 2 partes aproximadamente iguais
    A = metade direita de S
    B = metade esquerda de S
    C = Mesclar_Fechos( Merge_Hull(A) e Merge_Hull(B))
    Devolva C

Procedimento Fecho_Bruto
    Entrada: S, um conjunto de pares ordenados (x,y), de tamanho n.
    Saída: C, fecho convexo de S
    Para i=0 até n
        Para j=i+1 até n
            a = S[i].y - S[j].y
            b = S[j].x - S[i].x
            c = S[i].x * S[j].y - S[j].x * S[i].y
            positivos = 0
            negativos = 0
            Para k=0 até n
                Se a * S[k].x + b * S[k].y - c >= 0
                    positivos++
                Se a * S[k].x + b * S[k].y - c <= 0
                    negativos++
            Se negativos = n ou positivos = n
                Adiciona S[i] e S[j] em C
    Devolva C

Procedimento Mesclar_Fechos
    Entrada: S1 e S2, dois fechos convexos, de tamanhos m e n, respectivamente.
    Saída: C, mesclagem de S1 e S2
    Adicione S1 e S2 em C
    a = tangente superior que conecta S1 e S2
    b = tangente inferior que conecta S1 e S2
    Adicione retas a e b em C
    Remova todos os pontos que não fazem parte das bordas do fecho convexo C
    Devolva C
```

A complexidade temporal do Merge Hull é O(n log n). A complexidade espacial é O(n).

2) **(2 pontos)** Um fazendeiro vem utilizando um tipo de fertilizante que faz com que o pasto cresça muito rapidamente, de modo que as vacas não precisam percorrer o pasto em busca de alimento. Num pasto, representado por um vetor de *n* posições, estão *k* vacas, sendo que cada vaca ocupa uma posição do pasto diferente das demais vacas. Com a chegada do inverno, o fazendeiro decide construir *m* estábulos para abrigar as vacas. No entanto, ele deseja que os estábulos ocupem a menor quantidade possível de posições do pasto, de modo a minimizar os custos de construção. Por exemplo, num pasto com 20 posições e 4 vacas dispostas nas posições indicadas abaixo, a solução mais econômica para construir 2 estábulos é a seguinte:

		v_1									v_2	v_3		v_4					

Escreva uma função baseada em **programação dinâmica** que receba *n*, *m*, v_1, v_2, ..., v_k (v_i representa a posição da vaca *i* no pasto) e então indique como construir *m* estábulos para abrigar as *k* vacas de modo que a quantidade de posições cobertas pelos estábulos seja a menor possível. Determine a complexidade temporal e espacial da função e diga se ela é eficiente e se é de cota inferior.

Algoritmo Pasto_PD

Entrada: m, número de estábulos; k, número de vacas; v, vetor com as posições das vacas no pasto

Saída: P (m, k)

```
Para j = 1 até k
        P (j, 1) = v_j – v_{j+1}
Para i = 2 até m
        Para j = 1 até k
                P (i, j) = min (1 + P (i - 1, j - 1), P (i-1, j) + v_j - v_{j-1})
Devolva P (m, k)
```

A complexidade temporal do algoritmo é θ(mk). A complexidade espacial é θ(k).
Como a complexidade temporal do algoritmo está limitada a ordem do tamanho da entrada, ele é eficiente.
O algoritmo Pasto_PD não é de cota inferior, porque a velocidade pode ser aumentada utilizando um heap.

3) (**2 pontos**) Escreva um algoritmo baseado em **enumeração explícita** que receba uma coleção de números *C* e um valor *K* e devolva uma coleção contida em *C* cuja soma dos elementos seja a mais próxima de *K*. Por exemplo, para *C* = (3, 8, 4, 3) e *K* = 2 a resposta seria (3). Para *K* = 7 a resposta poderia ser (3, 3) ou (8). Determine a complexidade temporal e espacial de seu algoritmo.

Algoritmo mais_proximo

Entrada: Um conjunto de números c, de n números; Um valor k

Saída: Subconjunto de c cuja soma dos elementos é o mais próximo de k

```
diferenca = +∞
mais_prox = 0
Para i = 0 até 2^n – 1
        total = soma_sequencia(c, i)
        Se |k – total| < diferenca
                diferenca = |k – total|
                mais_prox = i
Devolva mais_prox
```

Função soma_sequencia

Entrada: Um vetor c, de n números; Um valor i que codifica uma subsequência de c

Saída: Soma de todos os elementos da subsequência de c codificada por i

```
soma=0
Para j = 1 até n
        se i % 2 = 1
                soma = soma + c[j]
        j++
        i = chão(i/2)
Devolva soma
```

A complexidade temporal do algoritmo mais_proximo é $\theta(n*2^n)$ e a complexidade espacial é O(1)

4) (**2 pontos**) O *problema da mochila inteira* é similar ao *problema da mochila binária*, com uma diferença: cada item pode ser colocado diversas vezes na mochila. Mostre como adaptar o algoritmo de *branch-and-bound* para o problema da mochila binária, visto em sala de aula, para o problema da mochila inteira.

Algoritmo PMI_BB

Entrada: c (capacidade da mochila), $p_1, p_2, ..., p_n$ (pesos dos itens), $v_1, v_2, ..., v_n$ (valores dos itens)

Saída: x* (solução ótima)

max = -1, k = 0, x [ ] = 0

Ordene os itens em ordem decrescente de valor relativo (valor/peso)

Faça

Para i = k + 1 até n

x[i] = $\lfloor (c - \sum_{j=1}^{i-1} p_j x_j)/p_i \rfloor$

Se $\sum_{j=1}^{n} v_j x_j > max$

x* = x

max = $\sum_{j=1}^{n} v_j x_j$

k = $max\left(\left\{i | i < n \ e \ \mathrm{x[i]} > 0 \ e \ \left[\sum_{j=1}^{i-1} v_j x_j + v_i(x_i - 1) + \left(c - \sum_{j=1}^{i-1} p_j x_j - p_i(x_i - 1)\right)\right] > max\right\} \cup \{0\}\right)$

Se k > 0

$x_k = x_k + 1$

Enquanto k > 0

Devolva x*

5) (**3 pontos**) No *problema da mochila fracionária* temos uma mochila de capacidade C e n itens de pesos $p_1, p_2, ..., p_n$ e valores $v_1, v_2, ..., v_n$. Podemos colocar uma fração de cada item na mochila (por exemplo, um dos itens poderia ser um saco de arroz de 10 kg e podemos colocar 20% desse saco na mochila). Queremos decidir que fração colocar de cada item sem exceder sua capacidade e maximizando o valor da mochila. Por exemplo, numa mochila de capacidade C = 10 e quatro itens cujos pesos e valores aparecem na tabela a seguir, a solução ótima é colocar 60% do item 2, 100% do item 4 e nada dos demais itens. O valor da mochila seria 98,8.

Item	1	2	3	4
Peso	8	5	6	7
Valor	72	48	57	70

Escreva um algoritmo ***guloso*** que resolva o problema da mochila fracionária. Determine a complexidade temporal e espacial do seu algoritmo.

Algoritmo Mochila_Fracionaria

Entrada: c (capacidade da mochila), $p_1, p_2, ..., p_n$ (pesos dos itens), $v_1, v_2, ..., v_n$ (valores dos itens)

Saída: vetor x, que maximize x*v, sob as restrições x*v ≤ p e 0 ≤ x_i ≤ 1, para todo i

Coloque os itens em ordem crescente de valor relativo (valor/peso)

Para i = 1 até n

Se $p_j \leq c - \sum_{j=1}^{i-1} p_j x_j$

x[i]= 1

Se não

x[i] = $(c - \sum_{j=1}^{i-1} p_j x_j)/ p_i$

Devolva x

A complexidade temporal desse algoritmo é θ(n*logn)

A complexidade espacial deste algoritmo é O(1)