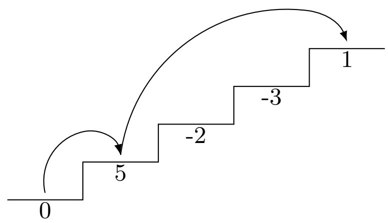
Universidade Estadual de Maringá
Departamento de Informática – Ciência da Computação
6889 – Projeto e Análise de Algoritmos / Prof. Daniel Kikuti

Aluno(a): ______________________________

Segunda avaliação (Valor: 10,0)

1. [Valor: 3,0] Considere uma escada com n degraus. Cada degrau está associado a uma recompensa ou penalidade representada por um valor inteiro. Seu objetivo é sair do degrau 0 e chegar no n-ésimo degrau usando um conjunto de m possíveis escolhas (em cada degrau), de modo a maximizar o total obtido.

   Por exemplo, suponha que $n = 4$ e os movimentos possíveis são $M = \{1, 2, 3\}$, isto é, ir para o próximo degrau, avançar dois degraus ou avançar três degraus. Para a escada a seguir, a solução ótima seria ir para o degrau 1 e depois para o degrau 4, totalizando $0 + 5 + 1 = 6$.



   (a) [Valor: 1,0] Mostre que este problema possui subestrutura ótima.

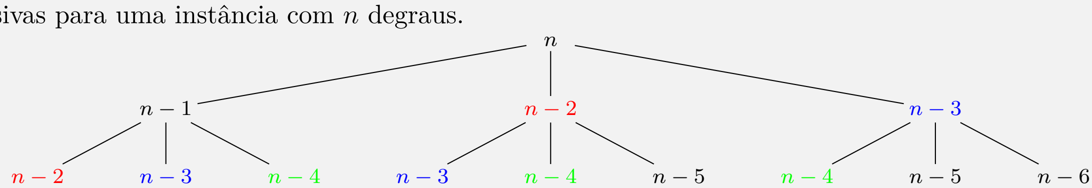
   Solução:
   Seja $opt(n)$ a solução ótima (maior valor) para uma escada com n degraus e considerando m escolhas possíveis em cada degrau.

$$opt(n) = \begin{cases} A[0] & \text{se } n = 0 \\ max_{m \in M}(A[n] + opt(n-m)) & \text{se } n \geq m. \end{cases}$$

   Para mostrar que este problema possui subestrutura ótima, considere a sequência de escolhas feitas nos degraus $S_i^* = \{m_1, \ldots, m_i\}$ que levam ao valor ótimo $opt(n)$. Vamos argumentar que $S_{i-1}^* = \{m_1, \ldots, m_{i-1}\}$ é a solução ótima para o subproblema contendo $k = n - m_i$ degraus. Suponha que existisse uma outra sequência de escolhas $S'_j = \{m'_1, \ldots, m'_j\}$ chegando no k-ésimo degrau e com valor estritamente maior que o da sequência S_{i-1}^*, isto é, $opt(k)' > opt(k)$. Assim, poderíamos usar a sequência S'_j seguida do movimento m_i para chegar ao n-ésimo degrau e obter um valor total $opt(k)' + A[n] > opt(n)$, o que contraria o fato de $opt(n)$ ser o valor ótimo.

   (b) [Valor: 1,0] Mostre que há sobreposição de problemas.

   Solução:
   Considere que os movimentos possíveis são $M = \{1, 2, 3\}$. A árvore a seguir apresenta algumas chamadas recursivas para uma instância com n degraus.



   (c) [Valor: 1,0] Apresente um algoritmo que usa programação dinâmica.

   Solução:
   Assuma que:

   - $A[0 \ldots n]$ vetor global contendo valores das recompensas ou penalidades;
   - $M[1 \ldots m]$ é um vetor global contendo os movimentos possíveis;
   - $memo[0 \ldots n]$ é um vetor global com $n + 1$ posições previamente inicializado com $-\infty$;

```
Stairs(n)
if memo[n] = −∞ then
    if n = 0 then
        ans ← A[0]
    else
        ans ← −∞
        for i ← 1 to m do
            if n ≥ M[i] then
                ans ← max(ans, A[n] + Stairs(n − M[i]))
    memo[n] ← ans
return memo[n]
```

2. [Valor: 4,0] Suponha que você está gerenciando a construção de painéis publicitários (*outdoors*) em uma rodovia com K quilômetros. Os locais para instalação dos painéis são dados por números $x_1, x_2, \ldots, x_n$ (distância do início da rodovia), todos no intervalo $[0, K]$. Se você instalar um painel no local x_i, você receberá uma recompensa $r_i > 0$. A regulamentação da rodovia impõe que dois painéis não podem estar próximos um do outro, isto é, não podem estar a uma distância menor ou igual a 5 km. Você deseja instalar os painéis em um subconjunto do conjunto de possíveis locais, de modo a maximizar sua recompensa, respeitando a restrição de distância.

   Exemplo: Suponha que $K = 20$, $n = 4$, $\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$ e $\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$. Então, a solução ótima seria instalar os painéis em x_1 e x_3, com uma recompensa total de 10.

   (a) [Valor: 1,5] Escreva um algoritmo guloso para este problema.

   Solução:
   Uma escolha gulosa para este problema seria instalar o painel com maior recompensa dentre os possíveis.

```
Guloso(x, r)
Seja S o conjunto de painéis com distância compatível com todos os painéis que já foram
  instalados (inicialmente S contém todos os elementos).
total ← 0
while S ≠ ∅ do
    Encontre o i-ésimo elemento em S que possui a maior recompensa.
    total ← total + r[i]
    Remova de S o elemento i e todos os painéis que possuem distância de i menor ou igual a 5.
return total
```

   (b) [Valor: 1,5] Explique em que situações a estratégia gulosa que você definiu geraria bons resultados e quando ela poderia falhar (se é que ela falha) ao encontrar o ótimo.

   Solução:
   O algoritmo devolveria a solução ótima quando não há painéis com distância menor ou igual a 5 ou quando a soma das recompensas dos painéis conflitantes com a escolha gulosa não ultrapassa o valor de sua recompensa. Por exemplo: $x = \{6, 7, 12\}$ e $r = \{4, 8, 3\}$ devolveria a solução ótima (instalar painel em $x_2 = 7$ com lucro de 8).

   A escolha gulosa não funcionaria para casos como: $x = \{6, 7, 12\}$ e $r = \{4, 8, 5\}$, pois devolveria uma solução com lucro de 8, enquanto que a solução ótima seria instalar painéis em x_1 e x_3 com lucro de 9.

   (c) [Valor: 1,0] Apresente uma formulação recursiva para encontrar o maior valor de recompensa total possível.

   Solução:
   Usaremos a mesma ideia usada no problema de intervalos ponderados. Consideraremos que cada painel i corresponde a um intervalo $[x_i - 5, x_i + 5]$ com peso r_i (se for necessário, desloque todos os painéis em 5km para não ficar com valores negativos). Ordene os painéis por término e assuma que a função $p(n)$ devolve o maior índice de um painel compatível com n. Assim, a formulação recursiva considera se o n-ésimo painel faz ou não parte da solução ótima.

$$opt(n) = \begin{cases} 0 & \text{se } n = 0 \\ max(r[n] + opt(p(n)), opt(n-1)) & \text{se } n > 0. \end{cases}$$

3. [Valor: 3,0] No problema da subsequência comum mais longa (*Longest Commom Subsequence – LCS*), são dadas duas *sequências* e o objetivo é encontrar a maior subsequência comum a ambas. Para dificultar um pouco as

coisas, esta questão pede para você encontrar a subsequência comum mais longa não apenas de duas, mas de três sequências dadas (X, Y e Z)

Exemplos:

- $X = abc$, $Y = bca$ e $Z = bac$. A subsequência comum mais longa é bc com tamanho 2.
- $X = abcde$, $Y = acdbe$ e $Z = beacd$. A subsequência comum mais longa é acd com tamanho 3.

(a) [Valor: 1,0] Escreva um algoritmo que usa a técnica de programação dinâmica para devolver o tamanho da subsequência comum mais longa.

Solução:
Seja $opt(i, j, k)$ a quantidade de elementos da maior subsequência comum a $X[1 \dots i]$, $Y[1 \dots j]$ e $Z[1 \dots k]$.

$$opt(i,j,k) = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \text{ ou } k = 0, \\ 1 + opt(i-1, j-1, k-1) & \text{se } X[i] = Y[j] = Z[k], \\ max(opt(i-1,j,k), opt(i,j-1,k), opt(i,j,k-1)) & \text{caso contrário.} \end{cases}$$

Se $|X| = m$, $|Y| = n$ e $|Z| = l$, então $memo$ é uma matriz $(m+1) \times (n+1) \times (l+1)$ tal que cada célula (i, j, k) contém o valor da solução ótima para $X[1 \dots i]$, $Y[1 \dots j]$ e $Z[1 \dots k]$. $memo2$ é uma matriz auxiliar de mesmas dimensões de $memo$ usada para reconstruir a solução (qual a subsequência comum mais longa). Assuma que:

- $memo$ é global e foi previamente incializado com -1;
- $memo2$ é global e foi previamente incializado com -1;
- X, Y e Z também são globais.

```
LCS(i, j, k)
if memo[i][j][k] = −1 then
    if i = 0 ou j = 0 ou k = 0 then
        memo[i][j][k] ← 0
    else if X[i] = Y[j] = Z[k] then
        memo[i][j][k] ← 1 + LCS(i − 1, j − 1, k − 1)
        memo2[i][j][k] ← '='
    else
        auxX ← LCS(i − 1, j, k)
        auxY ← LCS(i, j − 1, k)
        auxZ ← LCS(i, j, k − 1)
        if auxX > auxY e auxX > auxZ then
            memo[i][j][k] ← auxX
            memo2[i][j][k] ← 'x'
        else if auxY > auxX e auxY > auxZ then
            memo[i][j][k] ← auxY
            memo2[i][j][k] ← 'y'
        else
            memo[i][j][k] ← auxZ
            memo2[i][j][k] ← 'z'
return memo[i][j][k]
```

(b) [Valor: 1,0] Analise a complexidade do seu algoritmo.

Solução:
A complexidade do algoritmo memoizado é dada pela quantidade de subproblemas distintos vezes o tempo gasto em cada subproblema considerando que as chamadas recursivas tem custo constante. Ou seja, o algoritmo LCS tem custo $O(m \cdot n \cdot l)$.

(c) [Valor: 1,0] Escreva um algoritmo que imprime quais são os caracteres que fazem parte da subsequência comum mais longa.

Solução:

```
PrintLCS(i, j, k)
if i > 0 e j > 0 e k > 0 then
    if memo2[i][j][k] = '=' then
        PrintLCS(i − 1, j − 1, k − 1)
        Imprima X[i]
    else if memo2[i][j][k] = 'x' then
        PrintLCS(i − 1, j, k)
    else if memo2[i][j][k] = 'y' then
        PrintLCS(i, j − 1, k)
    else
        PrintLCS(i, j, k − 1)
```

Boa Prova!!!