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: 4,0] Dadas duas palavras x e y, a distância de edição entre as duas é definida como sendo o número mínimo de caracteres que precisam ser substituídos, removidos ou adicionados para transformar uma em outra. Exemplos:

   - "*cat*" e "*bat*". As duas palavras diferem apenas na primeira letra. Basta substituirmos o '*c*' de *cat* por um '*b*' para chegar em *bat*. Portanto, a distância de edição é 1.
   - As palavras "fly" e "$flying$" são idênticas nos três primeiros caracteres, mas a segunda palavra possui três caracteres adicionais. Adicionando "ing" na primeira palavra produz a segunda palavra. A distância de edição neste caso é 3.
   - Considere as palavras "*grave*" e "*groovy*". Podemos efetuar as seguintes substituições na primeira palavra: (1) '*a*' → '*o*', (2) '*e*' → '*y*', depois (3) insira a letra '*o*' na posição 4 (após o primeiro '*o*'). Portanto, a distância de edição neste caso é 3.

   (a) [Valor: 0,8] Apresente uma formulação recursiva para o problema. Mostre que o problema possui subestrutura ótima.

Solução:
Sejam $X_i = \{x_1, \ldots, x_i\}$ e $Y_j = \{y_1, \ldots, y_j\}$ prefixos de X e Y respectivamente. Seja $opt(i, j)$ o número mínimo de caracteres que precisam ser substituídos, removidos ou adicionados para transformar Y_j em X_i. Existem dois casos a serem considerados:

1. Se $X[i] = Y[j]$, então a solução ótima para X_i e Y_j será igual à solução ótima para X_{i-1} e Y_{j-1}.
2. Se $X[i] \neq Y[j]$, então precisamos considerar as operações possíveis (substituição, remoção e inserção) e escolher aquela que devolve o menor número de operações para transformar uma string na outra (consideramos que as operações são feitas em X).
   - Substituição: $X[i]$ será substituída pelo valor de $Y[j]$ e o subproblema a ser resolvido será encontrar o ótimo considerando os prefixos X_{i-1} e Y_{j-1}.
   - Remoção: $X[i]$ será removido e o subproblema a ser resolvido será encontrar o ótimo considerando os prefixos X_{i-1} e Y_j.
   - Inserção: insere-se o caractere $Y[j]$ após o caractere $X[i]$, portanto, o subproblema a ser resolvido será encontrar o ótimo considerando os prefixos X_i e Y_{j-1}.

$$opt(i,j) = \begin{cases} \text{i} & \text{se } j = 0, \\ \text{j} & \text{se } i = 0, \\ opt(i-1, j-1) & \text{se } X[i] = Y[j] \text{ e } i, j > 0, \\ 1 + \min(opt(i-1, j-1), opt(i-1, j), opt(i, j-1)) & \text{se } X[i] \neq Y[j] \text{ e } i, j > 0. \end{cases}$$

Para mostrar que há subestrutura ótima, precisamos mostrar que a solução ótima para o problema contém dentro dela soluções ótimas para os subproblemas. Assuma que $opt(i, j)$ é a solução ótima para os prefixos X_i e Y_j.
Precisamos então considerar cada uma das possibilidades. No primeiro caso ($X[i] = Y[j]$), suponha que exista uma solução $opt'(i-1, j-1)$ para os prefixos X_{i-1} e Y_{j-1} que é estritamente menor que o valor devolvido por $opt(i-1, j-1)$ (isto é, existe uma forma de transformar Y_{j-1} em X_{i-1} que usa menos operações). Então se substituirmos $opt(i-1, j-1)$

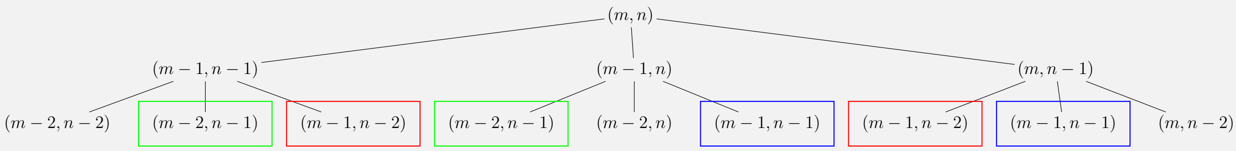
por $opt'(i-1, j-1)$, conseguiríamos resolver o problema para os prefixos X_i e Y_j com menos operações (pois, $X[i] = Y[j]$ e não seria necessário efetuar nenhuma operação adicional, logo $opt(i,j) = opt(i-1, j-1) > opt'(i-1, j-1)$), o que contradiz o fato de que $opt(i,j)$ é ótimo. [um exemplo aqui é *bat* e *cat*]

Para cada um dos três outros casos, o mecanismo de demonstração é o mesmo (recortar e colar). Considere situações em que a solução ótima efetua determinadas operações, por exemplo:

- Para substituição considere $x = aaa$ e $y = aab$.
- Para remoção considere $x = aaaa$ e $y = aaa$.
- Para inserção considere $x = aaa$ e $y = aaaa$.

(b) [Valor: 0,8] Mostre que há sobreposição de problemas.

Solução:
Considerando o pior caso, cada nó ramifica em três chamadas recursivas.



(c) [Valor: 0,8] Apresente um algoritmo que faz uso da técnica de programação dinâmica.

Solução:

Assuma que *memo* é uma matriz global de dimensões $(m+1) \times (n+1)$ previamente inicializada com -1 em todas as posições. Assuma que X e Y também são variáveis globais representando as duas strings.

DP(i, j)

1 **if** $memo[i][j] = -1$ **then**
2   **if** $i = 0$ *ou* $j = 0$ **then**
3   **if** $i = 0$ **then** $memo[i][j] \leftarrow j$
4   **else** $memo[i][j] \leftarrow i$
5   **else if** $X[i] = Y[j]$ **then**
6         $memo[i][j] \leftarrow \text{DP}(i-1, j-1)$
7   **else**
8         $aux \leftarrow \text{DP}(i-1, j-1)$
9         $aux \leftarrow \min(aux, \text{DP}(i-1, j))$
10         $aux \leftarrow \min(aux, \text{DP}(i, j-1)$
11         $memo[i][j] \leftarrow aux + 1$
12 **return** $memo[i][j]$

(d) [Valor: 0,8] Analise a complexidade do seu algoritmo.

Solução:
Existem $(m+1) \times (n+1)$ problemas distintos e tempo gasto em cada subproblema desconsiderando as recursões é $\Theta(1)$. Portanto, a complexidade do algoritmo será $\Theta(mn)$.

(e) [Valor: 0,8] Descreva um algoritmo que informa quais as operações devem ser feitas para transformar uma palavra em outra.

Solução:
Usaremos uma ideia parecida com a do algoritmo *LCS* (*Longest Common Subsequence*). Armazenaremos em uma estrutura auxiliar M qual a escolha feita. Alteraremos a linha

6 (guardando também o símbolo '$*$' na estrutura auxiliar) e as linhas $8, 9$ e 10 para a armazenar os caracteres: s para substituição, r para remoção e i para inserção. O algoritmo será:

Print(i,j)

1 **if** $i = 0$ *ou* $j = 0$ **then**
2   **if** $i = 0$ **then** Imprima: Insira os caracteres $Y[1] \dots Y[j]$ no começo de X
3   **else** Imprima: Remova os caracteres $X[1] \dots X[i]$ de X
4 **else if** $M[i][j] =$ '$*$' **then**
5   Print$(i-1, j-1)$
6 **else if** $M[i][j] =$'s' **then**
7   Print$(i-1, j-1)$
8   Imprima: Substitua $X[i]$ por $Y[j]$
9 **else if** $M[i][j] =$ 'r' **then**
10   Print$(i-1, j))$
11   Imprima: Remova $X[i]$
12 **else**
13   Print$(i, j-1)$
14   Imprima: Insira $Y[j]$ após de $X[i]$

2. [Valor: 3,0] "Para quê serve Programação Dinâmica?" Certamente, para os alunos de Ciência da Computação, a resposta para esta questão é tão fundamental como a resposta para: "O que vai ter para o almoço no RU?" Para saciar sua ansiedade, irei responder (a primeira pergunta) de maneira bem simples.

É um conteúdo que serve para formar um preguiçoso mais esperto e eficiente. Como assim? Nestas alturas do campeonato, provavelmente você já deve ter lido boa parte das questões e percebido que cada questão está associada a um valor e a uma dificuldade (estimada em alguma unidade de tempo). Talvez também já tenha passado por sua cabeça: "tenho que tirar x nesta prova". Assim, seu problema seria minimizar a quantidade de tempo gasto para alcançar ao menos a nota necessária.

Fácil? Hum... não muito. Testar todas as possibilidades pode demorar mais tempo que o tempo total de prova (dependendo do número de questões). Moral da história: é bom aprender Programação Dinâmica se quiser ficar folgado.

Exemplo:

- Para $x = 6$ e a tabela de questões a seguir, o tempo mínimo seria 4.

questões	1	2	3	4
pontos	1	2	3	4
dificuldade	2	2	2	2

Considerando o enunciado do problema acima, faça:

(a) [Valor: 1,5] Apresente um algoritmo que faz uso da técnica de programação dinâmica e que resolve o problema (devolve o tempo mínimo necessário para se obter a pontuação).

Solução:
Considerações iniciais para o problema:

- Sempre é possível alcançar um valor maior ou igual a x na prova (caso contrário, se a soma de todas as questões fosse menor que x, então o aluno nem teria motivos para fazer a prova).
- Não foi estipulado um tempo máximo de prova (ou seja, o aluno poderia demorar tempo igual à soma de todos os tempos estimados para cada questão).
- O aluno é indiferente entre tirar uma nota maior que x ou exatamente igual a x.

Sejam $opt(i, j)$ o tempo mínimo necessário para obter uma pontuação maior ou igual a j, $p[]$ e $d[]$ vetores contendo respectivamente a pontuação e dificuldade de cada questão. Usaremos a seguinte formulação recursiva:

$$opt(i,j) = \begin{cases} 0 & \text{se } j = 0, \\ \infty & \text{se } i = 0 \text{ e } j > 0, \\ \min(d[i], opt(i-1,j)) & \text{se } p[i] \geq j, \\ \min(d[i] + opt(i-1, j-p[i]), opt(i-1,j)) & \text{se } p[i] < j. \end{cases}$$

Para o algoritmo memoizado a seguir, assuma que $memo[i][j]$ foi previamente inicializado com -1 em todas as posições.

```
DP(i, j)
if memo[i][j] = −1 then
    if j = 0 then memo[i][j] ← 0
    else if i = 0 then memo[i][j] ← ∞
    else
        if p[i] ≥ j then
            memo[i][j] ← min(d[i], DP(i − 1, j))
        else
            memo[i][j] ← min(d[i] + DP(i − 1, j − p[i]), DP(i − 1, j))
return memo[i][j]
```

(b) [Valor: 1,5] Mostre que o algoritmo guloso que seleciona dentre as questões restantes aquela com maior relação ponto/dificuldade (até obter um valor maior ou igual a x) nem sempre devolve a solução ótima.

Solução:
Para mostrar que a escolha gulosa não funciona, considere a seguinte instância de entrada com os valores da relação pontos/dificuldade apresentados na última linha:

questões	1	2	3
pontos	1	3	8
dificuldade	1	2	4
p_1/t_1	1	1.5	2

Suponha que a nota necessária é $x = 4$, então a solução ótima seria fazer as questões 1 e 2 (com dificuldade total igual a 3). A solução gulosa iria sugerir a questão 3 como solução.

3. [Valor: 3,0] Carlinhos é um garoto viciado em doces. Ele é assinante da *All Candies Magazine* (ACM) e foi selecionado para participar da *International Candy Picking Contest* (ICPC).

Nessa competição, um número aleatório de caixas contendo doces são dispostas em M linhas com N colunas cada (existe um total de $M \times N$ caixas). Cada caixa tem um número indicando quantos doces ela contém.

O competidor pode escolher uma caixa (qualquer uma) e pegar todos os doces dentro dela. Mas existe uma sacada (sempre existe uma sacada): quando uma caixa é escolhida, todas as caixas das linhas logo acima e logo abaixo são esvaziadas, assim como as caixas à direita e à esquerda da caixa escolhida. O competidor continua pegando uma caixa até que não hajam mais doces.

A figura abaixo ilustra isso, passo a passo. Cada célula representa uma caixa e o número de doces que ela contém. A cada passo, a caixa escolhida é circulada e as células sombreadas representam as caixas que serão esvaziadas. Após oito etapas o jogo acaba e Carlinhos pegou $10 + 9 + 8 + 3 + 7 + 6 + 10 + 1 = 54$ doces.

1	8	2	1	9
1	7	3	5	2
1	2	10	3	10
8	4	7	9	1
7	1	3	1	6

1	8	2	1	9
0	0	0	0	0
1	0	0	0	10
0	0	0	0	0
7	1	3	1	6

1	8	2	0	0
0	0	0	0	0
1	0	0	0	10
0	0	0	0	0
7	1	3	1	6

0	0	0	0	0
0	0	0	0	0
1	0	0	0	10
0	0	0	0	0
7	1	3	1	6

0	0	0	0	0
0	0	0	0	0
1	0	0	0	10
0	0	0	0	0
7	0	0	0	6

0	0	0	0	0
0	0	0	0	0
1	0	0	0	10
0	0	0	0	0
0	0	0	0	6

0	0	0	0	0
0	0	0	0	0
1	0	0	0	10
0	0	0	0	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
1	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Considerando o enunciado do problema acima, faça:

(a) [Valor: 1,5] Apresente um algoritmo que faz uso da técnica de programação dinâmica e que resolve o problema de maximizar a quantidade de doces.

Solução:

Observe o que ocorre quando um célula (i, j) da matriz é selecionada para fazer parte da solução do problema. Note que se a matriz tem dimensões maiores que 3×3, ainda sobrarão células com doces na mesma linha que poderíamos pegar sem "apagar" mais doces das linhas $i + 1$ e $i - 1$ (visto que eles já foram apagados quando escolhemos (i, j)). Então, em cada linha queremos maximizar a quantidade de doces que conseguimos pegar sabendo que as células imediatamente vizinhas $(i, j + 1)$ e $(i, j - 1)$ serão zeradas.

Considerando que guardamos a informação do máximo por linha em um vetor $vet[]$, ideia aqui é análoga à explicação anterior, isto é, não podemos pegar doces em posições vizinhas neste vetor.

Podemos então propor um algoritmo de PD para o problema como segue:

Doces(vet[], n)

$memo[0] \leftarrow 0$
$memo[1] \leftarrow vet[1]$
for $i \leftarrow 2$ **to** $i <= n$ **do**
    $memo[i] \leftarrow max(memo[i-1], vet[i] + memo[i-2])$
return $memo[n]$

Na função $main()$ teríamos uma chamada $maxline[i] = \text{Doces}(vet, n)$ para cada linha i e, após guardar os máximos por linha, uma chamada devolvendo $\text{Doces}(maxline, n)$.

(b) [Valor: 1,5] Apresente um algoritmo guloso para este problema. Evidencie qual é a escolha gulosa. Argumente que seu algoritmo guloso está correto ou mostre uma situação em que ele não é capaz de devolver a resposta correta.

Solução:

Um algoritmo guloso para este problema seria:

Doces(m)

$total \leftarrow 0$
while *há doces* **do**
    Selecione a célula m com maior quantidade de doces
    $total \leftarrow total + m$
    Aplique as regras do jogo
return $total$

Este algoritmo guloso nem sempre produz a resposta correta. Para visualizar isto, considere a seguinte instância:

5	5	5
5	10	5
5	5	5

5	5	5
5	10	5
5	5	5

A solução ótima para o problema seria pegar os doces nas posições $(1,1)$, $(1,3)$, $(3,1)$ e $(3,3)$, totalizando 20 doces.

Boa Prova!!!