Universidade Federal do Rio Grande

Centro de Ciências Computacionais

Disciplina de Algoritmos e Estruturas de Dados II
Prof. Rodrigo de Bem

Lista de Exercícios – Técnicas de Resolução

INSTRUÇÕES:

1. Os exercícios práticos poderão ser implementados na linguagem de programação de sua preferência.

EXERCÍCIOS PRÁTICOS:

1) Implementação do algoritmo Merge Sort():

Entrada: vetor A, de tamanho n, com números reais $a_1, a_2, \ldots, a_n$;
Saída: vetor A reordenado de modo que $a_1' \leq a_2' \leq \ldots \leq a_n'$.

O algoritmo Merge Sort emprega a técnica de **divisão-e-conquista** da seguinte forma:

1. Cada subproblema é definido como a ordenação de um sub-vetor A[p . . r]. Inicialmente, p = 1 e r = n, mas esses valores mudam ao longo das chamadas recursivas.
2. Para ordenar A[p..r]:
    a. **Divide-se** o vetor A[p..r] em dois sub-vetores A[p..q] e A[q+1..r], sendo q o elemento central de A[p..r].
    b. **Conquista-se** resolvendo recursivamente os sub-vetores A[p..q] e A[q+1..r].
    c. **Combina-se** os dois sub-vetores ordenados A[p..q] e A[q+1..r] para obter um único vetor ordenado A[p..r]. Este passo é realizado pelo procedimento MERGE(A,p,q,r).

A recursão encerra quando o sub-vetor tem apenas 1 elemento, sendo assim trivialmente ordenado.

Algoritmo Merge Sort:

```
MERGE-SORT(A, p, r )
if  p < r then                //Verifica o caso base
    q ← ⌊(p + r)/2⌋                 //Divide
    MERGE-SORT(A, p, q)             //Conquista
    MERGE-SORT(A, q + 1, r )        //Conquista
    MERGE(A, p, q, r )              //Combina
```

Chamada inicial: `MERGE-SORT(A, 1, n)`

Procedimento MERGE:

- **Entrada**: Vetor *A e índices p, q, r tais que*
    - *p ≤ q < r .*
    - Os sub-vetores *A[p..q] e A[q + 1..r] estão ordenados. Pelas restrições em p, q e r, nenhum sub-vetor é vazio.*
- **Saída**: Os dois sub-vetores são unidos em um único vetor *A[p..r].*

```
MERGE(A, p, q, r )
n1 ← q − p + 1
n2 ← r − q
//Cria vetores L[1 . . n1 + 1] e R[1 . . n2 + 1]
for i ← 1 to n1 do
   L[i] ← A[p + i − 1]
for j ← 1 to n2 do
   R[ j ] ← A[q + j ]
L[n1 + 1]←∞
R[n2 + 1]←∞
i ← 1
```

```
j ← 1
for k ← p to r do
   if L[i] ≤ R[j] then
      A[k] ← L[i]
      i ←i + 1
   else A[k] ← R[ j ]
        j ← j + 1
```

2) Implementação do algoritmo <u>guloso</u> para solução do problema do troco:

Resolver o **<u>problema do troco</u>** consiste em, dada uma certa quantia a ser devolvida como troco a um cliente de uma birosca, calcular a menor quantidade de moedas a serem usadas, dado um conjunto de moedas existentes.

<u>Entrada:</u> conjunto de moedas existentes (conjunto C) e valor a ser dado como troco.
<u>Saída:</u> menor número possível de moedas a serem usadas como troco e quantas moedas de cada tipo foram utilizadas.

```
function Make_Change(n): set of coins
const C = {100, 25, 10, 5, 1};
S ← ∅; // S é o conjunto solução
s ← 0; // soma dos itens em S
while (s ≠ n) do//enquanto a solução não é alcançada
        x ← the largest item in C such that s+x ≤ n;
         if (there is no such item) then
            return "no solution found";
         S ← S ∪ {a coin of value x};
         s ← s + x;
end;
return S;
end;
```

3) Implementação da solução do problema da mochila 0-1 usando <u>programação dinâmica:</u>

Um ladrão entra em uma joalheria e tem a seu alcance n objetos de valor para colocar em sua mochila. Para i=1,2,...,n, o objeto i tem peso wi e um valor positivo vi. A mochila pode carregar um peso no máximo igual a W. Assim, o objetivo é encher a mochila de modo a maximizar o valor dos objetos roubados. Neste problema os objetos não podem ser fracionados.
O objetivo é maximizar $\sum_{i=1}^{n} x_i v_i$, respeitando a restrição de que $\sum_{i=1}^{n} x_i w_i \leq W$, sendo vi>0, wi>0 e xi∈{0,1}, para 1 ≤ i ≤ n.

<u>Entrada:</u> vetor de pesos w dos n objetos, vetor de valores v dos n objetos, capacidade W da mochila.
<u>Saída:</u> tabela V com a solução ótima de todos os possíveis subproblemas.

```
function knapsack(w[1..n], v[1..n], W)
array V[1..n,0..W];  //tabela V com a solução ótima dos subproblemas
for i←1 to n do
     V[i,0] ← 0;
for i←1 to n do
     for j←1 to W do
          if i=1 and j ≥ w[i] then
             V[i,j] ← v[i];
```

```
        else if i=1 and j < w[i] then
            V[i,j] ← 0;
        else if j < w[i] then
            V[i,j] ← V[i-1,j];
        else
            V[i,j] ← MAX(V[i-1,j], V[i-1,j-w[i]] + v[i]);
return V[n,N];
```

4) Implementação da solução do problema da mochila 0-1 usando <u>busca exaustiva (backtracking):</u>

Considere a mesma descrição do problema anterior. Nessa estratégia todas as possíveis soluções são avaliadas através de um algoritmo recursivo.

```
main procedure knapsack()
array w[1..n];       // pesos dos objetos
array v[1..n];       // valores do objetos
int W;               // capacidade total da mochila

for i=1 to n do      //inicialização
    x[i] ← 0;

// busca recursiva do melhor carregamento de i até n itens com peso máximo r
function backpack(i, r)
        best ← 0;
        for k ← i to n do // p
            if w[k] ≤ r and x[k] ≠ 1 then
               x[k] ← 1;
              //verificação da melhor opção: com o objeto k ou sem o objeto k
               best ← MAX(best, v[k] + backpack(k, r-w[k]));
               x[k] ← 0;
            end if;
  return best;
 end function;

 imprima backpack(1, W);
 end main procedure.
```

<u>EXERCÍCIOS TEÓRICOS:</u>

1. Descreva de maneira objetiva e coesa, com suas palavras, a estratégia de <u>divisão-e-conquista</u> para solução de problemas.

2. Os algoritmos que empregam a técnica de divisão-e-conquista têm suas funções de tempo de execução definidas por recorrências. Escreva a forma geral dessas funções e descreva o significado de cada um de seus termos.

3. Descreva resumidamente cada um dos métodos abordados no curso para resolução de recorrências.

4. Cite 3 algoritmos, **<u>não</u>** abordados no curso, com suas respectivas finalidades, que empregam a estratégia de divisão-e-conquista para solução de problemas.

5. Observe que o `while` (linhas 5-7) do algoritmo INSERTION SORT abaixo, usa uma busca linear para percorrer (para trás) o sub-vetor A[1 . . j - 1]. Poderia ser utilizada uma busca binária para melhorar o tempo de execução do INSERTION SORT para $\Theta(n \lg n)$?

```
INSERTION-SORT(A)
1. for j ← 2 to length[A] do
2.    key ← A[j]
3.    //Insert A[j] into the sorted sequence A[1..j - 1].
4.    i ← j - 1
5.    while i > 0 and A[i] > key do
6.          A[i + 1] ← A[i]
7.          i ← i - 1
8.    A[i + 1] ← key
```

6. Considere o seguinte algoritmo:

```
procedure DC(n)
if n ≤ 1 then
   return;
for i ← 1 to 8 do
    DC(⌊n/2⌋);
for i ← 1 to n³ do
    dummy ← 0;
```

a) Este algoritmo apresenta características de que tipo de estratégia de solução de problemas? Justifique.
b) Defina a função T(n) do tempo de execução do algoritmo.
c) Defina a ordem de crescimento assintótico da função T(n) usando a notação Θ.

7. Considere o seguinte algoritmo:

```
procedure waste(n)
for i ← 1 to n do
    for j ← 1 to i do
        write i,j,n;
if n > 0 then
    for i ← 1 to 4 do
        waste(⌊n/2⌋);
```

a) Defina a função T(n) do tempo de execução do algoritmo.
b) Defina a ordem de crescimento assintótico da função T(n) usando a notação Θ.

8. Descreva de maneira objetiva e coesa, com suas palavras, a estratégia <u>gulosa</u> para solução de problemas, e enumere suas principais características.

9. Quais os tipos de problemas aos quais os algoritmos gulosos são aplicáveis?

10. O professor Midas dirige um automóvel de Newark até Reno pela Interestadual 80. O tanque de gasolina de seu carro contém combustível suficiente para viajar *n* quilômetros e seu mapa mostra as distâncias entre os postos de gasolina na estrada. O professor deseja fazer o mínimo de paradas possível ao longo da viagem. Desenvolva um método que permita

ao professor Midas determinar em que postos de gasolina ele deve parar. Tente provar que a sua estratégia produz uma solução ótima.

11. O algoritmo de ordenação por seleção (*selection sort*) abaixo pode ser considerado um algoritmo guloso? Identifique no algoritmo os componentes elementares de uma estratégia gulosa (conjunto solução, função de seleção, etc.). Defina a função T(n) **exata** do algoritmo e sua ordem de crescimento usando a notação O.

```
procedure select(T[1..n])
for i ← 1 to n-1 do
    minj ← i;
    minx ← T[i];
    for j ← i+1 to n do
        if T[j] < minx then
            minj ← j;
           minx ← T[j];
    end;
T[minj] ← T[i];
T[i] ← minx;
end;
```

12. No problema do troco, considere moedas de 30, 24, 12, 6, 3 e 1. A estratégia gulosa sempre produzirá soluções ótimas considerando esses valores? Dê exemplos.

13. No problema do troco, considere moedas de 50, 25, 5, 1. A estratégia gulosa sempre produzirá soluções ótimas considerando esses valores? Dê exemplos.

14. No que consiste a estratégia de solução de problemas baseada na busca exaustiva ou força bruta? Por que seu uso é restrito a tamanhos de entrada pequenos?

15. Descreva de maneira objetiva e coesa, com suas palavras, a estratégia de programação dinâmica para solução de problemas, e enumere suas principais características.

16. Mencione a principal vantagem da programação dinâmica em relação à busca exaustiva?

17. Defina as duas principais características que um problema deve apresentar para que a estratégia de programação dinâmica configure-se uma boa maneira de resolvê-lo.

18. Os números de Fibonacci são definidos como:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2. \end{cases}$$

Escreva dois algoritmos: a) usando recursividade; b) usando programação dinâmica; para calcular o Fibonacci de um número n. Qual a complexidade de cada solução? Qual a mais eficiente?

19. O cálculo do coeficiente binomial é dado por:

$$\binom{n}{k} = \begin{cases} 1, & se\ k = 0\ ou\ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & se\ 0 < k < n \\ 0, & caso\ contrário. \end{cases}$$

Escreva dois algoritmos: a) usando recursividade; b) usando programação dinâmica; para calcular o coeficiente binomial dados n e k. Qual a complexidade de cada solução? Qual a mais eficiente?