**Assignment # 3**

**Dynamic Programming, Greedy Algorithms**

**Problem 1. Dynamic Programming**

**(1) Consider a modification of the rod-cutting problem in which, in addition to a price pi for each rod, each cut incurs a used cost of ci. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Design a dynamic-programming algorithm to solve this modified problem.**

**Solution:**

int modifiedRodCutting(int prices[], int costs[], int n)

{

int revenue[n + 1];

revenue[0] = 0;

for (int i = 1; i <= n; ++i)

{

int maxRevenue = 0;

for (int j = 1; j <= i; ++j)

{

maxRevenue = max(maxRevenue, prices[j - 1] - costs[j - 1] + revenue[i - j]);

}

revenue[i] = maxRevenue;

}

return revenue[n];

}

**(2) A robot, initially standing at point 0, needs to travel from point n on a straight line. In order to do that it can take steps of either size 1, size 2 or size 3. For example, if it is at point k, then it can go next to point k+1, point k+2 or point k+3. This means that there are many different ways in which it can get from point 0 to point n. For example, if n=3, there are 4 ways to get from 0 to 3: either take three steps of 1, or take a step of 2 followed by step of 1, or take a step of 1 followed by a step of 2, or take a single step of 3. Write a dynamic programming algorithm to compute the total number of ways in which the robot can get from point 0 to point n.**

**a. How many ways are there for the robot to get from 0 to 4?**

**Solution:**

Number of ways from 0 to 4 are **7**

**b. Let V[k] be defined as the number of ways for the robot to get from point 0 to point k. Write a recurrence for V[k] in terms of smaller sub-problems. Also give the base case(s).**

**Solution:**

**Recurrence Relation for V[k]:** V[k] = V[k - 1] + V[k - 2] + V[k - 3]

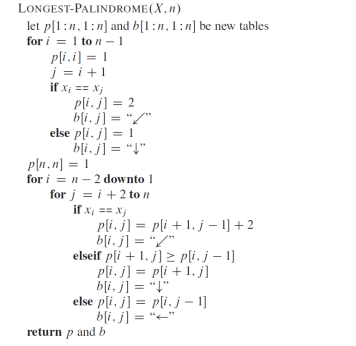
**Base Cases:** V[0] = 1, V[1] = 1, V[2] = 2

**c. Make the array V from v[0] to v[8] and fill in the values using the recurrence in (b).**

**Solution:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***i*** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **V[i]** | 1 | 1 | 2 | 4 | 7 | 13 | 24 | 44 | 81 |

**(3) A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and madam.**

**An efficient algorithm to find the longest palindrome that is a subsequence of a given input string is provided below. For example, given the input string X = character, this algorithm should return “carac”.** 

Given the input string **X = character**

**a. What is the longest palindrome subsequence.**

**Solution:**

**Input:** "character"

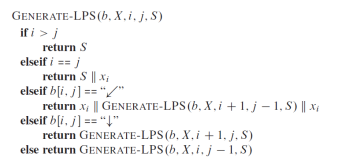
The longest palindrome subsequence for the input string "character" would be "carac".

**b. Draw the complete table for calculation of the longest palindrome subsequence of X.**

**Solution:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P** | **c** | **h** | **a** | **r** | **a** | **c** | **t** | **e** | **r** |
| **c** | 1 | 1 | 1 | 1 | 3 | 5 | 5 | 5 | 5 |
| **h** | 0 | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 |
| **a** | 0 | 0 | 1 | 1 | 3 | 3 | 3 | 3 | 3 |
| **r** | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 3 |
| **a** | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| **c** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| **t** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| **e** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **r** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**c. Print the longest palindrome subsequence using following method.**



**Solution:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **B** | **c** | **h** | **a** | **r** | **a** | **c** | **t** | **e** | **r** |
| **c** |  |  |  |  |  |  |  |  |  |
| **h** |  |  |  |  |  |  |  |  |  |
| **a** |  |  |  |  |  |  |  |  |  |
| **r** |  |  |  |  |  |  |  |  |  |
| **a** |  |  |  |  |  |  |  |  |  |
| **c** |  |  |  |  |  |  |  |  |  |
| **t** |  |  |  |  |  |  |  |  |  |
| **e** |  |  |  |  |  |  |  |  |  |
| **r** |  |  |  |  |  |  |  |  |  |

**(4) You are given a set of substrings of DNA sequence. A DNA sequence is defined as a string of 4 alphabets (A, G, C, T) as follows:**

**GCAACGTTAGA….**

**A substring of DNA sequence is a consecutive string. For example, ACG is a substring of above DNA sequence but GGA is not substring of this sequence. Given a new DNA sequence Sn with length n and a set of substrings K of another DNA sequence, find out if the new DNA sequence Sn can be completely divided into substrings of the given set K.**

**For example:**

**Let Sn = GCAACGTTAGA and K = {AGA, GT, GC, AACG, TT}**

**Sn can be divided into following substrings from K**

**GC, AACG, TT, AGA**

**Second Example:**

**Let Sn = GCAGCCTGTACT and K = {AG, GT, AACG, CC}**

**Sn cannot be divided into substrings from K**

**Given a DNA sequence Sn of length n and a set K of substrings, describe an efficient algorithm that will detect whether or not the input DNA sequence Sn can be split into substrings of set K. Your algorithm should also print the substrings. Analyze the time complexity of your algorithm.**

**You can assume that there is a function that given a substring, checks in O(1) time if the substring belongs to input set K of substrings.**

**Solution:**

**Problem 2. Greedy Algorithms**

**(1) Given a knapsack of capacity 50, what is the maximal value obtainable with three items of $60, $100 and $120 with weights 10, 20 and 30 respectively?**

**a. Show how the exhaustive algorithm will obtain this value. Recall that the exhaustive algorithm checks all possible candidate solutions and then picks the one which maximizes the accumulated value.**

**Solution:**

For this problem with three items:

**Knapsack Capacity = 50**

|  |  |  |
| --- | --- | --- |
| **Item** | **Value ($)** | **Weight** |
| 1 | 60 | 10 |
| 2 | 100 | 20 |
| 3 | 120 | 30 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Item1** | **Item2** | **Item3** | **Total Value** | **Total Weight** |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 60 | 10 |
| 0 | 1 | 0 | 100 | 20 |
| 1 | 1 | 0 | 160 | 30 |
| 0 | 0 | 1 | 120 | 30 |
| 1 | 0 | 1 | 180 | 40 |
| **0** | **1** | **1** | **220** | **50** |
| 1 | 1 | 1 | 280 | 60 |

**Optimal Solution**: Items 2 and 3

**Maximal Value**: $220

**b. Write pseudo-code for the Fractional Knapsack algorithm. The running time of the algorithm should be O(nlgn) for a problem over n items. The inputs to the algorithm are: arrays p and w of sizes n each containing the price and weights of the n items; a number W which is the capacity of the knapsack; and the number n.**

**Solution:**

FractionalKnapsack(p[], w[], W, n):

Calculate value-to-weight ratios for each item: r[i] = p[i] / w[i]

Sort items based on the ratio r[] in decreasing order

max\_value = 0

for i = 0 to n-1:

if W == 0:

return max\_value

a = min(w[i], W) // Take the whole item or fraction of it

max\_value = max\_value + a \* (p[i] / w[i])

W = W - a

return max\_value

**(2) We know that the activity selection algorithm works by picking non-overlapping activities in the increasing order of their finish times. Which of the following two strategies will also produce an optimal solution? Explain your answer:**

**a. Picking non-overlapping activities in the decreasing order finish times.**

**Solution:**

Selecting activities in decreasing order of finish times **doesn't guarantee an optimal solution**. While it's intuitive to select activities with later finish times first, this strategy may lead to suboptimal results. It's possible that by choosing a later finishing activity, it prevents other smaller activities from being scheduled.

**b. Picking non-overlapping activities in the decreasing order of start times.**

**Solution:**

Choosing activities based on decreasing start times also **doesn't ensure an optimal solution**. This strategy might select longer activities with earlier start times, leading to fewer options available for scheduling other activities later. It might overlook shorter activities that could fit in between other activities and maximize the total number of scheduled activities.

**(3) You are given a set of activities to schedule among a large number of n lecture halls, where any activity can take place in any lecture hall. You wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.**

**Solution:**

void scheduleActivities(int finish[], int start[], int activities[], int noOfActivities, int noOfHalls)

{

sort(activities);

int\* lectureHalls = new int[noOfHalls] {0};

for (int i = 0; i < noOfActivities; ++i)

{

int hallIndex = 0;

while (hallIndex < noOfHalls)

{

if (lectureHalls[hallIndex] > 0 && finish[activities[lectureHalls[hallIndex] - 1]] > start[activities[i]])

{

hallIndex++;

}

else

{

break;

}

}

if (hallIndex < noOfHalls)

{

lectureHalls[hallIndex] = i + 1;

}

}

delete[] lectureHalls;

}

**(4) A person is traveling by air from country 1 to country k. He can take different connecting flights on his way. For each connecting flight the air fare ai,j from country i to j is given. The air fares are arbitrary. For example, it is possible that a1,3 = 11 and a1,4 = 6. He wants to minimize the total air fare even if he has to take many connecting flights. Describe an efficient algorithm for solving this problem and analyze its time complexity.**

**Solution:**