**Q1**: Consider the following knapsack problem:

|  |  |  |
| --- | --- | --- |
| Item | Value | Weight |
| 1 | 1 | 1 |
| 2 | 3 | 2 |
| 3 | 4 | 1 |
| 4 | 2 | 2 |
| 5 | 3 | 3 |
| 6 | 6 | 2 |

The capacity of the knapsack is 9. Use the DP algorithm to solve it.

**Answer:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| {} | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| {1} | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| {1,2} | 0 | 1 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| {1,2,3} | 0 | 4 | 5 | 7 | 8 | 8 | 8 | 8 | 8 | 8 |
| {1,2,3,4} | 0 | 4 | 5 | 7 | 8 | 9 | 10 | 10 | 10 | 10 |
| {1,2,3,4,5} | 0 | 4 | 5 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| {1,2,3,4,5,6} | 0 | 4 | 6 | 10 | 11 | 13 | 14 | 15 | 16 | 17 |

Backtracking {6→5→3→2→1}

Opt = {1, 2, 3, 5, 6} max-Value = 6+3+4+3+1 = 17

**Q2:** Given 8 jobs with the following (v, s, f)-values (v=value, s= start time, and f= finish times): a=(4,0,5), b=(5,6,9), c=(6,5,8), d=(5,3,6), e=(4,5,7), f=(12,8,11), g=(2,7,10), h=(7,9,13).

Use a DP algorithm to find a set of mutually compatible jobs with the maximal total value.

**Answer:**

**Step 1**

Input: the information of the jobs.

a=(4,0,5), b=(5,6,9), c=(6,5,8), d=(5,3,6), e=(4,5,7), f=(12,8,11), g=(2,7,10), h=(7,9,13).

**Step 2**

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

a=(4,0,5), d=(5,3,6), e=(4,5,7), c=(6,5,8), b=(5,6,9), g=(2,7,10), f=(12,8,11), h=(7,9,13).

**Step 3**

Compute p[1], p[2], …, p[n], where the value of p[j] is the largest index i < j such that job i is compatible with j.

p[1]=0, => null

p[2]=0, => null

p[3]=1, => a

p[4]=1, => a

p[5]=2, => d

p[6]=3, => e

p[7]=4, => c

p[8]=5, => b

**Step 4 DP algorithm**

Function {

v[0] = 0

for j = 1 to n

v[j] = max(value[j] + v[p[j]], v[j-1])

}

v[1]=4, => max(value[1] + v[0], v[0])

v[2]=5, => max(value[2] + v[0], v[1])

v[3]=8, => max(value[3] + v[1], v[2])

v[4]=10, => max(value[4] + v[1], v[3])

v[5]=10, => max(value[5] + v[2], v[4])

v[6]=10, => max(value[6] + v[3], v[5])

v[7]=22, => max(value[7] + v[4], v[6])

v[8]=22, => max(value[8] + v[5], v[7])

**Step 5**

Output v[n]

the maximal total value = v[8]=22

Q3 Lisa will graduate next year, and she wants to find a good job, and build a career path. An ideal career path to her is that the salaries are never decreasing, and ideally, are always multiplying. One day, Lisa met a fortune teller, and was told a sequence of jobs to choose. Lisa has not taken CS4335, and she asked your help to design a method to choose the jobs. Again, we have formulated the problem formally.

A is a sequence of positive integers (represents the salaries): A = (a1, a2, ... an). A **multiplication subsequence of** A is a subsequence S=() satisfies that (1) S is obtained by remove some entries of **A** sequentially; that is, i1<i2<i3<...<ik, is in **A**; and (2) is a multiple of , is a multiple of , …; that is, if we let divide by , ,the remainder is zero.

For example, if A =(1, 2, 3, 3, 4, 5, 6, 7, 8, 15), then (1, 2, 4, 8), (1, 3, 3, 6), and (1, 3, 15) are multiplication subsequences of A.

1. Given A as a sequence of positive integers, design an algorithm to identify a longest multiplication subsequence.
2. Define the weight of a sequence as the sum of the elements in the sequence. Design an algorithm to identify a maximum weighted multiplication subsequence.

Answer:

Q(a)

Let dp[i] stores the the length of a longest multiplication subsequence of a1, …, ai. Without loss of the generality, we set a0=1.

Then the recurrence relations can be formulated as,

Clearly, we can set the trace array T as,

Then we can transform the above formulas into pseudo code:

**for** i ←1 to n:

dp[i] ←0

T[i] ←0

**for** i ←1 to n:

**for** j←1 to i-1:

**if** aimodaj=0 **and** dp[i]<dp[j]+1

dp[i]← dp[j]+1

// we need another pass to finding the longest subsequence

longest ← -∞

index ← -1

**for** i ←1 to n:

**if** dp[i] >longest:

longest ←dp[i]

**index** ← i

//now we print the longest subsequence,

//and we use the recursive function

Trace(i)

**if** i>0

Trace(T[i])

**Print** i

Initial call Trace(**index**)

The running time O(n2); that is, we need O(n2) in the worst case to build the dynamic array, and linear time to trace the solution.

Q(b)

Let dp[i] stores the sum of a maximum weighted multiplication subsequence of a1, …, ai. Without loss of the generality, we set a0=1.

Then the recurrence relations of Q(b) can be formulated as,

Clearly, the trace array T as,

Then we can transform the above formulas into pseudo code:

**for** i ←1 to n:

dp[i] ←0

T[i] ←0

**for** i ←1 to n:

**for** j←1 to i-1:

**if** aimodaj=0 **and** dp[i]<dp[j]+a[i]

dp[i]← dp[j]+a[i]

// we need another pass to finding the maximum weighted multiplication subsequence

longest ← -∞

index ← -1

**for** i ←1 to n:

**if** dp[i] >longest:

longest ←dp[i]

**index** ← i

//now we print the maximum weighted multiplication subsequence,

//and we use the recursive function

Trace(i)

**if** i>0

Trace(T[i])

**Print** i

Initial call Trace(**index**)