R-5.1 Let S = {a, b, c, d, e, f, g} be a collection of objects with benefit-weight values as follows: a:(12,4), b:(10,6), c:(8,5), d:(11,7), e:(14,3), f:(7,1), g:(9,6). What is an optimal solution to the fractional knapsack problem for S assuming we have a knapsack that can hold objects with total weight 15? Show your work.

Solution

S = {a, b, c, d, e, f, g}

a: (12,4), b:(10,6), c:(8,5), d:(11,7), e:(14,3), f:(7,1), g:(9,6)

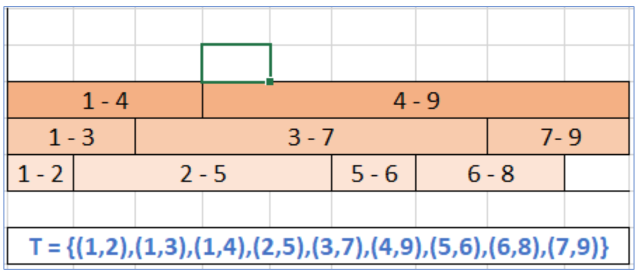
The benefit per weight: **a → 3**, b → 5/3, c → 1.6, d → 11/7, **e → 14/3**, **f → 7**, g → 1.5

Greedy choice: F (7, 1) → A (12, 4) → E (14, 3) → B (10, 6) , total weight is 14 < 15

⇨ **F (7, 1) → A (12, 4) → E (14, 3) → B (10, 6) → C (1.6, 1)**

R-5.3 Suppose we are given a set of tasks specified by pairs of the start times and finish times as T = {(1,2), (1,3), (1,4), (2,5), (3,7), (4,9), (5,6), (6,8), (7,9)}. Solve the task scheduling problem for this set of tasks.

Solution



R-5-11 Solve Exercise R-5.1 above for the 0-1 Knapsack Problem.

Solution

k/w		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	(12, 4)	0	0	0	0	12	**12**	12	12	12	12	12	12	12	12	12	12
2	(10, 6)	0	0	0	0	12	**12**	12	12	12	12	22	22	22	22	22	22
3	(8, 5)	0	0	0	0	12	**12**	12	12	12	20	22	22	22	22	22	30
4	(11, 7)	0	0	0	0	12	**12**	12	12	12	12	12	23	23	23	23	30
5	(14, 3)	0	0	0	14	14	14	14	26	**26**	26	26	26	26	26	37	37
6	(7, 1)	0	7	7	14	21	21	21	21	**33**	**33**	33	33	33	33	33	44
7	(9, 6)	0	7	7	14	21	21	21	21	33	33	33	33	33	33	42	**44**

{(12, 4), (14, 3), (7, 1), (9, 6)}

R-5-12 Sally is hosting an Internet auction to <u>sell n widgets</u>. She receives <u>m bids</u>, each of the form "I want ki widgets for di dollars," for i = 1, 2, …, m. Characterize her optimization problem as a knapsack problem. Under what conditions is this a 0-1 versus fractional problem?

Solution:

$bid_i = (d_i, k_i)$, the weight for each bid is d_i/k_i

a knapsack problem accept bid have < k_i widget

0-1 versus fractional problem: accept if we don't accept any bid < k_i

After Lesson 11b on memorization, do the following and submit next week:
A. Based only on the characterizing equations (B[k,w]), give a recursive pseudo code algorithm for <u>the 0-1 knapsack problem</u> (do this from the equations and without looking at my solution in the notes), <u>then memorize it so it is efficient</u>. Compare your algorithm to the two given in the lecture notes (iterative dynamic programming version and recursive non-memorized algorithm) in terms of time and space complexity.

Solution:

```
Algorithm 01KnapsackRecursive(S, W)
        return 01KnapsackRecursiveHelper(S, W, S.size() - 1)

Algorithm 01KnapsackRecursiveHelper(S, W, k)
        if S is null v W = 0 then
                return 0
        if k = 0 then
                return 0
        (b_k, w_k) := S.elemAtRank(k)
        if w_k > W then
                return 01KnapsackRecursiveHelper(S, W, k - 1)
        else
                int v1 = 01KnapsackRecursiveHelper(S, W, k - 1)
                int v2 = 01KnapsackRecursiveHelper(S, W - w_k, k - 1) + b_k
                return Max(v1, v2)

Algorithm 01KnapsackMemorizedRecursive(S, W)
        int[][] arr = new int[k + 1][W + 1]
        for int i = 0 to k then
                for int j = 0 to W then
                        arr[i][j] = -1
        return 01KnapsackRecursiveHelper(S, W, S.size() – 1, arr)

Algorithm 01KnapsackMemorizedRecursiveHelper(S, W, k, arr)
        if S is null v W = 0 then
                return 0
        if k = 0 then
                return 0
        if arr[k][W] <> -1
                return arr[k][W]
        (b_k, w_k) := S.elemAtRank(k)
        if w_k > W then
                int v = 01KnapsackRecursiveHelper(S, W, k – 1, arr)
                arr[k][W] = v
                return v
```

```
else
        int v1 = 01KnapsackRecursiveHelper(S, W, k – 1, arr)
        int v2 = 01KnapsackRecursiveHelper(S, W - w_k, k – 1, arr) + b_k
        arr[k][W] = Max(v1, v2)
        return arr[k][W]
```

C-5.9 How can we modify the dynamic programming algorithm from simply computing the best benefit value for the 0-1 knapsack problem (like A above) to computing the assignment (subset) that gives the maximum benefit? Design a pseudo code algorithm to do the trace back through the 2-dimensional array as we described in the lecture.

Solution:

TBD

B. Suppose we have a set of objects that have different sizes **s1, s2, …, sn**, and we have some positive upper limit **L**. Design an efficient pseudo code algorithm to determine the subset of objects that produces the largest sum of sizes that is no greater than L. Hint: dynamic programming similar to 0-1 knapsack problem, except only size/weight and no benefit.

Solution

```
Algorithm 01KnapsackMemorizedRecursive(S, L)
    int[][] arr = new int[k + 1][L + 1]
    LO := new List
    for int i = 0 to k then
        for int j = 0 to L then
            arr[i][j] = -1
    return 01KnapsackRecursiveHelper(S, L, S.size() – 1, arr)

Algorithm 01KnapsackMemorizedRecursiveHelper(S, L, k, arr, LO)
    if S is null v L = 0 then
        return 0
    if k = 0 then
        return 0
    if arr[k][L] <> -1
        return arr[k][L]
    obj_k := S.elemAtRank(k)
    if size(obj_k) > L then
        (size, LO) = 01KnapsackRecursiveHelper(S, L, k – 1, arr, LO)
        arr[k][LO] = size
        return size
    else
        (size1, LO) = 01KnapsackRecursiveHelper(S, L, k – 1, arr, LO)
        (size2, LO) = 01KnapsackRecursiveHelper(S, L - size(obj_k), k – 1, arr, LO) + size(obj_k)
        arr[k][L] = Max(size1, size2)
        if size2 > size1 then
        LO.insertFirst(obj_k)
        return (arr[k][W], LO)
```