let. S → {a, b, c, d, e, f, g}

1) a) 3 b) $\frac{5}{3}$, c) $\frac{8}{5}$ d) $\frac{11}{7}$

e) $\frac{14}{3}$ f) 7 g) $\frac{3}{2}$

Sorting in descending order:

$f=7$, $e=\frac{14}{3}$, $a=3$, $b=\frac{5}{3}$, $c=\frac{8}{5}$, $d=\frac{11}{7}$, $g=\frac{3}{2}$

Posibility:

weight = 18

take (f) => $18-7=11$

take (e) => $11-\frac{14}{3}=\frac{19}{3}$

take (a) => $\frac{19}{3}-3=\frac{10}{3}$

take (b) => $\frac{10}{3}-\frac{5}{3}=\frac{5}{3}$

take (c) => $5/3-\frac{8}{5}=\frac{1}{15}$

take (d) -> ~~$\frac{1}{15}-\frac{11}{7}=\frac{-}{105}$~~ ~~$\frac{1}{15}-\left(\frac{15}{11}\cdot\frac{7}{7}\cdot\frac{1}{25}\right)\frac{1}{15}-\frac{11}{1275}\left(\frac{11}{7}\right.$~~

take (g) => $0.06-\frac{7}{165}(d)=0$

Total: $7+14+12+10+8+\frac{7}{165}(11)$

$=51.47$

R. 12.5

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

$a = (12, 4)$ $b = (10, 6)$ $c = (8, 5)$ $d = (11, 7)$

$e = (14, 3)$ $f = (7, 1)$ $g = (9, 6)$

val	weight	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
7	1	0	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
14	3	0	7	7	14	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
12	4	0	7	7	14	21	21	21	26	33	33	33	33	33	33	33	33	33	33	33
8	5	0	7	7	14	21	21	21	26	33	33	33	33	34	41	41	41	41	41	41
10	6	0	7	7	14	21	21	21	26	33	33	33	33	34	41	43	43	43	43	44
9	6	0	7	7	14	21	21	21	26	33	33	33	33	34	41	43	43	43	43	44
11	7	0	7	7	14	21	21	21	26	33	33	33	33	34	41	43	44	44	44	44

$44 - 11 = 33 \Rightarrow$ include (d)

$33 - 12 = 21 \Rightarrow$ include (d) + (a),

$21 - 14 = 7 \Rightarrow$ include (d) + (a) + (e)

$7 - 7 = 0 \Rightarrow$ include (d) + (a) + (e) + (f)

Optimal : a + e + d + f. total = 44

$4 + 3 + 7 + 1 = 15$

```
Algorithm 0-1-Heber( S, w, k, B )
                              ↓ Hashtable
  if B.findValue((w,k)) == NO-SUCH-key) then
    if k=0 || W=0
          return

    item = S.elemAtRank (k-1)
    bk = item.benefit()
    wk = weight(item)
    if wk > w then
        ben = 0-1-Helper(S, w, k-1)
        B.insertItem((w,k),ben)
    else
        ben = Max( 01-Helper(s,w,k-1), 01-Helper(s,w-wk,k-1)+bk)
        B.insertItem( (w,k ),ben)

  return B.findValue (w,k)
```

1) B exam Question

R.5-12.

where:

n = weight

m = number of bids - items knapsack

k_j = weight of ith item

d_i = value of ith item

Fractional Knapsack

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack

Here items are breakable ie we can break items for maximizing the total value of knapsack.

it usues greedy algorithm to optimize profit

0-1 Knap sack problem

In the 0-1 kanasack problem, we are not allowed to break items, we either take the whole item or don't take it. uses dynamic algorithm to optimize profit.

11 b).

```
Algorithm Kapsack-Reursivo (S, w)
 return 0-I-Helper(S, w, S.size)
Algorithm 0-I-Helper(S, w, k)
    if k=0 || w=0 then
        return 0
    item := S.elementAtRank(k-1)
    bk := item.benefit()
    wk := item.weight()
    if wk > w then
        return 0-I-Helper(S, w, k-1)
    else
        return max(0-I-Helper(S, w, k-1), 0-I-Helper(S, w-wk, k-1) + bk)
Algorithm Kapsack-Memorized (S, w)
   return 0-I-Helper(S, w, S.size, Hashtable)
Algorithm 0-I-Helper(S, w, k, H)
   if H.findValue((w,k)) == NO_SUCH_KEY then
       if k=0 || w=0 then return 0
       item := S.elementAtRank(k-1)
       bk := item.benefit()
       wk := item.weight()
       if wk > w then
           H.insertItem((w,k), 0-I-Helper(S, w, k-1, H))
       else ben := max(0-I-Helper(S, w, k-1, H), 0-I-Helper(S, wk-w, k-1, H) + bk)
           H.insertItem((w,k), ben)
   return H.findValue((w,k))
```

~~6.5.9~~)

B.

```
Algorithm 01Knapsack Memorized Recursived (S, L)
    int [][] = new Arr with K+1 an L+1
    LO := new List
    for i:=0 to k then
        for j=0 to L then
            arr[i][j] = -1
    return helper (S, L, S.size()-1, arr)

Algorithm Helper( S, L, k, arr, LO)
    if S == null || 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) = Helper (S, L, k-1, arr, LO)
            arr[k, LO] = size
            return size
        else
            (size1, LO ) = Helper ( S, L, k-1, arr, LO)
            (size2, LO) = Helper ( 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)
```