Amit Borase

Question 1

If there are any items with the non-positive profit values, then ignore them completely.

This problem can be solved using Dynamic Programming technique. Let M(i,W') denote the maximum value of the items picked from items 1 to i with a total weight constraint of W'. With this, the solution of the problem will be to find M(n,W), where n is the total number of items and W is the weight constraint of the sack.

Algorithm:

1. Ignore all the items that have negative values.
2. Initialize a 2-D solution space M of dimensions n+1 (of rows) and W+1 (of columns).
3. Set all elements in first row (No items to select from) and first column (weight constraint of zero) to zero. (base case)
4. Start from second row (item 1) and and second column (Weight constraint of 1), i.e. start from M[1][1] and compute it using below set of rules.
   (a) For M[i][j], if weight of ith item is greater than j, then M[i][j] = M[i-1][j]. Also Update the backtracking pointers to indicate non-selection of item i.
   (b) if weight of ith item is smaller of equal to j, then there are below 2 possibility.
      - profit from non-selection of item i is more than the profit from selection of the same. In that case M[i][j] = M[i-1][j], update tracking pointers to indicate the same.
      - profit from selection of item i is more than the profit from non-selection of the same. In that case M[i][j] = M[i-1][j-w(i)] + v(i), update tracking pointers to indicate the selection of item i.
5. Continue this till we reach the element M[n][W] of the solution space.
6. M[n][W] is the maximum pofit that can be made
7. Use tracking pointers to find the selected item by back-tracking.

Proof of Correctness:

Let M(i,W') denote the maximum value of the items picked from items 1 to i with a total weight constraint of W'. With this the solution of the problem is to find M(n,W), where n is the total number of items and W is the weight constraint of the sack.

Base-case:

case1: If the weight constraints is zero, then the no matter how many items we've we'll not be able to select any item; hence the maximum profit in that case is zero. i.e. M[i][0] = 0 for all i's.

case2: If there are no available items that can be selected, then no matter how big the weight constraint is, we'll not be able to make any profit. hence the maximum profit in that case is zero. i.e. M[0][i] = 0 for all i's.

Induction:

Optimal Substructure For any given 1<=i<=n and 1<=j<=W, M[i][j] denotes the maximum profit that could be made if we've items only till item i and weight constraint of j. To maximize this profit, we have to select based on below rules.

In any case there will be below two possibilites for this step:

- Either the weight of the item i is greater than the weight constaraint of j. In that case, we can never select item i. hence the max profit we can make will be the profit that we can make using by using items till item i-1.

- Or if the weight of the item i is less than equal to weight constraint of j, then we may or maynot select the item i depending on the profits associated iwth each case. We'll select item i, if the profit from item i plus the profit from item i with constraint of j-w(i) is more compared to the profit from filling sack of weight constarint j with items till i-1. Otherwise we'll not slect item i.

Hence the optimal substructure becomes.

If w(i) > j, then M[i][j] = M[i-1][j] else M[i][j] = maxM[i-1][j], v(i)+M[i-1][j-w(i)]

Pseudo-code:

Listing 1: Question 1

```
INPUT: S = {(w(i), v(i)), where i is 1 to n} and Weight constraint W
OUTPUT: M(n, W) (maximum profit that could be made) and list of selected item.

KNAPSACK_MAIN(n, W)
    int M[n+1][W+1] = {-1}
    # Base cases
    Set M[0][i] = 0, for all i in 0 to W
    Set M[i][0] = 0, for all i in 0 to n

    list item= {}  # list of items ot be selected   # solution space
    track[n+1][W+1] = {(-1.-1)} # Matrix of tuples to track selection

    for i in (1 to n)
        for j in (1 to W)
            if (w(i) > j)
                M[i][j] = M[i-1][j]
                trac[i][j] = (i-1, j)
            else
                if (M[i-1][j] > M[i-1][j-w(i)]+v(i)))
                    M[i][j] = M[i-1][j]
                    trac[i][j] = (i-1, j)
                else
                    M[i][j] = M[i-1][j-w(i)] + v(i)
                    trac[i][j] = (i-1, j-w(i))

    (x,y) = trac[n][W]
    (x',y') = (n, W)
    while (y>0)
```

```
        if y != y'
            item.append(x)
        (x',y') = (x,y)
        (x,y) = trac[x][y]

    return (M[n][W], item)
```

Running-time Analysis:

At every calculation of M[i][j], we only compare and select pre-calculated set of values from solution space, hence each computation of M[i][j] takes constant O(1) time.

We compute total n*W M[i][j]'s, hence that takes O(n*W) time.

The backtracking operation tries to reach upper most left corner of the solution space from the lower–most right corner, hence in worst-case it takes O(n+W) operation.

Total run-time is O(n*W) + O(n+W) = O(n*W) time.

Question 2

We solve this by using dynamic programming paradigm. We'll maintain a 2-D array, whose element M[i][j] indicates the minimum of coins of value i or lesser required to make chnage for j amount.

We'll build the solution space, by calculating minimum of each possible solution space.

Algorithm:

1. Ignore all the items that have negative values.
2. Initialize a 2-D solution space M of dimensions n+1 (of rows) and C+1 (of columns).
3. Set all elements in first row (No coins to select from) and first column (amount to be changed for is zero) to zero. (base case)
4. Set all elements in the second row (only coin of value v1=1 to select from) to the value equal to respective column number (base case)
5. set all the element in the second column (amount to be chnaged for of 1) to 1, since minimally, we can use coin v1=1 to chnage for amount of 1. (base case)
6. Start from third row (v2) and third column (amount to be changed for of 1), i.e. start from M[2][1] and compute it using below set of rules.
   (a) For M[i][j], if value of ith coin is greater than amount j, then we cannot selct ith coin hence M[i][j] = M[i-1][j]. Also Update the backtracking pointers to indicate non-selection of coin i.
   (b) if value of ith coin is smaller or equal to amount j, then there are below 2 possibility.
      - of coins with non-selection of coin i is lesser than the of coins after selecting ith coin. In that case M[i][j] = M[i-1][j], update tracking pointers to indicate the same.
      - of coins with selection of item i are lesser than the of coins from non-selection of the same. In that case M[i][j] = M[i][j-vi] + 1, update tracking pointers to indicate the selection of coin i.
7. Continue this till we reach the element M[n][C] of the solution space.

8. M[n][C] is the minimum of coins needed to make change for C.

9. Use tracking pointers to find the of coins selected of each type by back-tracking.

Proof of Correctness

Base-cases:

case1: If the amount to be changed for is zero, then no matter how many different valued coins we've we don't have to select any coin; hence the minimum coins required in that case is zero. i.e. M[i][0] = 0 for all i's in (0, n)

case2: If there are no available coins to select as a chnage, then no matter how small the amount ot be changed for is, we'll not be able to make any chnage. hence the minimum coins required will remain zero. i.e. M[0][i] = 0 for all i's in (0, C)

case3: If v1=1 is the only coin available for the chnage, then no matter what the amount to be chnaged for is we'll have to chnage it using coins of value 1, hence required number of coins will be same as the amount to be chnaged for. i.e. M[1][i] = i for all i's in (1, C)

case4: If amount tp be chnaged for is 1, then no matter how many coins we've we can minimally chanage this amount using single coin of value v1=1, hence required number of coins will be 1. i.e. M[i][1] = 1 for all i's in (1, n)

Induction:

Optimal Substructure

For any given 1<=i<=n and 1<=j<=c, M[i][j] denotes the minimum number of coins of value lesser than that of coin vi, that can be used to chnage for the amount j. To minimize the of coins to use, we have to select coins based on below set of rules.

For any i, there will be below two possibilites for this step:

- Either the value of the coin i is greater than the amount of j. In that case, we can never select coin i to chnage for j. hence the minimum of coins required will be the minimum of coins required of value lesser than coin vi.

- Or if the value of the coin i is less than or equal to the anount j to be chnaged for, then we may or maynot select the coin i depending on the of coins associated with each case. We'll select coin i, if the number of coins after using single coin of value vi, is lesser than the of coins required when coin i is not selected. Otherwise we'll select only the coins with value lesser than that of coin i.

- Note that we can possibly select multiple ith coins, but here in each step we select only single such coin of value vi. The sub-structure handles the additional selection of the same coin of value vi.

Hence the optimal substructure becomes.

If v(i) > j, then M[i][j] = M[i-1][j] else M[i][j] = minM[i-1][j], 1+M[i][j-v(i)]

Pseudo-code

Listing 2: Question 2

```
INPUT: S = {c(i), where i is 1 to n} and C is the amount of money to be chnaged ↩
    for.
OUTPUT: M(n, C), minimum number of coins required to change for amount C, and an ↩
    array coin giving the numbers of coins selected of each coin_value.

MIN_CHANGE_MAIN(n, C)
    int M[n+1][C+1] = {-1} # solution space

    # Base cases
    Set M[0][i] = 0, for all i in 0 to C
    Set M[i][0] = 0, for all i in 0 to n
    Set M[1][i] = i, for all i in 1 to C
    set M[i][1] = 1, for all i in 1 to n

    int coin[]= {0}  # of coins selected
    track[n+1][W+1] = {(-1.-1)} # Matrix of tuples to track selection

    for i in (1 to n)
        for j in (1 to C)
            if (v(i) > j)
                M[i][j] = M[i-1][j]
                trac[i][j] = (i-1, j)
            else
                if (M[i-1][j] < M[i][j-v(i)] + 1))
                    M[i][j] = M[i-1][j]
                    trac[i][j] = (i-1, j)
                else
                    M[i][j] = M[i][j-v(i)] + 1
                    trac[i][j] = (i, j-v(i))

    (x,y) = trac[n][C]
    (x',y') = (n, C)
    while (y>0)
        if y != y'
            coin[x]++
        (x',y') = (x,y)
        (x,y) = trac[x][y]

    return (M[n][C], coin[])
```

Time complexity analysis

At every calculation of M[i][j], we only compare and select pre-calculated set of values from solution space, hence each computation of M[i][j] takes constant O(1) time.

We compute total n*C M[i][j]'s, hence that takes O(n*C) time.

The backtracking operation tries to reach upper most left corner of the solution space from the lower–most right corner, hence in worst-case it takes O(n+C) operation.

Total run-time is O(n*C) + O(n+C) = O(n*C) time.

Question 3

Algorithm

1. We run a Modified BFS on given graph to build the children-list for each of the vertices in V.

2. Now compute the vertex covers with and without root in the cover. These can be done using below recursive method.

(a) Given a node v, Recursively compute both the vertex covers (M[c][o] and M[c][1]) of its children nodes (including and excluding children node). Do this for all children of v. Also compute the cover-0 and cover-1 (vertex covers for both case, including v and excluding v)

(b) Now Compute M1, sum of vertex covers of all children nodes including children (M[c][1]), also compute cover-0, sum of 1-covers of all children (including children).

(c) Similarly Compute M2, sum of vertex covers of all children excluding children (M[c][0]). also compute cover-1, sum of 0-covers of all children (excluding children).

(d) the M1 will be the size of vertex cover of the node v excluidng v (M[v][0]) and cover-0 will the respective vertex cover set of v.

(e) For vertex cover of node v including v (M[v][0]), there are two choices. Either the M1 and M2 can be selected based on whose cardinality is minimum. Add 1 for including the node v. This will be the size of minimum vertex cover of node v including node v. Based on the what gets selected out of M1 and M2, choose corresponding vertex cover set and add v to it and that will the minimum vertex cover set of v.

3. Return the minimum of the two computed vertex covers of the the root, alongwith the corresponding vertex cover set.

Proof of correctness

Base-Case:

case 1: For all the leaf nodes of the graph, the size of the vertex cover (excluding themselves) of their subtree is zero, because there is no subtree under the leaf nodes.

case2: For all leaf nodes of the graph, the size of vertex cover (inclduing themseleves) for the subtree under them is only 1, since there si no subtree under the leaf-node hence it has only leaf-node in the vertex cover, hence the cardinality is 1.

Induction Step:

For any subtree rooted at node v, the minimum vertex cover of it including and excluding v can be denoted by M[v][1] and M[v][0] respectively. These can be computed using below set of optimal substructure rules.

case1: When a node v has to be excluded from the vertex cover, then the only choice is to add all its childrens into the vertex. hence the vertex cover of v excluding v can be obtained by summing up vertex covers of its children (including childrens).

case2: When a node v has to be included into the vertex cover, then there are two choices for the inclusion of the childrens into the vertex cover, We can either exclude the childrens since node v is included or we can include all the children even though node v is in cover. The later can be done in case the summation of vertex covers of childrens includinf childrens is actually minimal than the summmation of vertex covers excluding childrens, then the former can be choosen to maintain minimality of the vertex cover.

Hence the optimal substructure becomes:

M[v][0] = summation(M[c][1], for all c in child(v))

M[v][1] = 1 + min(summation(M[c][1], for all c in child(v)), summation(M[c][0], for all c in child(v)))

Pseudo-code

Listing 3: Question 3

```
INPUT: S = {c(i), where i is 1 to n} and C is the amount of money to be changed ↩
    for.
OUTPUT: M(n, C), minimum number of coins required to change for amount C.

BFS_CHILDREN_FIND(G(V,E), child[], S, levels[])
    # Run modified BFS on given tree G that adds a node to
    # its parents child list whenever it is discovered.
    BFS_MOD(G, child[], S)

MIN_COVER(v)
    int M1 = 0
    int M2 = 0
    cover_0[|child(v)|] = {}
    cover_1[|child[v]|] = {}
    cov_0 = {}
    cov_1 = {}

    for each c in child[v]
        (M[c][0], M[c][1], cover_0[c], cover_1[1]) = MIN_COVER(c)

    for each c in child[v]
        M1 = M1 + M[c][1]
        M2 = M2 + M[c][0]
        cov_0 = cov_0 + cover_1[c]
        cov_1 = cov_1 + cover_0[c]

    if (M1<M2)
        cover = cov_0 + {v}
    else
        cover = cov_1 + {v}

    return (M1, min(M1, M2) + 1, cov_0, cover)

MIN_COVER_MAIN(G)
    # initialize solution space
    int M[V][2] = {0}
    cover_0 = {}
    cover_1 = {}

    list child[V] = {}

    BFS_CHILD_LIST_BUILD(G, child[])

    (M[r][0], M[r][1], cover_0, cover_1) = MIN_COVER(r)

    if (M[r][0] < M[r][1])
        cover = cover_0
    else
        cover = cover_1

    return (min(M[r][0], M[r][1]), cover)
```

Time complexity analysis

The above algorithm computes both the vertex covers simultaneously in each recursive calls , hence no vertex is recursively called twice. in the worst case there will be recursive calls of depth n. As can be seen aboev, every recursive call take constant time as it only does summations and comparisons.

Hence the running time of this algorithm is O(n)

Question 4

Whenever a machine does a horizontal or vertical cut on the cloth, it divides the cloth into two parts either horizontally or vertically. max Profit of the complete cloth is max of the profit of the cloth without cut or sum of the profits of both the cuts.

Since there are X-1 and Y-1 cuts possible on cloth of size X and Y, we can maintain a 2-D solution space to track the maximum profit possible from the cloth of the size X and Y, the size of the solution space will be X cross Y, where (x, y) is the original size of the given cloth.

Algorithm

1. Initilize a memoization matrix of size X*Y with -1
2. Intilize a solution space matrix of size X*Y with 0
3. Set up profit values to maximum possible of all the elements that match in dimensions with any of the product. (base case)
4. Compute the element M[X][Y] using below set of recursive operations
   (a) if result M[x][y] is already memoized then return it.
   (b) if it is the smallest possible cloth rectangle then treturns its profit value from solution space.
   (c) Recursively compute the x-max which is max possible profit for the cuts along x-axis.
   (d) Recursively compute the y-max which is max possible profit for the cuts along y-axis.
   (e) memoize and return the max of x-max, y-max and the M[x][y]
5. M[x][y] is the maximum possible profit that could be made from the cloth os size (X,Y)

Proof of correctness

Base Case: If the profit made from the product pi requiring cloth of size (ai,bi) is ci then the profit made from the single cloth of size (ai, bi) can be maximum of its exisiting profit value and the profit value of product pi i.e. ci. This way all the elements in solution space with matching dimensions with any of the product dimensions will have max possible profit values

Induction Step:

For any given cloth piece of size (x and y), there are below two possibilites:

- we can cut the given cloth along x-axis to obtain two pieces of cloth with dimensions (x-x', y) and (x',y). Now we select x' such that the the profit made from both such pieces of cloth will be maximum.

- we can also cut the given cloth along y-axis to obtain two pieces of cloth with dimensions (x, y-y') and (x,y'). Now we select y' such that the the profit made from both such pieces of cloth will be maximum.

- Or we can decide to not cut the cloth at all and calculate its possible maximum profit.

If we take a max out of all above three cases, then we get the maximum possible profit out of given cloth of size (x,y)

Hence the optimal substructure will be: M[x][y] = max(M[x][y], x-max, y-max) where x-max and y-max are as below:

x-max = max ((M[x-i][y] + M[i][y] for all i in (1 to x-1) y-max = max ((M[x][y-i] + M[x][i] for all i in (1 to y-1)

Pseudo-code

Listing 4: Question 3

```
 1  INPUT: S = {p(i), where i is 1 to n} and X x Y sized cloth.
 2  OUTPUT: Max profit that could be made out of given cloth.
 3
 4  MAX_PROFIT(x, y)
 5      if (mem[x][y] != -1)
 6          return (mem[x][y])
 7
 8      if (x==1 and y ==1)
 9          return M[1][1]
10
11      x_max = 0
12      for each i in (1 to x-1)
13          x_max = max (x_max, (M[x-i][y] + M[i][y])
14
15      y_max = 0
16      for each j in (1 to y-1)
17          y_max = max (y_max, (M[x][y-j] + M[x][j])
18
19      profit = max (M[x][y], x_max, y_max)
20      mem[x][y] = profit
21      return profit
22
23  MIN_PROFIT_MAIN(S, X, Y)
24      # initialize solution space and memoization space
25      int M[X][Y] = {0}
26      int Mem[X][Y] = {-1}
27
28      # set base cases
29      for each pi in S
30           M[ai][bi] = max(ci, M[bi][ai])
31           M[bi][ai] = M[ai][bi]
32
33      return (MAX_PROFIT(X, Y))
```

Time complexity analysis

- In the worst case there can be X*Y unique calls to the function MAX-PROFIT.
- For loops through 12 to 13 and through 16 to 17 can run for X and Y times respectively. Hence the each call to function MAX-PROFIT() can take O(X+Y) time.

- Loop through 29 to 31 initilizes the profit values in M[][] for element matching any of the products pi's in dimension, hence it takes O(n) time .

Hence the total running time for aboev algorithm is X*Y(O(X+Y) + O(n) = O(X*Y(X+Y))+O(n)

Question 5

Let $w_1, w_2, ..., w_n$ be the weight sof the n items to be selected to fulfill the maximum weight constraint of W.

This problem cannot be solved in polynomial time when the weights of the items become larger, the run time them converges towards exponential time. To overcome this, we formulate a approximation algorithm that tries to improve the run-time at the cost of optimality of the solution.

Key Strategy: The main issue with the input set is the sizes of the wieghts are too big. To overcome this, we'll try to scale down all the input weights w_i's as well as the maximum weight bound W, by a constant factor and then run our 0-1 knapsack formulation (taught in the class) on the newly obtained scaled down problem set.

We define a constant K obtained using below equation:

$K = \epsilon * w_{max}/n$

where, 0 < ϵ <= 1 and w_{max} = max(w_i, for all i in 1 to n)

We'll call K as a scaling down factor. We can adjust the value of ϵ such that the scaling factor will sufficiently scale down the input weights.

Now we use factor K to scale down all input weights and the maximum weight constraint W as below.

For all i in 1 to n, w'_i = $floor(\frac{w_i}{K})$

And scaled down weight constraint W will be , W' = $floor(\frac{W}{K})$

Now we use this scaled down inputs and weight constraints with the below dynamic formulation (taught in the class), M(i, w) = max(M(i-1, w), M(i-1, w-w_i) + w_i)

Algorithm

1. We obtain the flexible constant K using the formula given above.
2. Scale down the weights of all the items using the scaling factor K, as shown above
3. Scale down the weight constraint W as well, using the scaling factor K as shown above.
4. Let weight constraint W' and $w'_1, w'_2, ..., w'_n$ be the scaled down version of the original problem. Now apply the 0-1 Knapsack algorithm taught in the class on this new problem, to get set S of items that must be selected.

Proof of correctness

BASE CASE:

Induction Case: Let W(OPT) be max weight filled by the optimal solution and let W'(S) denote the value of the max weight filled by the above presented approximate DP algorithm. Then:

Now for any i in (1 to n) we've, w'_i = $floor(\frac{w_i}{K})$

=> w'_i <= $\frac{w_i}{K}$

=> w'_i* K <= w_i and the difference between the two will be at-most be K, since we got rid of the floor.

=> Thus summing up for all n items, the most that the weight filled by the optimal solution OPT can decrease by is n*K. Hence

=> W(OPT) - K * W'(S) <= nK

At every DP step, we get a set of items that is optimal for the given scaled down problem instance. hence it must be as good as choosing set OPT for the inputs of smaller weights.

Therefore, W(S) >= K * W'(OPT)

=> W(S) >= W(O) - nK = W(OPT) - $\epsilon * w_{max}$ since W(O) <= W(OPT) and $K = \epsilon * w_{max}/n$

=> W(S) >= (1-$\epsilon) * W(OPT)$

Pseudo-code

Listing 5: Question 3

```
INPUT: weight constraint W and X = {$w_1, w_2,...,w_n$}
OUTPUT: Set S collection of items that must be selected to fulfill weight ↩
    constraint to the max.

SCALE_DOWN_WEIGHTS(W, X, w_max, n)
    # select epsilon to be between 0 and 1, excluding 0 and including 1.
    Set K = epsilon * w_max / n;
    for i in (1 to n)
        w'_i = floor(w_i / K)

    W' = floor(W / K)

    return (W', X' = {w'_1,...,w'_n})

O_1_KNAPSACK_DP(W, X)
    #Same as discussed in class

O_1_KNAPSACK _APPROX(W, X)
    w_max = 0
    S = {}

    W' = 0
    X' = {}

    for i in (1, n)
        w_max = max (w_max, w_i)

    (W', X') = SCALE_DOWN_WEIGHTS(W, X, w_max, n)

    S = O_1_KNAPSACK_DP(W', X')

    return S
```

Time complexity analysis

The time required to count scale down factor and to scale down all the input weights is O(n).

The time taken by DP formulation is O(n*W') = O(n*$floor(\frac{W}{K})$)

But we have reduced W to the O(n*$floor(\frac{w_{max}}{K})$)

Total time required is O($n^2 * floor(\frac{w_{max}}{K})$)
but $\frac{w_{max}}{K} = \frac{n}{\epsilon}$
hence total run-time is O($n^2 * floor(\frac{n}{\epsilon})$) = O($\frac{f(n)}{\epsilon}$)