Homework 8

8.2 1a.

item	weight	value
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

capacity $W = 6$

$$V[i,j] = \begin{cases} \max(V[i-1,j],\ v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0 \end{cases}$$

$V(0,j) = 0$
$V(i,0) = 0$

		capacity i						
	j	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1=3$ $v_1=25$	1	0	0	0	25	25	25	25
$w_2=2$ $v_2=20$	2	0	0	20	25	25	45	45
$w_3=1$ $v_3=15$	3	0	15	20	35	40	45	60
$w_4=4$ $v_4=40$	4	0	15	20	35	40	55	60
$w_5=5$ $v_5=50$	5	0	15	20	35	40	55	65

b. Part A has one optimal subset which is {item 3, item 5} which has the value $65.

c. The table generated by dynamic programming can be used to tell whether there is more than one optimal subset by if and only if there is no equality between $V[i-1, j]$ and $v_i + V[i-1, j-w_i]$ during operation. If there is an equality then there is more than one, else there is only one optimal subset.

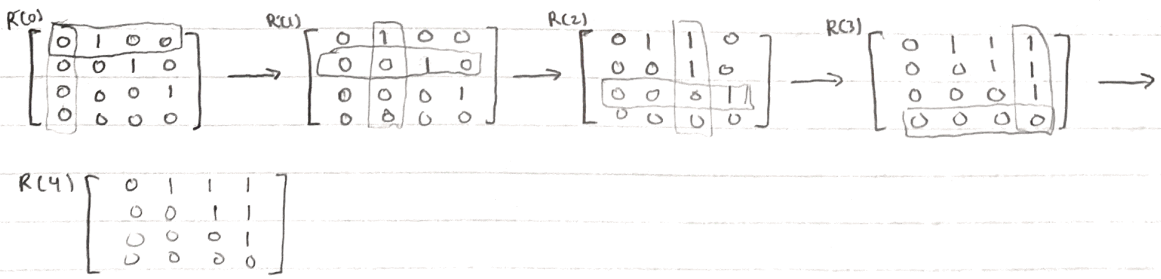
2a. Algorithm BottUp Knapsack(size, w[1...n], v[1...n]

```
// Input: size: the number of items, w[]: array of weights for n
          items, and v[]: array of values for n items
// Output: 2D array table[0...n, 0...size] which contains
          the value of the optimal subset.
for i ← 0 to size do
    table[0, j] ← 0
for j ← 0 to n do
    table[j, 0] ← 0
for i ← 1 to n do
    for ← 0 to size do
        if w[i-1] ≤ j
            table[i, j] ← max(table[i-1, j], v[i-1] + table[i-1, j - [i-1]])
        else
            table[i, j] ← table[i-1, j]
return table[n, size]
```

6.

i	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	—	—	45	45
3	0	15	20	—	—	—	60
4	0	15	—	—	—	—	60
5	0	—	—	—	—	—	65

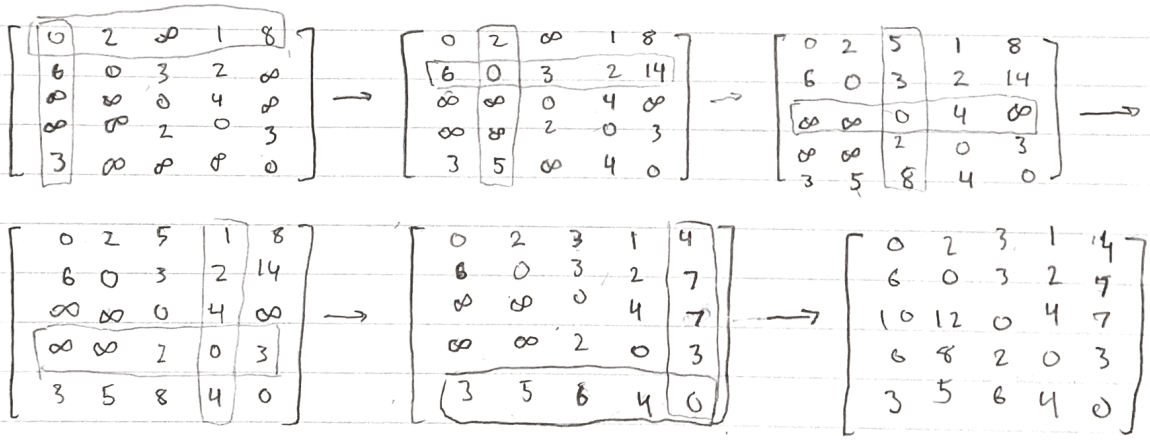
8.4 1.



2a. Warshall's algorithm has a cubic time efficiency because a graph has n vertices, thus the algorithm computes n matrices that have n^2 elements each. So n^3 elements is the total number of elements computed and each computation takes constant time. Thus the time efficiency is n^3.

b. The time efficiency of Warshall's algorithm is inferior to that of the traversal-based algorithm for sparse graphs represented by their adjacency list, because a traversal algorithm like BFS and DFS take $\Theta(n+m)$ since it has n vertices and m edges. Even if we did this traversal algorithm n times we would get $\Theta(n^2+m)$ which is just $\Theta(n^2)$ and $\Theta(n^2)$ is more efficient than $\Theta(n^3)$

7.



9.1 1. Algorithm GreedyChange(denominations[1...m], change)

// Input: array of coin denomination and positive integer of change owed

// Output: returns array that contains the number of coins of each denomination coins[1...m]

for i ← 1 to m do

coins[i] ← change / denominations[i]

change ← change mod denomination[i]

if change = 0 then

return coins

else

return "no solution"

The time efficiency of this algorithm is $O(m)$, because the algorithm loops through all of the denominations, which is 1 to m.

2 Algorithm GreedyAssignment(A[0..n-1, 0..m-1])

// Input: matrix with rows being people, columns being jobs

// Output: optimal solution to assigning people to jobs

Step 1: For each row, find smallest element, subtract it from every element in the row

Step 2: For every column, find smallest element, subtract it from every element in the column

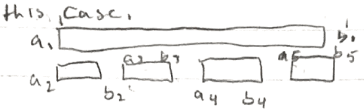
step 3: Cover all zeros in the matrix using minimum number of horizontal and vertical lines

step 4: If the minimum number of lines is n, an optimal assignment is possible and finished. Else move to step 5

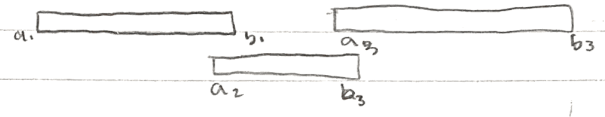
step 5: Determine smallest entry not covered by any line, subtract this entry from each uncovered row, and then add it to each covered column. Return to step 3.

This always yields an optimal solution.

4a. Earlist start first will not yield the optimal solution because there may be an interval that starts first but ends last. Thus the number of intervals will be one, which will not get the optimal answer. In this case.



b. Shortest duration first will also not yield the optimal solution because there may be a small interval that overlaps with two large intervals, connecting them. Hence the number of intervals will be one, while the optimal in this case is two.



c. Earliest finssh first will yield the optimal solution because the interval that finishes first will give other intervals that don't overlay the interval that ends first will get a chance to try and fit in. Thus yielding the optimal solution.

9b starting with node a

Tree vertices	Remaining vertices
a (-,-)	<u>b(a,3)</u>, c(a,5), d(a,4), e(-,∞), f(-,∞), g(-,∞), h(-,∞), i(-,∞), j(-,∞), k(-,∞), l(-,∞)
b(a,3)	c(a,5), d(a,4), <u>e(b,3)</u>, f(b,6), g(-;∞), h(-,∞), i(-,∞), j(-,∞), k(-,∞), l(-,∞)
e(b,3)	c(a,5), <u>d(e,1)</u>, f(e,2), g(-,∞), h(-,∞), i(e,4), j(-,∞), k(-,∞), l(-,∞)
d(e,1)	<u>c(d,2)</u>, f(e,2), g(-,∞), h(d,5), i(e,4), j(-,∞) k(-;∞), l(-,∞)

c(d,2) f(e,2), g(c,4), h(d,5), i(e,4), j(-,∞), k(-,∞), l(-,∞)

f(e,2) g(c,4), h(d,5), i(e,4), j(f,5), k(-,∞), l(-,∞)

i(e,4) g(c,4), h(d,5), j(i,3), k(-,∞), l(i,5)

j(i,3) g(c,4), h(d,5), k(j,8), l(i,5)

g(c,4) h(g,3), k(g,6), l(i,5)

h(g,3) l(i,5), k(g,6)

l(i,5) k(g,6)

k(g,6)

Thus we get ab(3), be(3), ed(1), dc(2), ef(2), ei(4), ij(3), cg(4), gh(3), il(5), gk(6).

15 Algorithm changeMinHeap(A, value, replace)

// Input: A min heap represented as an array A, value that is replacing an element's value, and a value being replaced (replace)

// Output: same min-heap, but with one element's value replaced with new value.

step 1: Create a new node and assign value (the replacing value)

step 2: Delete the node that contains the value that is being replaced

step 3: heapify the heap by making sure that each node maintains the min-heap properties.

The time efficiency is O(log n) because it takes overall time O(log n) + O(log n), because it takes O(log n) to delete a node and O(log n) to insert a node

9.2 16.

sorted list

de	cd	ef	ab	be	gh	ij	ad	cg	ei	ac	dh	fj	il
1	2	2	3	3	3	3	4	4	4	5	5	5	5

bf	hi	gk	hk	kl	jl
6	6	6	7	8	9

subgraph that does not create a cycle

de cd ef ab be gh ij cg ei il gk

$1 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + 4 + 5 + 6 = 36$

The minimum spanning tree is 36.

3. To make Kruskal's algorithm an algorithm that find a minimum spanning forest for an arbitrary graph, we need to change "while encounter < |V| - 1" to "while |E|". this will will allow, along with some additional code, the algorithm to find a minimum spanning forest because the algorithm will stop after exhausting the sorted lists of edges.

9.3 1a. To solve the single-source shortest path problem for directed weighted graphs, Dijkstra's algorithm would need to be adjusted to take into account edge directions when processing adjacent vertices.

b. To find the shortest path between two vertices of a weighted graph or digraph, the algorithm must start at any one of two given vertices and stop when the other vertex is add to the tree.

c. To find the shortest paths to a given vertex from each other vertex of a weighted undirected graph, solve the single-source problem with the given vertex as the source and reverse all paths obtained in the solution. If the weighted graph is directed, first reverse all edges, then solve the single source problem with the given vertex as the source, and finally reverse the direction of all paths obtained in the solution.

d. To solve the single-source shortest path problem in a graph with nonnegative numbers assigned to its vertices and length of a path defined as the sum of the vertex numbers on the path, creat a new graph by replacing every vertex v with two vertices v' and v'' that are connected by an edge whose weight is equal to the given weight of vertex v. All edges entering and leaving v in the original graph of v' will be entering and v'' will be leaving in the new graph, side note Assign weight of each original edge to zero.

2b.

a(-, ∞)	b(a, 3), c(a, 5), d(a, 4)
b(a, 3)	c(a, 5), d(a, 4), e(b, 3+3), f(b, 3+6)
d(a, 4)	c(a, 5), e(d, 4+1), f(b, 9), h(d, 4+5)
c(a, 5)	e(d, 5), f(a, 9), h(d, 9), g(c, 5+4)
e(d, 5)	f(e, 5+2), h(d, 9), g(c, 9), i(e, 5+4)
f(e, 7)	h(d, 9), g(c, 9), i(e, 9), j(f, 7+5)
h(d, 9)	g(c, 9), i(e, 9), j(f, 12), k(h, 9+7)
g(c, 9)	i(e, 9), j(f, 12), k(g, 9+6)
i(e, 9)	j(f, 12), k(g, 15), l(i, 9+5)
j(f, 12)	k(g, 15), l(i, 14)
l(i, 14)	k(g, 15)
k(g, 15)	

Shortest paths

a to b : a - b (3)
a to d : a - d (4)
a to c : a - c (5)
a to e : a - d - e (5)
a to f : a - d - e - f (7)
a to h : a - d - h (9)
a to g : a - c - g (9)
a to i : a - d - e - i (9)
a to j : a - d - e - f - j (12)
a to l : a - d - e - i - l (14)
a to k : a - c - g - k (15)

7. Algorithm ShortestDag (G, x)

// Input : weighted dag $G = (V, E)$ and vertex x

// Output: the length d_v of a shortest path from x to v and
penultimate vertex p_v for every vertex u in V

topologically sort the vertices of G

For every vertex v do

$d_v \leftarrow \infty$

$p_v \leftarrow$ null

$d_s \leftarrow 0$

for every vertex u in topological order do

For every vertex u adjacent to v do

if $d_v + w(v,u) < d_u$

$d_u \leftarrow d_v + w(v,u)$

$p_u \leftarrow v$

8. The minimum-sum descent problem can be solved by Dijkstra's algorithm by a top down approach. We take the apex of the triangle and set it as initial node and value 0. From there, we have two triangles, check both for minimal length and take the path with minimal length. Repeat this step for every layer of the triangle till you reach the bottom. The result will be minimum sum to descend.