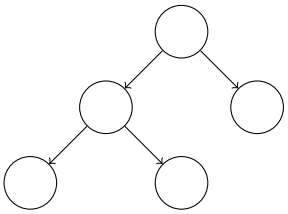
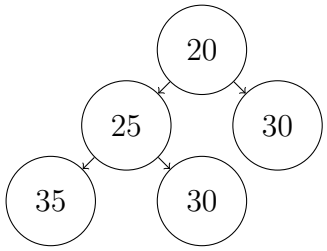
COSC 202, Exam 3 Practice

The solutions are not intended to be detailed. On the exam you need to provide answers in sufficient detail to get full credit. If you are uncertain what counts as sufficient detail for any particular question, please ask your instructor.

1. Suppose you start with an empty min-ordered heap and perform the following sequence of operations in the following order: insert 20, insert 10, insert 5, insert 25, insert 30, insert 35, remove min, insert another 30, and finally remove min.
   Give the location of each item in the resulting heap by filling in the nodes:
   Hint: After each insert or remove and before the next operation, the heap should be re-fixed for a min-ordered heap.



Solution:



2. Given an array C[ ] of size m and a target value X, the goal is to find the minimum number of coins required to reach the target X. For example, if C = [25, 10, 5] and X = 30, then the minimum number of coins is 2 (one coin of denomination 25, and one of denomination 5). Similarly, if C = [9,6,5,1] and X = 13, minimum number of coins is 3 (two coins of 6, one coin of 1). If it is not possible to get to a value X given C, then the code should return ∞. For example if C = [10,5,2] and X = 3, then it is impossible to get to that value given the coins we have, so it should return ∞.

   (a) Write the recurrence relation for MinCoin[X].

Solution:

$$MinCoin[X] = \min \begin{Bmatrix} MinCoin[X - C[0]] + 1 \\ MinCoin[X - C[1]] + 1 \\ \vdots \\ MinCoin[X - C[i]] + 1 \\ \vdots \\ MinCoin[X - C[m-1]] + 1 \end{Bmatrix}$$

(b) Formulate a recursive algorithm that implements the above recurrence relation.

```
int minCoin(int C[], int m, int X)
   Your code goes here
    return min
```

Solution:

```
int minCoin(int C[], int m, int X)
   if(X == 0)
     return 0
   int min = INT_MAX
   for(int i = 0 to m-1)
   {
      if(C[i] <= X)
      {
         int curr_min = minCoin(C, m, X-C[i])
         if(curr_min != INT_MAX && curr_min + 1 < min)
           min = curr_min + 1
      }
    }
    return min
```

(c) Formulate a Dynamic Programming implementation of the recurrence relation by completing the following code:

```
int minCoin(int C[], int m, int X)
    int coinChange[X+1]
    coinChange[0] = 0
    Your code goes here
    return coinChange[X]
```

Solution:

```
int minCoin(int C[], int m, int X)
    int coinChange[X+1]
    coinChange[0] = 0
    for(int i= 1 to X)
    {
        coinChange[i] = INT_MAX
        for(int j= 0 to m-1)
          if(C[j] <= i)
              int curr_min = coinChange[i- C[j]]
              if(curr_min != INT_MAX && curr_min + 1 < coinChange[i])
                  coinChange[i] = curr_min + 1
    }
    return coinChange[X]
```

INT_MAX stands for infinity.

(d) For the coin denomination of $C = [2, 5, 7]$ and $X = 10$, trace your code and find the minimum number of coins. Please show intermediate steps by filling the coinChange table below:

value	0	1	2	3	4	5	6	7	8	9	10
min coins											

Solution:

value	0	1	2	3	4	5	6	7	8	9	10
min coins	0	∞	1	∞	2	1	3	1	4	2	2

(e) What is the time and space complexity of the DP implementation in terms of X and m?

Solution: Time complexity is $\Theta(X \times m)$. Space complexity is $\Theta(X)$.

3. Recall the **pretty print** problem: Given an input vector representing lengths of n number of words: $W = [w_0, w_1, \ldots w_{n-1}]$, and an input integer representing maximum length L, we are to minimize sum of number of spaces at the end of each line. i.e., the penalty function is linear, $f(x) = x$. A greedy strategy fits as many words possible in the first line and repeats the process for other lines.

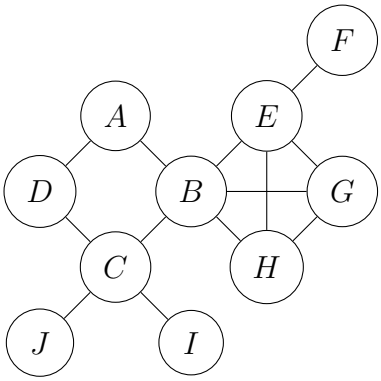
   (a) What is the runtime of the above greedy algorithm with respect to n and L?

   Solution: $\Theta(n)$

   (b) When $f(x) = x$, does the above greedy algorithm always reach optimum results? If yes, justify correctness. If no, describe a counter example.

   Solution: Yes.
   Proof of correctness: Imagine an alternative strategy **alter(n)** that claims to have a total slack less than **greedy(n)**, i.e., assuming **slack**[**alter(n)**] is less than **slack**[**greedy(n)**]. We will disprove this assumption. We first compare solutions word by word and identify the first word, w_i, where solutions diverge. Strategy **alter(n)** moves w_i to a new line, although it would have fit in the current line, while **greedy(n)** doesn't. By having w_i in a new line, **alter(n)** already has $w_i + 1$ higher slack than **greedy(n)**. The most optimum scenario for **alter(n)** is that the remaining words are rearranged such that all the $w_i + 1$ extra slack (1 for space) fill up empty slacks on the remaining lines. This way, **slack**[**alter(n)**] = **slack**[**greedy(n)**]. Under no circumstances can **alter(n)** save more spaces! By induction, this argument holds for the remaining words, solidifying that **greedy(n)** is indeed optimum.

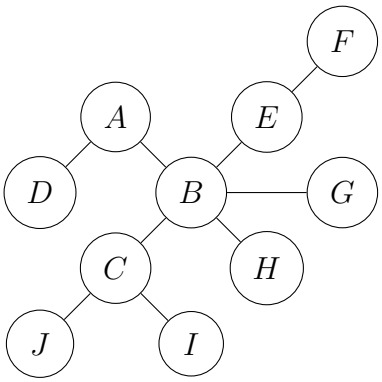
4. Consider the following connected, undirected, and unweighted graph. Suppose we start a Breadth-First Search (BFS) from vertex A as the source. Draw the resulting tree.



Hint: You can use a queue to keep track of vertices. Break ties alphabetically.

queue									

Solution:



queue	A	B	D	C	E	G	H	I	J	F

Note. values in the queue are for practice, only. Nodes will not all appear in the queue at the same time.

5. You are given a large database of webpages on the internet and your goal is to assign a score for how popular each page is. You can define popularity of some page x in terms of the number of webpages that reference x.

   (a) How could you encode the information in the database as a graph? What are the nodes and what are the edges?

   Solution: Nodes can be webpages. A directed edge from A $\rightarrow$ B means that page A references page B.

   (b) Describe how you could compute the popularity of each webpage. Analyze the time complexity.

   Solution: We need a reverse adjacency map where the keys are all the vertices and the value for any vertex x is a hashset of vertices with an incoming edge to x. The popularity of page x can be defined as the size of this hashset. Time complexity: O(V + E).