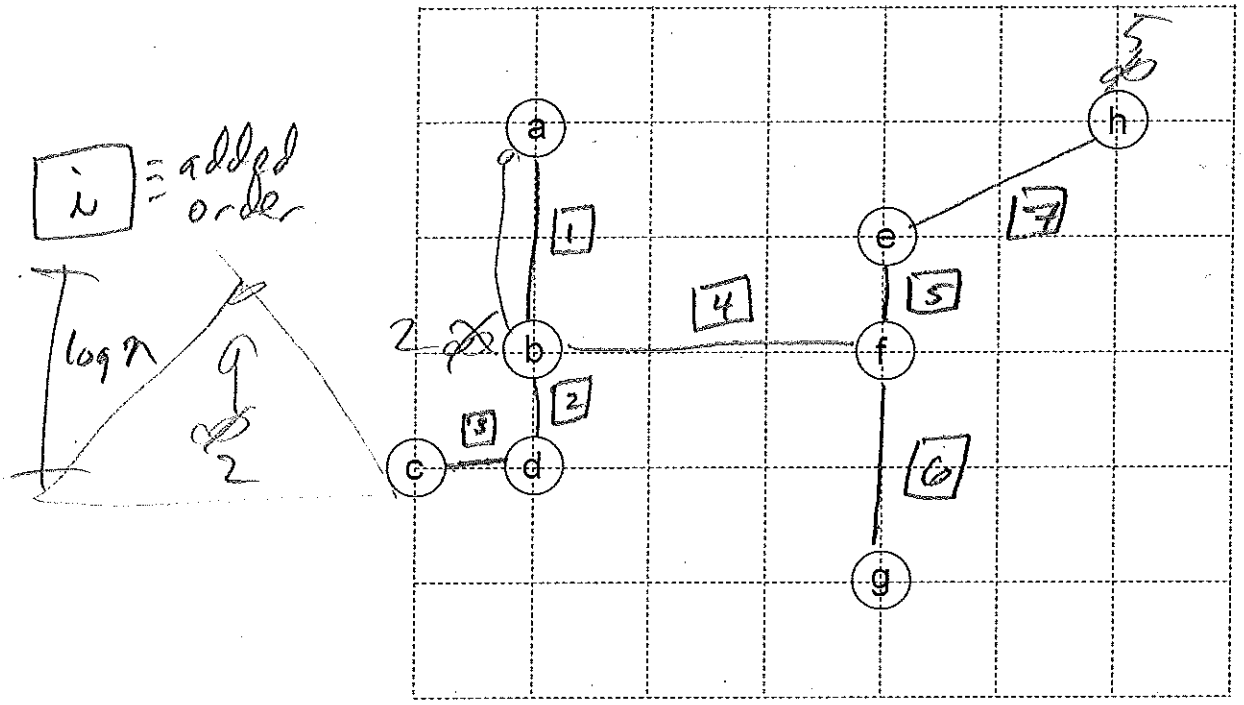
Data Structures (CS 1520) Lecture 27 Name:_____

1. Suppose you had a map of settlements on the planet X
(Assume edges could connecting all vertices with their Euclidean distances as their costs)



We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads build to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at "a".

a) Assuming we start at city "a" which city would you connect first? Why this city?

(b) because it's closest to (a)

b) What city would you connect next to expand your partial road network?

(d) because it is closest to partial road system

c) What would be some characteristics of the resulting "graph" after all the cities are connected?

no cycles ⇒ "tree"

d) Does your algorithm come up with the overall best (globally optimal) result? Yes

Min. Spanning tree, MST

2. Prim's algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a *greedy algorithm*. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices).

a) What greedy criteria does Prim's algorithm use to select the next vertex and edge to the partial minimum spanning tree? add vertex closest to partial road system.

b) What data structure would be useful it finding the next vertex to add to the partial MST? min. heap

The textbook's Prim's Algorithm code (Listing 7.12 p. 346) was incorrect, but **fixed on-line to**:

```
def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert) + currentVert.getDistance()
          if nextVert in pq and newCost < nextVert.getDistance():
            nextVert.setPred(currentVert)
            nextVert.setDistance(newCost)
            pq.decreaseKey(nextVert,newCost)
```

c) The `PriorityQueue` class used is shown on the next page and is similar to the `BinHeap` class we have been using, except:

- `self.heapArray` is a list of tuples with the first tuple value (tuple index 0) being the "priority" and second tuple value (tuple index 1) being its associated value.
  In Prim's algorithm what is the priority value? distance to partial road system
  In Prim's algorithm what is the associated value? vertex
- a `__contains__` method is added to check if a value is in the priority queue.
  In Prim's algorithm where is the `__contains__` method invoked?
- a `decreaseKey` method is added to allow a priority value to be reduced (i.e., increasing its priority).
  In Prim's algorithm when is the `decreaseKey` method used?

d) As written what is the big-oh of each of the methods?

- `__contains__` method? O(n)
- `decreaseKey` method? O(n)

The `PriorityQueue` class including the `__contains__` and `decreaseKey` methods:.

```
class PriorityQueue:
    def __init__(self):
        self.heapArray = [(0,0)]
        self.currentSize = 0

    def buildHeap(self,alist):
        self.currentSize = len(alist)
        self.heapArray = [(0,0)]
        for i in alist:
            self.heapArray.append(i)
        i = len(alist) // 2
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self,i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapArray[i][0] > self.heapArray[mc][0]:
                tmp = self.heapArray[i]
                self.heapArray[i] = self.heapArray[mc]
                self.heapArray[mc] = tmp
            i = mc

    def minChild(self,i):
        if i*2 > self.currentSize:
            return -1
        else:
            if i*2 + 1 > self.currentSize:
                return i*2
            else:
                if self.heapArray[i*2][0] < self.heapArray[i*2+1][0]:
                    return i*2
                else:
                    return i*2+1

    def percUp(self,i):
        while i // 2 > 0:
            if self.heapArray[i][0] < self.heapArray[i//2][0]:
                tmp = self.heapArray[i//2]
                self.heapArray[i//2] = self.heapArray[i]
                self.heapArray[i] = tmp
            i = i//2

    def add(self,k):
        self.heapArray.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def delMin(self):
        retval = self.heapArray[1][1]
        self.heapArray[1] = self.heapArray[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapArray.pop()
        self.percDown(1)
        return retval

    def isEmpty(self):
        if self.currentSize == 0:
            return True
        else:
            return False

    def decreaseKey(self,val,amt):
        # this is a little wierd, but we need to find the
        # heap thing to decrease by looking at its value
        done = False
        i = 1
        myKey = 0
        while not done and i <= self.currentSize:
            if self.heapArray[i][1] == val:
                done = True
                myKey = i
            else:
                i = i + 1
        if myKey > 0:
            self.heapArray[myKey] = (amt/self.heapArray[myKey][1])
            self.percUp(myKey)

    def __contains__(self,vtx):
        for pair in self.heapArray:
            if pair[1] == vtx:
                return True
        return False
```

tuples with priority/distance

value associated with priority / vertex key

3. If we want to speed-up the `__contains__` and `decreaseKey` methods, then what type of data structure could we add to aid in: checking for the existence of a key value, and if a key value exists in the heap, then at what index does it reside?

dictionary: keyToIndexDict

4. What modifications would need be needed to other methods to keep the data structure in question 3 up-to-date?

del keyToIndexDict[retval]

keyToIndexDict[self.heapArray[1][1]] = 1

vertex

new distance

linear search to find vertex to decrease

O(n)

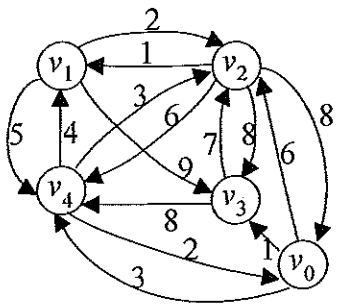
new tuple

O(log n)

list of tuples

O(n)

1. *Traveling Salesperson Problem* (TSP) -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A *tour* (or *Hamiltonian circuit*) is a path from a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertice, it does not matter where you start, so we will generally start at v_0.)



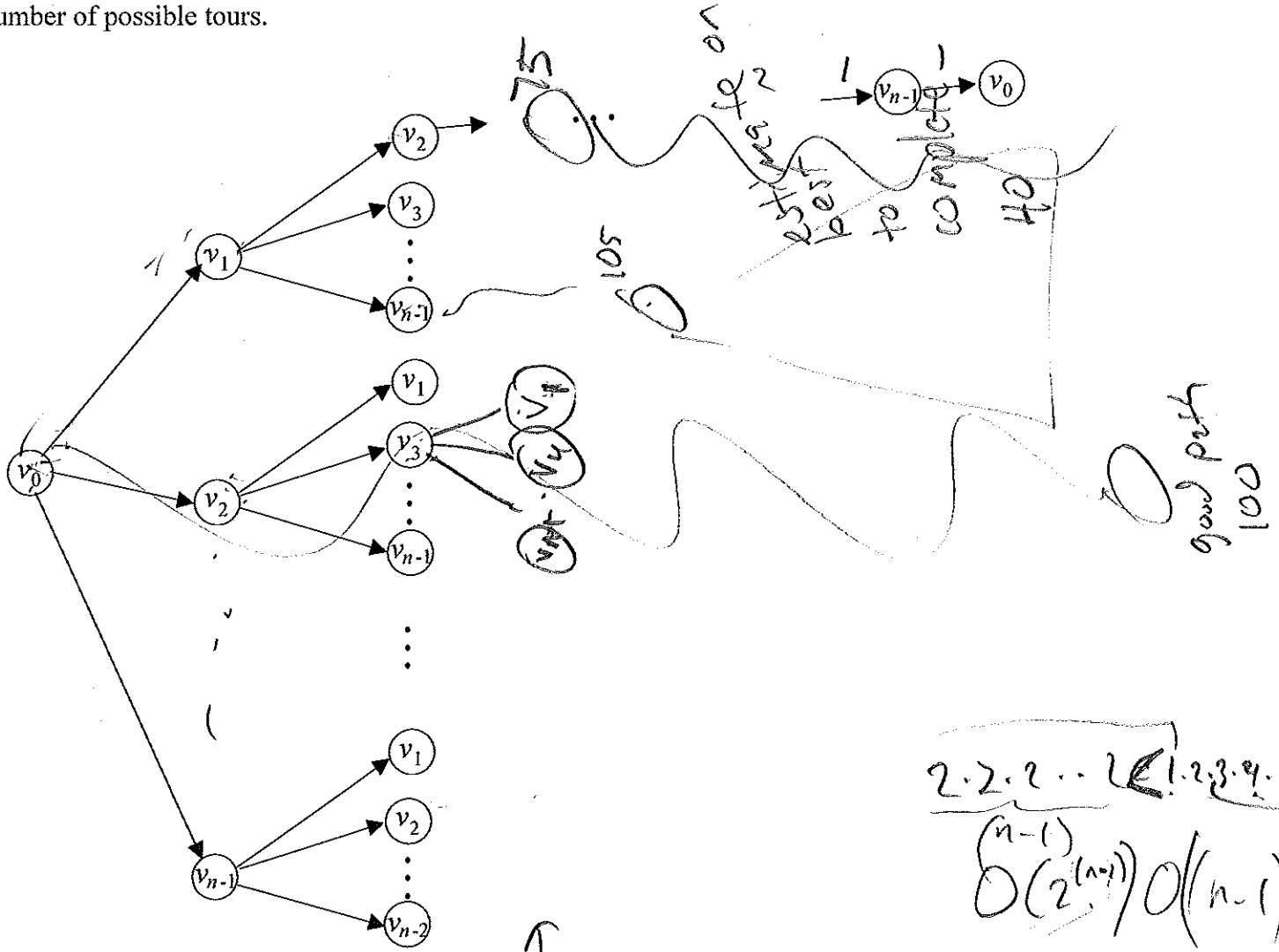
What are the length of the following tour?

a) $[v_0, v_3, v_4, v_1, v_2, v_0]$ 1 8 4 2 8 = 23

b) List another tour starting at v_0 and its length.

$[v_0\ v_3\ v_2\ v_1\ v_4\ v_0] = 16$

c) For a graph with "n" vertices (v_0, v_1, v_2, ... , v_{n-1}), one possible approach to solving TSP would be to brute-force generate all possible tours to find the minimum length tour. "Complete" the following decision tree to determine the number of possible tours.



$1 \quad (n-1) \quad (n-1)(n-2) \quad (n-1)(n-2)(n-3) \quad (n-1)(n-2)\cdots 2\cdot 1 = (n-1)!$

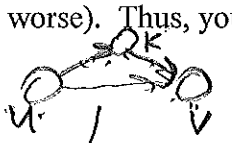
$2 \cdot 2 \cdot 2 \cdots 2 < 1 \cdot 2 \cdot 3 \cdot 4 \cdot (n-1)$

$O(2^{(n-1)})\ O((n-1)!)$

Unfortunately, TSP is an "*NP-hard*" problem, i.e., no known polynomial-time algorithm.

$O(n^2)\ O(n^3)\ O(n^n)$

2. **Handling "Hard" Problems**: For many optimization problems (e.g., TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially ($O(2^n)$ or worse). Thus, you could wait centuries for the solution of all but the smallest problems!



Ways to handle these "hard" problems:

- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the 2^n worse solutions, e.g, Backtracking (section 4.6) and Best-first-search-branch-and-bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., TSP problem satisfying the triangle inequality, Fractional Knapsack problem
- Use an approximation algorithm to find a good, but not necessarily optimal solution

Backtracking general idea: (Recall the coin-change problem from lectures 10 and 13)

- Search the "*state-space tree*" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

My simple backtracking solution for the coin-change problem **without pruning**:

```
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
       return 1
    else:
       for i in coinValueList:
           if i <= change:
               numCoins = 1 + recMC(change - i, coinValueList)
               if numCoins < minCoins:
                   minCoins = numCoins
    return minCoins
```

Results of running this code:

```
Change Amount: 63 Coin types: [1, 5, 10, 25]
Run-time: 45.815 seconds
Fewest number of coins 6
Number of Backtracking Nodes: 67,716,925
```

Consider the output of running the backtracking code **with pruning** twice with a change amount of 63 cents.

```
Change Amount: 63  Coin types: [1, 5, 10, 25]
Run-time: 0.036 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 1-cent coins is 3
number of 5-cent coins is 0
number of 10-cent coins is 1
number of 25-cent coins is 2
Number of Backtracking Nodes: 4831
```

```
Change Amount: 63  Coin types: [25, 10, 5, 1]
Run-time: 0.003 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 25-cent coins is 2
number of 10-cent coins is 1
number of 5-cent coins is 0
number of 1-cent coins is 3
Number of Backtracking Nodes: 310
```

a) With the coin types sorted in ascending order what is the first solution found?

b) How useful is the solution found in (a) for pruning?

c) With the coin types sorted in descending order what is the first solution found?

d) How useful is the solution found in (c) for pruning?