This document contains summary of some of the items discussed as part of the COMS 311 Lectures. This document **does not replace the lecture materials**. This document may contain some topics that are not covered as part of the lecture; you will not be tested on those parts, they are made available to you for gaining further knowledge on topics/concepts that are related to class-lecture.

1 Some Series

Finite series. You should be able to prove-by-induction the following statements. Try it out to improve your skills on application of induction proofs.

$$1 + 2 + 3 + \ldots + n = n(n+1)/2$$

$$1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n+1)(2n+1)/6$$

$$1^3 + 2^3 + 3^3 + \ldots + n^3 = n^2(n+1)^2/2^2$$

$$1 + r + r^2 + r^3 + \ldots + r^{n-1} = (r^n - 1)/(r-1)$$

Infinite series.

$$1 + r + r^2 + r^3 + \ldots = 1/(1-r) \text{ when } r < 1$$

2 Semantics of Summation

Assume: $L \leq U$:

$$\sum_{i=L}^{U} f(i) = f(L) + f(L+1) + \ldots + f(U)$$

$$\sum_{i=L}^{U} f = \underbrace{f + f + f + \ldots + f}_{(U-L+1) \text{ occurrences of } f}$$

3 Runtime Analysis

Definition 1 (Dominance of function) *A function $f : \mathtt{N} \to \mathtt{N}$ is dominated by a function $g : \mathtt{N} \to \mathtt{N}$ iff*

$$\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 . f(n) \leq c.g(n).$$

Dually: A function $f : \mathtt{N} \to \mathtt{N}$ dominates a function $g : \mathtt{N} \to \mathtt{N}$ iff

$$\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 . f(n) \geq c.g(n).$$

c: scaling factor.

1. $f(n) \in O(g(n))$ iff f is dominated by g. The valuation of the function f is no worse than the valuation of scaled ($c\times$) valuation of function g for all $n \geq n_0$. Intuition: function f grows at most as fast as function g.

   The following concepts of function-domination relations is not covered in class.

2. $f(n) \in \Omega(g(n))$ iff f dominates g. Intuition: function f grows at least as fast as function g.

3. $f(n) \in \Theta(g(n))$ iff f dominates and is dominated by g. Intuition: function f and g grows at the same rate.

4. $f(n) \in o(g(n))$ iff f is dominated by g for any scaling factor c. Intuition: f grows slower than g.

5. $f(n) \in \omega(g(n))$ iff f dominates g for any scaling factor c. Intuition: f grows faster than g.

Some noteworthy conclusions: $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$, and $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$. $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ iff $f(n) \in \Theta(g(n))$.

Relationship with Limits. The limiting values for the functions can be used to infer the dominance and, in turn, their relationship in terms of $O, \Omega, \Theta, o, \omega$. For instance, the $\lim_{n\to\infty} f(n)/g(n)$ captures the ratio of the functions as n tends to infinity.

1. if the above limiting value is not infinity, then $f(n)$ grows at most as fast as $g(n)$, i.e., $f(n) \in O(g(n))$.

2. if the above limiting value is not 0, then then $f(n)$ grows at least as fast as $g(n)$, i.e., $f(n) \in \Omega(g(n))$.

3. if the above limiting value is neither 0 nor ∞, then $f(n) \in \Theta(g(n))$.

4. Think about the limiting values that would imply the relationship between f and g is either o or ω.

One of the rules that becomes very useful when dealing with limits of ratio of functions is the rule due to L'Hospital (French Mathematician). It states that if $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$, then $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} \frac{\frac{d}{dn}f(n)}{\frac{d}{dn}g(n)}$.

4 Overview of Dynamic Array

The following concept of amortized runtime for dynamic array will not be assessed. This is just presented as interesting/good-to-know information.

Dynamic array is used to store and retrieve data elements, when the actual number of elements is not known a priori and hence, it is not possible to "statically" allocate the array. The basic idea of dynamic array is

1. Allocate some initial space for array to store the data

2. If there is not enough space to store new data, double the space allocation and create a new array. Copy the data from the old array to new one.

Worst case runtime for inserting data to a dynamic array corresponds to the case, when the array capacity is full. In this case, there is a cost for copying the data resulting in runtime $O(n)$.

Overall time for inserting n elements in a dynamic array. Consider that initially, an array with capacity 1 is created. For each insert, there is an unit time cost for inserting the element and every time, the array size is doubled, there is a time cost for copying elements. Therefore, the time for inserting n elements is

$$\begin{aligned} & \underbrace{(1+1+\ldots+1)}_{n \text{ inserts}} + \underbrace{(1+2+2^2+\ldots+2^k)}_{k \text{ doubling of capacity}} \\ = \quad & n + (2^{k+1}-1) \end{aligned}$$

We know that 2^k is greater than or equal to n as otherwise, the array would not hold n elements. Proceeding further, $2^{k+1} = 2n$ and hence, the overall time for inserting n elements in the dynamic array is $O(n)$.

We say that the amortized runtime for inserting one element in the dynamic array is $O(1)$ (intuitively, this is the average over n inserts).

5 Chain Hashing

In the context of hashing, the two main points to remember and use are (a) the load factor of a chaining based hash table is number of elements to store divided by the capacity of the hash table, and (b) the runtime of search/delete operations in such a hash table is **expected**.

Objective is to create a dictionary of elements, where each element is identified using some key. Such a key can be of any type (string, integer) with the property that elements have unique keys. The domain of key can be (significantly) larger compared to the number of elements to be stored. (Example: ISU id has 9 digits; however, we do not have enough ISU members to exhaust all the 9 digit numbers.) The basic operations of storage needs to efficient: adding new elements, finding existing elements, searching for elements.

A hash function maps each key to some integer value such that the integer value can be used as an index of a storage location (e.g., array). Intuitively, given a key k and a hash function h, $h(k)$ is th index of the storage location where the element with key k is stored. Typically, the storage capacity is proportional to the number of elements being stored. Even when the capacity is larger, it is not guaranteed that the hash function will always map two different keys to two different storage location. This is because the domain of key is much larger than the number of elements being stored and hence is also much larger than the storage capacity.

When the hash function maps two or more keys to the same index of the storage location, it is referred as hash collision. To handle such unavoidable scenarios, each storage location, instead of having the capability of holding exactly one element, points to a linked list, where the list contains all the elements whose keys are hashed to this storage location (index). At a high-level, consider the storage to be of the form: an array of linked lists. We will term this as the hash table (not to be mixed with the built-in classes in programming languages), a conceptual data structure, and the process of hashing to this table is termed as chain hashing.

To insert an element to the hash table T, we need to apply the hash function h on the key k of the element, and add the element to linked list associated to $T[h(k)]$. The runtime is proportional to the cost of applying the hash function, which involves basic arith-operations and hence is $O(1)$.

To find an element in the hash table T, we need to apply the hash function h on the key k of the element, and perform linear search on the linked list associated to $T[h(k)]$. Hence, the runtime is proportional to the length of the linked list, which, in the worst case, is $O(n)$, where n is the number of elements in T. Therefore, the objective is to reduce the length of the list such that the $O(n)$ runtime overhead can be avoided.

Consider that the hash function is such that (a) each index of the hash table is equally likely to be the result of application of hash function on any key and (b) result of application of hash function on one key does not depend on or influenced by the result of hash function on a different key. Based on these two properties, one can infer that the probability that an element will stored in any location of the hash table T is $\frac{1}{m}$, where m is the total capacity of the array structure implementing T (due to property (a)). The expected length of a list at any index of the array implementing T is $\frac{n}{m}$ (due to property (b)). This is referred to as the load factor of the hashing, intuitively capturing that as n (elements being stored) increases compared to m (storage capacity), the hash collision increases. If m is chosen such that it is proportional to n, i.e., $m \in \theta(n)$, then $\frac{n}{m} \in O(1)$.

Therefore, the expected runtime for finding element in a hash table is $O(1)$.

6 Master Theorem

The master theorem provides an asymptotic analysis for recurrence relations, frequently encountered in the analysis of divide-and-conquer algorithms. Various formulations of the master theorem exist, but for the purpose of this course, we will exclusively refer to the version presented in Theorem 4.1 in the 4th edition of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function that is defined and nonnegative on all

sufficiently large reals. Suppose the recurrence $T(n)$ is defined on natural numbers n as

$$T(n) = aT\Big(\frac{n}{b}\Big) + f(n).$$

Then the asymptotic behavior of $T(n)$ can be characterized as follows:

Case 1: If there exists a constant $\epsilon > 0$ such that $f(n) \in O\big(n^{\log_b a - \epsilon}\big)$, then $T(n) \in \Theta\big(n^{\log_b a}\big)$.

Case 2: If there exists a constant $k \geq 0$ such that $f(n) \in \Theta\big(n^{\log_b a} \log^k n\big)$, then $T(n) \in \Theta\big(n^{\log_b a} \log^{k+1} n\big)$.

Case 3: If there exists a constant $\epsilon > 0$ such that $f(n) \in \Omega\big(n^{\log_b a + \epsilon}\big)$ and if $f(n)$ additionally satisfies the regularity condition $af\Big(\frac{n}{b}\Big) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

Note that here $aT\big(\frac{n}{b}\big)$ actually means $a_1 T\big(\lfloor \frac{n}{b} \rfloor\big) + a_2 T\big(\lceil \frac{n}{b} \rceil\big)$ for some constants $a_1 \geq 0$ and $a_2 \geq 0$ satisfying $a_1 + a_2 = a$.

7 Priority Queue/Heap

Heap is a tree structure, where

1. in any subtree the root (i.e., the parent) of the subtree has a value that is greater than (less than) equal to the values of its children.
2. the tree is almost complete. The leaf nodes are fill up the tree from left to right.

If the parent-node value is greater than or equal to that of its child-nodes, then the heap is termed as max heap; otherwise it is referred to as min heap. If each node in the heap tree has at most 2 children then the heap is called a binary heap. The height of a heap tree is $O(\log n)$ as the tree is balanced by definition.

7.1 Array-based Implementation

As the heap is an almost complete tree, the elements can be stored in an array while capturing the parent-children relationship (using the array index).

Let the root index is denoted by r. Typically, there are two ways the root index is decided: either 0 or 1. The parent-children indices relationship is based on r. It is as follows: given that parent is at index p, and its left-child is in index l and right-child is in index r.

$$l = \begin{cases} 2 \times p + 1 & \text{if } r = 0 \\ 2 \times p & \text{if } r = 1 \end{cases} \qquad r = \begin{cases} 2 \times p + 2 & \text{if } r = 0 \\ 2 \times p + 1 & \text{if } r = 1 \end{cases}$$

Similarly, one can identify the parent index from the index of the children ((`child-index` $-1)/2$ when $r = 0$; otherwise it is (`child-index`$/2$). The index of the last element n-th element is $n - 1$ when $r = 0$; otherwise it is n.

0	1	2	3	4	5	
| 10 |	30 |	15 |	100 |	50 |	80 |	if $r = 0$
1	2	3	4	5	6	if $r = 1$

min heap

0	1	2	3	4	5
| 100 |	80 |	30 |	50 |	10 |	15 |
1	2	3	4	5	6

max heap

7.2 Heap operations

The operation heapifyUp takes as input the index of an element, which is not in its correct position as per the heap property, in particular, it's value is lower than (higher than) the value of its parent in a min heap (resp. max heap). The operation pushes the element up the heap tree by exploring along the child-parent relationship.

```
// i: index from where heapify up is performed
// root is at index 1
// heap array is A: minheap

heapifyUp(int i) {
   if (i==1)
      return;
   if (A[i] > A[i/2]) // nothing to do
       return;
   else {
       swap(A[i], A[i/2]);
       heapifyUp(i/2);
   }
}
```

The dual operation is heapifyDown. In this case, the parent is not in correct position. The element is pushed down along the tree-path that contains the child with the larger (smaller) value for max heap (resp. min heap).

```
// i: index from where heapify down is performed
// size: number of elements in the heap-tree
// root is at index 1
// heap array is A: minheap

heapifyDown(int i, int size) {
  if (2i > size)  // i is leaf-index
      return
  if (2i+1 > size) // only left child exist
      j=2i;
  else            // both children exist
     if (A[2i] < A[2i+1]) // left is smaller
        j=2i;
     else         // right is smaller
        j=2i+1;
     if (A[i] < A[j]) // nothing to do
        return
     else {
        swap(A[i], A[j]);
        heapifyDown(j, size);
    }
}
```

The above operations are used to implement find-and-extract the maximal element for max heap or minimal element for min heap, and for inserting new elements in the heap.

```
// size: number of elements in the heap
```

```
// heap array: A
// root of the heap at index 1
// Example code for extracting the root of a heap
extractMin() {
   root = A[i];
   A[1] = A[size];       // write the last element to the root
   size--;               // update the size of the heap-tree
   heapifyDown(1, size); // heapifyDown to rearrange the heap
   return root;
}
```

```
// size: number of elements in the heap
// heap array: A
// root of the heap at index 1
// Example code for inserting an element e in the heap
insert(e) {
   size = size + 1; // number of elements is increased by 1
   A[size] = e;     // the new element is added to the last index
   heapifyUp(size); // heapifyUp to rearrange the heap
}
```

The run-time for the find-and-extract, insert, heapifyUp and heapifyDown are $O(\log n)$ as the operations involve exploration of the (potentially the longest) path in the tree, which is th height of the tree.

Consider inserting the following elements to form a heap.

```
1963 1776 1941 1963 1492 1865 1783 1804 1918 1941 2001
```

MakeHeap. The operation makeHeap rearranges the elements in an array such that heap property is satisfied after the rearrangement. The strategy applied in this algorithm is (a) to consider the tree representation of the array elements and (b) to rearrange elements to satisfy heap property starting from the lower levels of the tree. Note that, the sub-trees rooted at the leaf-level already satisfy the heap property as such trees contain just one element (that element being one of the leaf-nodes of the original tree).

The algorithm is as follows:

```
// A is the array to be arranged to be a heap
makeHeap() {
  size = number of elements in A;
  for (i=last-parent; i>=r; i--) // r: root of heap
     heapifyDown(i, size);
}
```

The last-parent is the highest index of the element that has at least one child. The runtime of the algorithm is $O(n)$. This is because, the maximum number of constant time operations (compare and swap in this case) that occur for elements at level l is l; the level describes the distance of any node from a leaf. All leaf nodes are at level $l = 0$. The number of nodes at level l is of the order $\frac{n}{2^{l+1}}$. Therefore, the runtime is defined by the series

$$1.\frac{n}{2^2} + 2.\frac{n}{2^3} + 3.\frac{n}{2^4} + \ldots + L.\frac{n}{2^{L+1}}$$

where L is the level of the root. Hence, $\frac{n}{2^{L+1}} = 1$ as there is only one root element. Proceeding further,

$$\begin{aligned} & 1.\frac{n}{2^2} + 2.\frac{n}{2^3} + 3.\frac{n}{2^4} + \ldots + L.\frac{n}{2^{L+1}} \\ = \; & \frac{n}{2}\left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots + \frac{L}{2^L}\right) \end{aligned} \tag{1}$$

Some basic algebra is needed to find the sum of the series in parenthesis above.

$$\begin{aligned} S &= \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots + \frac{L}{2^L} \\ \frac{S}{2} &= \frac{1}{2^2} + \frac{2}{2^3} + \ldots + \frac{L-1}{2^L} + \frac{L}{2^{L+1}} \end{aligned}$$

Subtracting the second from the first

$$\begin{aligned} \frac{S}{2} &= \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots + \frac{1}{2^L} - \frac{L}{2^{L+1}} \\ S &= \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \ldots + \frac{1}{2^{L-1}} - \frac{L}{2^L} \\ &= 2 \times \left(1 - \frac{1}{2^L}\right) - \frac{L}{2^L} \\ &= 2 \times \left(1 - \frac{2}{n}\right) - 2 \times \frac{\log(n) - 1}{n} \end{aligned}$$

Therefore, from Equation 1

$$\frac{n}{2}\left(2 \times \left(1 - \frac{2}{n}\right) - 2 \times \frac{\log(n) - 1}{n}\right) \in O(n)$$

7.3 Heapsort

Given an array of integers, invoke makeHeap to arrange the elements in the form of a heap. Then extract elements in a for-loop. If the heap is a min-heap, then the elements will be extracted in ascending order; otherwise, the elements will be extracted in descending order. The runtime is proportional to the sum of cost of makeHeap and iterative extract operations. For n elements, makeHeap is $O(n)$. For n sequential extract operations, the cost is

$$\log(n) + \log(n-1) + \log(n-2) + \ldots + \log(1) = \log(n!)$$

By Stirling's approximation: $\log n! \approx n \log n - n$. Therefore, the runtime for sorting using heap (heapsort) is $O(n \log n)$.

7.4 A typical application

Finding top-k elements in order or the k-th top element are problems where arranging the elements in a heap can reduce the overall runtime from $O(n \log n)$ (runtime for sorting all n elements) to ($O(n \log k)$ (runtime for managing a heap of size k). The strategy is as follows. Pick the first k elements from the array and form a minheap ($O(k)$) (if the objective is to find top-k largest elements) and then insert a new element from the array ($O(\log k)$). Extract the minimum element ($O(\log k)$) and discard it. Continue with the process till there are no elements left. The heap should now contain the k largest elements. This is because everytime we are discarding an element (using extraction process) that is smaller than at least k other elements.

8 Strongly Connected Components

Basic Definitions & Background. You have already read and learned about the depth-first exploration (we will refer DFS explore). The basic idea is to recursively "explore" from vertices that are yet to be marked visited/discovered. We will associated with each vertex two properties: starttime and endtime–they represent the order in which the recursive call to the vertex initiates and terminates, respectively. Initially, both starttime and finishtime of all vertices are undefined. Given a directed graph $G = (V, E)$, its DFS exploration is encoded as follows:

```
1. cnt = 0;
2. for all v in V
3.   if v does not have starttime then
4.     DFS(v);

5. DFS(v)
6.   starttime(v) = cnt; cnt++;
7.   for all v -> v'
8.     if v' does not have starttime then
9.       DFS(v');
10.  endtime(v) = cnt; cnt++;
```

Note that, not all edges of G results in a recursive call (Lines 7–9). The edges that results in recursive calls are referred to as tree edges; these edges constitute the DFS-tree resulting from the DFS exploration of the graph. There are three other types of edges:

1. forward edge: an edge from a vertex u to its descendent v in the DFS-tree. (starttime(u) < starttime(v) and finishtime(v) < finishtime(u));
2. backward edge (backedge): an edge from a vertex u to its ancestor v in the DFS-tree (dual of forward edge condition);
3. cross edge: any edge that is not a tree, forward or backward edge. This type of edge does not connect ancestor-descendants in the DFS-tree.

Definition 2 (Strongly Connected Component (SCC)) *An SCC is a subset C of vertices in the graph such that*

1. *Reachability property. every vertex in the subset C can reach every vertex in C. (Reachability does not imply that there is a direct edge between vertices)*
2. *Maximality property. No additional vertex can be added to the subset C without violating the the first property.*

There are two types of SCCs: a trivial SCC is a singleton set and a non-trivial SCC contains at least two vertices.

Definition 3 (Component/Quotient Graph) *A graph $G = (V, E)$ containing $C_1, C_2, \ldots C_k$ SCCs induces a component/quotient graph $G/SCC = (V', E')$ where $V' = \{1, 2, \ldots, k\}$ and $E' = \{(i, j) \mid \exists u \to v \in E, u \in C_i \wedge v \in SCC_j\}$. A component graph is a graph over SCCs of the original graph. This is a directed acyclic graph (DAG).*

A bottom strongly connected component (BSCC) is an SCC such that vertices in BSCC B can only reach the vertices in B. There can be multiple BSCCs in the graph.

8.1 Tarjan's SCC Detection Algorithm

The following algorithm for SCC is not covered in class.

Tarjan's SCC algorithm will use backedges to identify candidate vertices that belong the same SCC. We will proceed by introducing couple of concepts/properties that are central to understanding the algorithm. These concepts are defined in the context of DFS exploration. Given a graph $G = (V, E)$ containing $C_1, C_2, \ldots, C_k$ strongly connected components (SCCs), for a DFS exploration of the graph

1. the entry-vertex of an SCC is the first vertex in that SCC that is visited in the DFS exploration. That is, the starttime of the entry-vertex in an SCC is smaller than that of any other vertices in the same SCC.

2. the index(v) is the starttime of the vertex v and the lowlink(v) is the smallest index of vertex u, an ancestor[1] of v in the DFS tree, that v may reach.

Based on the above concepts, the following claims hold true (WLOG self-loops are not considered).

Claim 1 *For a DFS exploration, there is exactly one entry vertex per SCC.*

Why?

Claim 2 *In a DFS exploration, for any entry vertex v, if v belongs to a non-trivial SCC (containing more than one vertex), then there is a backedge to v.*

For a non-trivial SCC C, in a DFS exploration if v is the entry vertex, then it must be reachable from at least one other vertex u in C. As u and v belongs to the same SCC, and v is the entry vertex in the DFS exploration, u is reachable from v in the DFS tree, which, in turn, implies that the edge from u to v is a backedge.

Claim 3 *In a DFS exploration, for any entry vertex v of SCC C, index(v) = lowlink(v) and for all other vertices u (if one exists) in C, index(u) > lowlink(u).*

Assume that v is an entry vertex of an SCC C and its index(v) $\neq$ lowlink(v); in particular, the index(v) > lowlink(v).[2] Therefore, v can reach one of its ancestor w (with lowlink(v) = index(w)) in the DFS tree, which further implies that v and w can reach other and w must be in the SCC C. In other words, the w is another entry vertex of C, which contradicts claim 1.

The second part of the claim follows from the first part.

Claim 4 *If a vertex v in an SCC has a tree edge (from the DFS tree) to another vertex u in another SCC, then lowlink(v) < lowlink(u).*

Use component graph to prove this.

The above claims are used in the following **steps for Tarjan's algorithm**.

1. For all vertices v that are yet to get an index value, start the recursive DFS exploration.
   (a) Initialize the index and lowlink values to be same (starttime of the recursion for the vertex).
   (b) Push v to a Stack.
2. For all children of v
   (a) Recursively explore all the children of v that are yet to get an index value. These may be children belonging to the same SCC as v, in which case, their lowlink can be less than or equal to the lowlink of v. Update lowlink(v) to a child's lowlink, if the latter is less than lowlink(v).
   (b) If a child u is already in the stack, then it must have been visited before v. If it is the case that the edge from v to u is cross edge, then v and u do not belong to the same SCC. This is avoided (see Step 3 below) by ensuring that at the time v is pushed to the stack only its ancestors from DFS tree are present in the stack. That is, the edge from v to u is a backedge; the lowlink(v) is updated to index(v) if the latter is less than the former.
3. After all the children have been examined, if the lowlink(v) = index(v), then one can infer that v is the entry vertex of some SCC. All vertices in the stack from the top till v belong to the same SCC. Pop the vertices from the stack until v is popped as well and add all the popped vertices to a new SCC.

[1] ancestor relationship of a vertex includes itself.
[2] Note that, index(v) cannot be less than lowlink(v) as the node itself is considered one its ancestor.

The claims 1, 2, 3 and 4 are used in the above algorithm—lowlink of each vertex is computed as DFS exploration proceeds, and when the exploration from a vertex is complete, if it is identified as the entry vertex in the DFS exploration, then that vertex along with its descendents (in DFS tree) that are not already part of some other SCC are placed in a new SCC.

We need to show that the lowlink is computed in a way (step 2) that satisfies claim 3 and popping vertices from the stack till the identified entry vertex is popped indeed identifies a new SCC (step 3). We will first prove an invariant property for the stack.

Claim 5 *In the above algorithm, when any vertex v is pushed to the stack, the contents of the stack are ancestors of v from the DFS tree.*

WLOG, assume that v is the first vertex (starting from the bottom of the stack) such that the stack contents below v contains at least one vertex u that is not v's ancestor in the DFS tree.

There must be some vertex w that is one of the common ancestors of both u and v, and u and v are in two different branches of the DFS tree rooted at w.

As u is not v's ancestor in the DFS tree, u cannot reach v. The exploration from u terminates before exploration from v starts (endtime(u) $<$ starttime(v)). Furthermore, all vertices in the stack below u are u's ancestors (as per our assumption). Therefore, on termination of exploration from u, one of the following holds:

1. u's lowlink is equal to its index; in which case u will be removed from the stack (step 3 of the algorithm); OR
2. u's lowlink is less than its index value, which implies there is some ancestor of u (below u in the stack) that has its lowlink equal to index value; in which case u is removed from the stack as well (step 3 of the algorithm).

u cannot be in stack below v if u is not v's ancestor, which contradicts our assumption.

Lemma 1 *Lowlink computed as per Step 2 of the above algorithm satisfies claim 3.*

The correctness of Step 2a of lowlink computation is immediate. The lowlink value is recursively updated (if needed) from the unexplored children. Either the child belongs to the same SCC as v or the child belongs to a different SCC. In case of former, the lowlink of the child is less than or equal to the lowlink of the v. In case of latter, the child cannot reach any vertex with index value less than v and therefore, its lowlink is not less than the lowlink already assigned to v and is not used to update lowlink of v (Claim 4).

The step 2b, on the other hand, may be the result of crossedge or backedge. The existence of crossedge implies the existence of some vertex in the stack that is not v's ancestor. This is invalidated by claim 5. Therefore, step 2b corresponds to only backedges, in which case the lowlink is correctly updated as well.

Step 2 only considers the tree edges and backedges.

Lemma 2 *Step 3 of the algorithm identifies **only** the vertices of the SCC to which v belongs.*

Assume that there is at least one vertex u in stack above v, when exploration from v finishes, and u does not belong to the SCC C_v of v. Therefore, u belongs to a different SCC C_u that can be reached from v and none of the vertices from that C_u can reach any vertex in C_v.

The entry vertex (say, w, which is not necessarily $\neq u$) of C_u is visited at least before u (below u in stack) and after v (above v in stack) due to DFS exploration; The lowlink and index value of w are equal (Lemma 1) and finishtime(w) $<$ finishtime(v). This implies, on termination of exploration from w, all vertices (including u), which are descendents of w and are present in stack, are removed from the stack and added to the C_u (Step 3 of algorithm).

Therefore, u cannot be present in the stack, when exploration from v finishes.

Lemma 3 *Step 3 of the algorithm identifies **all** the vertices of the SCC to which v belongs.*

Assume all vertices belonging to SCC of v are not present in the stack above v, when the recursive exploration from v finishes. Let u be one such vertex that belongs to the SCC of v but it not present in the stack above v. From Lemma 2, we know that u cannot be removed from the stack and added to a different SCC. This implies, u is not visited before the exploration from v finishes. This contradicts the property of DFS exploration: all descendents of v are visited before exploration from v finishes.

Pseudocode The correctness of the algorithm follows from the above lemmas.

```
1. cnt = 0;
2. for all vertices v in V
3.   if v does not have an index value
4.     TarjanDFS(v);

5. TarjanDFS(v)
    // Step 1
6.    index(v) = lowlink(v) = cnt; cnt++;
7.    push v to stack S; onstack(v) = true;
    // Step 2
8.    for all v -> v'
      // Step 2a
9.      if v' does not have an index value
10.       TarjanDFS(v');
11.       lowlink(v) = min(lowlink(v), lowlink(v'));
      // Step 2b
12.     elseif onstack(v') == true
13.        lowlink(v) = min(lowlink(v), index(v'));
    // Step 3: on termination of for all
14.   if index(v) == lowlink(v)
15.     create a new SCC C
16.     pop all vertices from stack till v (including v) and add to C
```

Properties to note The algorithm does not necessarily compute the lowlink of a vertex v as the smallest index of its ancestor. It computes lowlink such that the claim 3 is satisfied (Lemma 1).

Runtime of this algorithm is same as DFS: $O(|V| + |E|)$. Discuss why?

8.2 Kosaraju's SCC Detection Algorithm

Kosaraju's SCC detection algorithm primarily relies on the endtime of the vertices in DFS exploration, i.e., the relative ordering of the vertices in terms of the time when the exploration from vertices terminate. Recall that, if vertex v is an ancestor vertex u in a DFS exploration tree, then the endtime of v must be greater than the endtime of u.

Claim 6 *Given a graph G with SCCs $C_1, C_2, \ldots, C_k$, if there is a path from C_i to C_j in the quotient graph, then there exists some vertex in C_i whose endtime is greater than the endtimes of all vertices in C_j.*

The validity of the claim can be realized as follows. First consider the case that the DFS exploration proceeded by exploring the vertices in C_i before exploring the vertices in C_j. In this case, due to the nature of the DFS exploration, all vertices of C_j must be explored before the exploration completes from all vertices in C_i. (Note that, there is no path from C_j to C_i in the quotient graph.) Therefore, there exists one vertex whose DFS call terminates after the DFS call to all vertices in C_j terminates.

Next consider the case where the DFS exploration proceeded by exploring the vertices of C_j before exploring the vertices of C_i. As there is no path from C_j to C_i, the exploration from all vertices in C_j must terminate before any vertex in C_i will be explored.

Claim 7 *Given a graph G with SCCs $C_1, C_2, \ldots, C_k$, let e_i the maximum endtime of any vertex in C_i, then the descending order of SCCs in terms of e_i's results in topological ordering of vertices in quotient graph.*

This claim follows from Claim 6. Why?

Claim 8 *For any graph G, reversing the edges (resulting in transpose of G: G^T) does not alter the SCCs in G.*

Follows from the definition of SCC.

Claim 9 *Given a graph G with SCCs $C_1, C_2, \ldots, C_k$, if C_j is the SCC contains vertex with the largest endtime, then C_j is one of the BSCC in the G^T (transpose of G).*

(Review topological ordering and DFS)

Steps of Kosaraju's algorithm.

1. Phase 1
   (a) DFS explore the entire graph and as exploration terminates from a vertex, add it to the stack.
   (b) This ensures that the vertex with the highest endtime will be at the top of stack.

   Reverse the graph and perform the following:

2. Phase 2
   (a) Reinitialize visited property of all vertices and reverse the graph
   (b) Repeat the following till the stack is empty
      i. Pop the top of stack to get a vertex v
      ii. If v is not yet visited, DFS explore from v and add vertices visited as a result of the exploration to a new SCC
      iii. Otherwise, If v is already visited, then it must have been already part of some SCC; hence do nothing.

   The runtime of the algorithm is $O(V + E)$ because each step: Phase 1, reversing the graph and Phase 2 has the runtime $O(V + E)$.

```
kosarajuSCC(G=(V, E)) {
   initialize empty Stack S;
   dfsExplore(G); // Phase 1
   G' = reverseG(G);   // Reverse the graph
   getSCC(G');
}
```

Subroutines for the Phase 1:

```
// Phase 1
dfsExplore(Graph G=(V, E)) {
  for all v in V
    v.explored = false;  // initialize

  for all v in V {
    if (v.explored == false)
       dfs(v);
```

```
  // For topological ordering of DAG, you can
  // pop the vertices of the stack generated
  // in dfs subroutine: see below: and the vertices
  // will appear in topological order.
}

dfs(v) {
  v.explored = true;
  for all u in adj[v] { // that is, Neighbors of v
      if (u.explored == false)
          dfs(u);        // recursive exploration
  }
  add v to Stack S;      // this is a global stack
}
```

Reversing the graph:

```
ReverseG(G = (V, E)) {
 for all vertices v in V {
    for all vertices w in adj[v]
        add v to radj[w]
 }
 return G'=(V, E') where E' is radj;
}
```

Subroutines for Phase 2

```
// Phase 2: this will use radj relation: the reverse graph
getSCC(G'=(V, E')) {
  for all v in V
    v.explored = false;  // initialize

  scc = 1;
  while Stack S != empty {
      v = pop(S);
      if (v.explored == false) {
          dfsSCC(v, scc);
          scc++;
      }
  }
}

// reachability to get the SCCs of v
dfsSCC(v, scc) {
   v.explored = true;
   v.inSCC = scc;   // assign scc id to v
   for all u in radj[v] { // reverse graph
       if (u.explored == false) {
           dfsSCC(u, scc);   // add to u to SCC of v
       }
   }
}
```

Investigate the order in which SCCs are identified by ~~Tarjan's and~~ Kosaraju's algorithm.

9 Greedy Algorithms: Single Source Shortest Path Algorithm due to Dijkstra

Basic Definition & Background. In this section, we will consider weighted graphs, where each edge in the graph is associated with a weight, whose valuation is determined by a weight function. For any edge $e = (u, v)$, its weight is denoted by $wt(e)$. Recall that, a path graph is a sequence of vertices, where each successive vertices are connected by an edge relation. In the context of the weighted graph, we will say that a weight or length of a path is the sum of the weights of the edges that form the path. For instance, let $\pi = v_1, v_2, v_3, \ldots, v_k$ be a path from vertex v_1 to vertex v_k; then the length of the path is

$$\sum_{i=1}^{k-1} wt(v_i, v_{i+1})$$

One of the objectives is to find the shortest length path to all vertices from a specified vertex. This is referred to as the single source shortest path problem. Note that, when the weights of all edges are identical, we can apply BFS exploration strategy to address this problem. However, in the context of weights, pure BFS strategy is not a correct strategy. For instance, consider a graph, where v_0 has an edge v_1 and v_2, with edge weights 10 and 5, respectively; and v_2 has an edge to v_1 with edge weight 2. In this case, the shortest path distance from v_0 to v_1 is via v_2, with the distance being $5 + 2 = 7$ (which is strictly less than the directed edge from v_0 to v_1, a solution that would have been obtained using pure BFS).

We will first consider this problem in the context where all weights of **non-negative**. The solution to this problem is due to E. Dijkstra (Dijkstra algorithm). This algorithm uses two properties that hold for the problem:

1. Optimal substructure property: For any shortest path from u to v, if there is an intermediate vertex w, then subpath u to w and subpath w to v are shortest paths.

2. Greedy choice property: Consider S is a set of vertices for which the shortest distances $\delta(v)$ from source vertex is known. For all vertices $u \in V/S$ let $d(u) = min_{v \in S}\{\delta(v) + wt(u, v)\}$. Then, the vertex x with the minimal d-value is the one where $d(x) = \delta(x)$. Intuitively, one can greedily choose the the vertex that is closest to the source and conclude the shortest distance of the vertex. The proof of this claim depends on the negative weight values. The proof has been done in class.

Dijkstra algorithm with min-heap implementation of priority queue

```
Input G = (V, E, wt) and source s.

1. Initialize d(v) for all vertices to infinity, and v.parent = undefined;
2. d(s) = 0; s.parent = 0;
3. add all vertices to a minheap H with key being d-value
4. Initialize v.inQ = true for all vertices.
5. while H != empty
6.   v = extractMin(H);
7.   v.inQ = false;
8.   for all (v, v') in E and v'.inQ == true
9.       if (d(v') > d(v) + wt(v, v') then
10.           d(v') = d(v) + wt(v, v')
11.          v'.parent = v;
12.          updateHeap(H, v', d(v'))
```

The objective of function `updateHeap(H, v', d(v'))` is to bring v' to its correct position (using the standard heapifyUp subroutine) in the heap as the $d(v')$ valuation decreases in the line 10. This requires that we keep track of the indices of each vertex in the array implementation of the heap. One can realize that by setting a map-structure, M such that $M(v)$ returns the index of v in the array implementation of heap.

The parent-property associated with the vertices can be used to generate a shortest path from source to any vertex in the graph and the d-property captures the length of such shortest path.

The runtime of the algorithm $O((V + E)logV)$, where the factor $log(V)$ is due to the heap operations: extractMin and updateHeap.

10 Greedy Algorithms: Minimum Spanning Tree Due to Prim

Basic Definition & Background. As we have done in Section 9, we will consider weighted graph, in particular, weighted, undirected, connected graph without any constraint on the valuation of the weights. A spanning tree T for a weighted undirected graph G is a tree over the vertices of the graph, such that all vertices are connected (i.e., there is a path between any pair of vertices). As this is a tree, therefore, there is unique path between the vertices (no cycle); and as there is a path between any pair of vertices, it contains $|V| - 1$ edges, where V is the set of vertices in the graph. We will represent a spanning tree in terms of the edges that are present in the tree, e.g., $T = \{e_1, e_2, \ldots, e_k\}$ denotes a spanning tree containing edges e_1, e_2, ..., e_k.

A spanning tree is associated with a weight, which is computed as the sum of the weights of the edges that form the spanning tree. For a graph, there can be multiple spanning trees. For a spanning tree T, we will use $wt(T)$ to denote its weight. The spanning tree with the minimal weight is referred to as minimum spanning tree (MST). The objective is to identify a MST for an undirected, weighted, connected graph.

We will consider Prim's algorithm, which is based on greedy algorithmic strategy. As we have done in the previous section, we will consider two properties that hold for the MST problem and will use them to realize the algorithm.

1. Optimal substructure property: Consider a graph G and its MST T^*. If $e = (u, v)$ be an edge in T^*, then we can construct a graph G/e by contracting the edge e. This construction involves, merging the vertices u and v to form a new vertex, and adding all edges associated with u and with v with the new vertex (if there is an edge from u to x and v to x, then the edge with the minimal weight is added between the new vertex and x).

   The graph G/e is smaller than G (has one less vertex and has at least one less edge). We claim that if T is a MST of G/e, then $T \cup \{e\}$ is a MST of G. Intuitively, the MST of G can be obtained from the MST of G/e and e.

   The proof of this claim relies on few observations. First note that, $T^* - \{e\}$ is a spanning tree of G/e. This is immediate because, any connectivity between vertex-pairs other than the pair u and v is maintained in $T^* - \{e\}$ and the graph G/e contains all pairs of vertices except the pair u and v (which are merged).

   As T is MST of G/e, we conclude that

   $$wt(T) \leq wt(T^* - \{e\}) = wt(T^*) - wt(e) \tag{2}$$

   Next, consider the tree $T \cup \{e\}$. This tree is a spanning tree for the graph G. This is because all connectivity between pairs of vertices (other than the pair u and v) are maintained in T and $T \cup \{e\}$ (in particular the edges in T creates a partition of graph G where one group are reachable to and from u and another group reachable to-and-from v) . The addition of e allows for connectivity between u and v in G.

   The weight of the spanning tree $T \cup \{e\}$ is $wt(T) + wt(e)$, which is $\leq wt(T^*)$ (from Equation 2). Therefore, $T \cup \{e\}$ is MST of G.

2. Greedy Choice Property: Partition the set of vertices in G into A and B. We refer to this partition as a cut. The edges that connect vertices in group A with vertices in group B are referred to as cross-edges. The claim is that for a cut, the minimal weight cross-edge is part of some MST.

   Consider a cut A and B, and a minimal cost cross-edge $e = (u, v)$, where $u \in A$ and $v \in B$. Consider next a MST T^* that does not contain e. Our proof-strategy involves changing the edges in T^* to create a new MST that includes e.

   By our consideration, in T^*, u and v are not directly connected. Therefore, there must be a different (and exactly one) path that connects u and v. As these vertices are in two different groups, such a path must include a cross-edge between A and B. Let us denote that cross-edge as e'.

   Now, we argue (exchange argument) that we can create a spanning tree by replacing e' in T^* with e, i.e., $T = (T^* - \{e'\}) \cup \{e\}$. All the vertices that we connected (via some paths) due to the presence of e' are now connected (via possibly different paths) due to the inclusion of e. Note that, inclusion of e in place of e' does not result in any cycles in T. In short, T is a spanning tree.

   The weight of T is $wt(T) = wt(T^*) - wt(e') + wt(e)$. As e is a minimal cost cross-edge, $wt(e) \leq wt(e')$, which, in turn, implies that $wt(T) \leq wt(T^*)$. Therefore, T is a MST and it contains e.

The validity of the above properties in the context of MST leads to the following algorithm.

1. Select an arbitrary vertex v in G.
2. Create a cut $A = \{v\}$ and $B = V/v$ (all vertices other than v).
3. By greedy choice property, we know that the minimal weight cross-edge is part of a MST. Let $e = (v, v')$ be one such edge. We add e to our partial result for computing MST.
4. Using Optimal substructure property, we contract G with respect to e to obtain G/e and we know the MST of G includes the MST of G/e. In G/e, vertices v and v' are merged into one (let us call the merged vertex as vv'). Now our objective is to find the MST for G/e.
5. We repeat the process of creating a cut, now with $A = \{vv'\}$ and B containing the rest of the vertices of G/e.
6. Again, by greedy choice property, we know that the minimal weight cross-edge must be part of some MST of G/e. We add such an edge to our partial result.
7. The process continues (till contraction leads to a graph containing just one vertex) by contracting with respect to the newly added cross-edge.

The algorithm is realized as follows using minheap implementation of priority queue, which keeps track of the minimal weight cross-edge that connects a vertex in group B with a vertex in group A.

```
Input G = (V, E, wt)

1. Initialize d(v) for all vertices to infinity, and v.parent = undefined;
2. d(s) = 0; s.parent = 0;  // s is some arbitrary vertex
3. add all vertices to a minheap H with key being d-value
4. Initialize v.inQ = true for all vertices.
5. while H != empty
6.   v = extractMin(H);   // this is a vertex getting added to group A
7.   v.inQ = false;
8.   for all (v, v') in E and v'.inQ == true
9.       if (d(v') >  wt(v, v') then
10.           d(v') = wt(v, v')
11.          v'.parent = v;
12.          updateHeap(H, v', d(v'))
```

Notice the difference between above algorithm (MST) and the single source shortest path algorithm. The only difference in lines 9 and 10, where the d-values are being updated. In case of MST algorithm, the d-values indicate the closeness of vertices to any vertex in the partial MST (any vertex in group A) and in case of single source shortest path algorithm d-value indicate the closeness of vertices to the source vertex. Hence, for MST, the d-value of a vertex is updated based on the weight of the edge that connects the vertex to some vertex in group A; while for single source shortest path algorithm d-value of a vertex is updated based on the sum of the weight of the edge that connects the vertex to some vertex v (whose shortest distance from source is already known) and the distance of v.

11 Greedy Algorithms: Minimum Spanning Tree Due to Kruskal

The following algorithm for MST is not covered in class.

We will rely on the background on MST from the previous section.

The central claim of Kruskal's algorithm is that for any weighted, undirected and connected graph, the edge with minimum weight can be part of an MST. Intuitively, this makes sense because the edge connects two vertices and it does so by adding minimal weight to the spanning tree that we are trying to build. Formally, one can think of a cut where the vertices of the minimum weight edge are in two different groups of a cut and by greedy choice property, the minimum cost cross-edge belongs to some MST.

Once an edge e in the graph G is determined to be part of MST, we can apply contraction along e to obtain G/e. By optimal substructure property, if T' is MST of G/e, then $T' \cup \{e\}$ is MST of G. Note that, G/e is strictly smaller than G and we apply the same claim (as above) to identify an edge in G/e that should belong the MST of G/e (this is a minimal weight edge in G/e). The process is repeated to iteratively identify edges to form the MST until the contraction results in a graph with one vertex. This is realized in Kruskal's algorithm as follows:

```
1. Sort edges E in ascending order to form E';
2. T = emptyset; // MST edges will be added to T
3. Put each vertex in a set of its own;  // no MST edge is identified
4. for i=1 to |E|   // forloop over the number of edges
5.    E'[i] = (u, v); // E'[i] is the i-th edge as per ascending order
6.    if u and v are not in same set  // that means u v cannot reach each
                                      // other due to prior inclusion of some
                                      // other edges in T
7.       add E'[i] to T
8.       union the sets where u and v belongs
9. return T
```

Lines 2 and 3 indicate that as edges are yet to be selected for the MST, none of the vertices are connected. As edge is selected to be added to MST (Line 7), the vertices that are connected by the edge are placed in the same set (implicitly realizing contraction) (Line 8). Furthermore, note that using the condition at Line 6, only the edges that do not connected vertices in the same set are considered; this ensures that the edges that are relevant to the (contracted) graph are considered. In other words, edges between vertices, which are already connected (i.e., reachable) due to some other edges previously selected to be part of some MST, are not considered, thus avoid creating cycles (MST cannot contain cycles).

The runtime of the algorithm is driven by the sorting of edges $O(E \log E)$ (if mergesort is applied) and the overhead of the operations at Line 6 (**find** the sets where u and v belongs) and Line 8 (**union** two sets) that are repeated $|E|$ number of times. There is a special data structure union-find, which can perform this these operations efficiently: union in constant time and $|E|$ find operations in time proportional to $|E|\alpha)$ where α is inverse Ackermann's function (grows so slowly that it can be considered constant). More of these, if time permits, at the end of the semester.

Furthermore, if the edge-weights are such that the domain is proportional to the number of edges, then one can apply linear time sorting (for example, radix sort) to perform the operation in Line 1; this makes

the overall runtime $O(E)$.

12 Greedy Algorithms: Dealing of (Sets/Sequences) Scheduling

We have learnt about the application of greedy strategy for addressing optimization problem in the context of scheduling: interval scheduling and minimizing the maximum delay. These two problems illustrate the use of two different techniques for proving the claim that a strategy based on greedy choice property indeed leads to an optimal solution.

Greedy Stays Ahead Argument. For the interval scheduling problem, the claim for using greedy choice property states that: select the intervals in order of earliest finish time leads to an optimal solution. To prove this claim, we use the argument based on greedy stays ahead. In such an argument, we consider a sequence of interval generated by greedy-strategy and some optimal sequence. We, then, assume that for some arbitrary k, the first k elements in the sequences match. Using this information, and the greedy choice property, we proceed to show that the $k+1$-th element in the optimal sequence can be replaced by the $k+1$-th element from the sequence generated by the greedy-strategy, and the new sequence, thus obtained, is also optimal. In other words, there is some optimal sequence such that the first $k+1$ elements of this sequence and the sequence generated by the greedy-strategy match. Proceeding further, one can inductively construct an optimal sequence of length n matches with the first n elements in the sequence generated by greedy-strategy. Finally, we need to prove that the length of the sequence generated by greedy-strategy cannot be greater than n.

Exchange Argument. For the minimizing the maximum delay problem, the claim for using greedy choice property states that: ordering the jobs in terms of earliest deadline results in an optimal schedule. In this case, the proof technique uses exchange argument. First we start with an optimal schedule, which does not satisfy the "earliest deadline" ordering. This implies, there exists some inversion (prove the existence of inversion). We proceed by proving the exchanging the position of the jobs in the inversion leads to a new schedule that is also optimal. This allows us to remove all inversion eventually, resulting an optimal schedule which satisfies the greedy choice property.

Typically, if the optimization objective is to identify a subset of elements satisfying certain optimization criteria, then the claim based on a correct greedy choice property can be proved using greedy stays ahead argument. On the other hand, if the optimization objective is to identify an ordering of elements in a set satisfying certain optimization criteria, then the claim based on a correct greedy choice property can be proved using exchange argument.

13 Dynamic Programming

Dynamic Programming (DP) strategy is particularly suitable for efficiently solving optimization problem. This strategy is credited to Richard Bellman. The name of the strategy "dynamic programming" in Bellman's own words (from Eye of the Hurricane: An Autobiography):

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not

a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

13.1 Steps in realizing a DP algorithm

1. Identify a recursive formulation for solution of the problem. This is akin to applying divide and conquer strategy (number of subproblems to be solved may be exponential but if the number of distinct subproblems is polynomial, then DP strategy becomes effective)
2. Consider a dictionary-storage for keeping track of the result of recursive calls. The parameters of the recursion indicates the dimensions of the dictionary (one-to-one mapping).
3. Identify the dependency between the recursive calls, which, in turn, reveals the dependency between subproblems. Write an iterative computation strategy such that if subproblem P depends on the subproblem P', then the iterative strategy solves P' before P.

13.2 Case Study: Weighted Interval Scheduling

Recall that each interval has a start and end. Additionally for the weighted interval scheduling problem, each interval also has a weight. This is a classic example of application of DP. Instead of directly finding the set of intervals that is optimal (sum of the weights of the intervals in the set is maximized), the solution strategy focuses on identifying the maximal sum of the weights that can be realized by the optimal solution. Once such a maximal sum is identified, it is easy to identify the set that realizes that sum. This is typical of most of the case studies where DP is applied.

The recursive formulation relies on the decision of whether or not to include an interval in the final solution and the implication of such selection. The central theme in this formulation is that such decisions are made on intervals in a specific order. In class-lecture, the selected order is ascending order (AO) with respect to finish-times of the interval. An auxiliary information is pre-computed: $p(i)$ is the largest index j of an interval in AO that does not overlap with the i-th interval. Equipped with this, the function $opt(i)$ captures the maximal weight that can be realized from in intervals $1, 2, \ldots, i$. Therefore,

$$opt(i) = \texttt{max}\{wt(i) + opt(p(i)), opt(i-1)\}$$

$$opt(0) = 0$$

where $wt(i)$ if the weight of the i-th interval. The recursive formulation indicates (a) dictionary should be a one-dimensional array and (b) i-entry in the dictionary must be computed before $i+1$-entry. Therefore, the DP solution for computing opt (when the intervals are ordered as per the ascending order of finish times) is

```
dict[0] = 0
for i=1 to n
  if dict[i-1] < wt(i) + dict[p(i)] then
     dict[i] = wt(i) + dict[p(i)] then
  else
     dict[i] = dict[i-1]
return dict[n]
```

The function $p(i)$ can be computed using sorted order of arrays. Order (ascending) the intervals as per their end value in an array A. That is, the $A[i]$ contains the index for the interval k whose end value is i-th largest in the array A.

Similarly, order (ascending) the intervals as per their start value in an array B. Output with an array p, where the $p[i]$ is same as the function $p(i)$.

```
i=1; j=1;
while i <= n and j <= n
   if finish(A[i]) < start(B[j])
      i++;     // this means interval j in B has start time
               // after the finish time of the interval i in A.
    else
      p[B[j]] = A[i-1]
      j++
```

In the above algorithm, if finish of $A[i]$ is smaller than the start of $B[j]$, then we need to look at the interval $A[i+1]$ to check whether its finish is less than start of $B[j]$ or not. Otherwise, we know that $p[B[j]]$, i.e., the index of the intervals between $1 \ldots j$ that has the largest finish less than the start of $B[j]$, is $A[i-1]$.

The runtime for (pre)-computing $p(i)$ is $O(n \log n)$ due to the necessity to have sorted orders A and B. Similarly, note that the runtime for the `DP` algorithm is also $O(n \log n)$ as we need to pre-compute the $p(i)$. If we assume the sorted orders A and B given, then the runtime for the `DP` algorithm would be $O(n)$ (determined the for-loop).

To find the solution set. Given the dictionary, where $dict[n]$ holds the maximum sum of weights that can be realized by some subset of weighted intervals, the natural next step is to identify such a subset. Note that, there can be many subsets with this property. The follwing algorithm realizes the subset.

```
findSol(i) // returns a the subset from intervals 1...i
           // such that the maximum sum of weight is realized
           // dict[i] is already computed
  if i==0 return emptyset
  else
      if dict[i] = wt(i) + dict[p(i)] then
         result = i  union findSol(p(i))
      else
         result = findSol(i-1)
  return result
```

As an exercise, think about whether you can answer the question: how many subsets can realize the maximum sum of weights? Write an algorithm to compute that number.

13.3 Case Study: Subset Sum Problem

We are given a set of items, where each item has a value v_i (for computing purposes, you can consider that you are given a set of numbers). We are also given a budget B. The problem is to identify a subset of the items such that sum of the value of the items in the subset is $\leq B$ and the sum is also maximal.

As we have done in the previous section, we will focus on finding the maximal sum of values that can be realized. To develop the DP solution strategy for this problem, we need to first develop a recursive characterization for the solution. Let $SS(i, b)$ represents the maximal sum of values that can be realized from the items $1 \ldots i$, when the budget is b. Therefore,

$$SS(i,b) = \begin{cases} \texttt{max}\{v_i + SS(i-1, b-v_i),\ SS(i-1,b)\} & \text{if } b - v_i >= 0 \\ SS(i-1,b) & \text{otherwise} \end{cases}$$

$$SS(i,b) = 0 \quad \text{if } i = 0 \text{ or } b = 0$$

The first condition for the first recursion corresponds to the case where the i-th item can be part of the solution when v_i does not exceed the budget b. Therefore, the solution will the maximum of the sum of the values that can be attained by either including i-th items (in which case the budget available for the rest of items is $b - v_i$) or by excluding it[3]. If the i-th item cannot be added, then the solution for $SS(i, b)$ is identical to $SS(i-1, b)$. The base case for the recursive characterization corresponds to the case when no items can be considered or no budget is available.

Based on the recursive characterization, we observe that ordering in the recursive calls requires resolving the SS-calls for smaller valuations of the parameters (i and b) before resolving the SS-calls for the larger valuations of the parameters. *If we consider the case where the values of the items and the budget are integers*, based on the above observation, we can write an iterative algorithm using a dictionary as follows:

```
// input: array V, where V[i] is the value of the item i
//        budget B
// dict[i][b]: stores the maximum sum of values within the budget b

for (i=0 to n) dict[i][0] = 0;
for (b=0 to B) dict[0][b] = 0;
for (i=1 to n)
  for (b=1 to B)
     if (V[i] <= b)
        dict[i][b] = max{ V[i] + dict[i-1][b-V[i]], dict[i-1][b] }
     else
        dict[i][b] = dict[i-1][b]

return dict[n][B]
```

The runtime of the above algorithm is $O(nB)$. Though it may appear to be poly-time algorithm, note that the runtime depends on the valuation of one of the inputs (budget B) rather than the size of the input (which is $n + 1$). This algorithm is often referred to as *pseudo-polynomial*.

An accurate realization of the runtime using the size would require considering the number of bits that represent the budget B. If the number of bits representing the budget B is m, then the runtime would be $O(n2^m)$; thus revealing that the runtime is exponential with respect to the size of its input.

13.4 Case Study: Single Source Shortest Paths in Weighted Graph

This algorithm is due to Bellman-Ford. Consider the following functions in the context of a given source:

- $dist(i, v)$: shortest path distance from s to v using exactly i edges
- $sdist(i, v)$: shortest path distance from s to v using at most i edges. Therefore,

$$sdist(i, v) = \texttt{min}\{dist(0, v), dist(1, v), \ldots, dist(i, v)\}$$

- $sdist^*(v)$: shortest path distance from s to v using any number of edges. Therefore,

$$sdist^*(v) = \texttt{min}\{dist(0, v), dist(1, v), \ldots\}$$

We can write

$$\begin{aligned} sdist(i, v) &= \texttt{min}\{sdist(i-1, v),\ dist(i, v)\} \\ &= \texttt{min}(\{sdist(i-1, v)\} \cup \texttt{min}\{wt(u, v) + dist(i-1, u) \mid (u, v) \in E\}) \\ &= \texttt{min}(\{sdist(i-1, v)\} \cup \texttt{min}\{wt(u, v) + sdist(i-1, u) \mid (u, v) \in E\}) \end{aligned}$$

[3]Recall, the general strategy for all subset problem characterization is to consider two options for each item: **to include or not to include, that is the question**.

The last equality holds because $dist(u, i) \geq sdist(i, u)$ for all u and i. Now, we have a recursive definition for *sdist*, with the base case being

$$sdist(0, v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{cases}$$

The function $sdist^*(v)$ may not be well-defined in the presence of negative-weight cycles (cycles, where the sum of the weights of the edges is negative). This is because, one can go around a negative-weight cycle indefinitely to reduce the shortest path distance to some vertices (which will approach negative infinity).

Bellman-Ford algorithm not only identifies the correct shortest path distances in the absence of negative-weight cycles but can also identify the vertices whose shortest path distances are not well-defined when negative-weight cycles are present in the path from the source to those vertices.

The following claim holds when there is no negative weight cycle in any path from the source to the vertex v:

$$sdist^*(v) = sdist(|V| - 1, v)$$

. This claim is the basis for the Bellman-Ford algorithm. This claim holds because any path from the source to v with more than $|V| - 1$ edges will contain a cycle (based on the fact that a longest loop-free path in a graph with V vertices is $|V| - 1$). If there is no negative weight cycle in any path from source to v, then any cycle will not minimize the distance to v. That is, the shortest path distance from source to v can be computed by considering all the paths containing at most $|V| - 1$ edges ($sdist(v, |V| - 1)$). The function *sdist* is well-defined and can be computed using DP strategy.

The recursive structure indicates that (a) dictionary is two-dimensional and (b) the shortest path distances with at most i edges must be computed before the shortest path distances with at most $i + 1$ edges. This leads to following DP strategy for computing *sdist*.

```
dict[0][s] = 0; // s is the source
dict[0][v] = infty // for all other vertices
for i=1 to |V|-1
  for v in V
    dict[i][v] = dict[i-1][v];
    for (u,v) in E
      if dict[i][v] > dict[i-1][u] + wt(u,v)
         dict[i][v] = dict[i-1][u] + wt(u,v)
return dict
```

The runtime of the algorithm is $O(|V||E|)$.

Presence of negative-weight cycles. The corollary to the claim that led to Bellman-Ford is: When there is a negative weight cycle in the path from source to a vertex v, then the shortest path distance for v is not equal to $sdist(v, |V| - 1)$. A stronger claim can be made: if there is a negative weight cycle, then there exists at least one vertex v, for which $sdist(v, |V| - 1) > sdist(v, |V|)$. This is because if there is a negative weight cycle, then there exists at least one vertex that participates in that cycle and its shortest path distance becomes smaller after the cycle is explored one time (path containing at most $|V|$ edges indicates the cycle is explored at least one time in the path). Such a vertex can be identified by running the outermost for-loop of Bellman-Ford algorithm one more time and checking whether the dictionary entries in the $|V|$-row is identical to that in $|V| - 1$ row; if not, then there is a negative weight cycle in the graph. Also, one can identify the vertex (column in the dictionary) or vertices whose dictionary entry in $|V|$ row is smaller than that in $|V| - 1$ row; the shortest distances from source to these vertex (or vertices) are not well-defined due to the presence of negative weight cycles. The same is true for all other vertices that are reachable from these vertices; that is, simple BFS or DFS will reveal all vertices whose shortest path distances are not well-defined.

Side notes. One can compute the path that realizes the shortest path distance from source vertex by using dictionary entries (following the same strategy as we have done in weighted interval scheduling problem). One can also add a parent information to vertex v and update that information as `dict[i][v]` gets updated.

The algorithm can be optimized to stop early when the i-th row and $i-1$-th row have the same valuations in the dictionary. The algorithm can be also optimized by considering the for-loop over vertices and its neighbors as a single for-loop over all the edges.

These optimizations do not impact the runtime of the algorithm.

Think about whether the algorithm can be space optimized. Is it necessary to maintain the dictionary of size $O(|V|^2)$?