Asymptotic Notations

Let $f$ and $g$ be functions on real numbers. Then:

- $f(n) = O(g(n))$, if there are constants $c$ and $n_0$ so that $f(n) \leq cg(n)$, for $n \geq n_0$.

- $f(n) = \Omega(g(n))$, if there are constants $c > 0$ and $n_0$ so that $f(n) \geq cg(n)$ for $n \geq n_0$.

- $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. 
Theorem

Let \( f(n) = \sum_{i=0}^{k} a_i n^i \) (so \( f(n) \) is polynomial of degree \( k \)). Prove \( f(n) = O(n^k) \).

Proof.

Note that for \( n \geq 1 \), we have

\[
    f(n) \leq \sum_{i=0}^{k} |a_i| n^i \leq \sum_{i=0}^{k} |a_i| n^k \leq n^k \sum_{i=0}^{k} |a_i|,
\]

Now take \( n_0 = 1 \), and \( c = \sum_{i=0}^{k} |a_i| \). Then, \( f(n) \leq cn^k \), for \( n \geq n_0 \), and consequently, \( f(n) = O(n^k) \).
• Prove $n! = O(n^n)$.

• Is it true that $2^n = O(n^{1000})$?

• Prove $n^n = \Omega(2^n)$. 
• Show $\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}$.

• Calculate $\sum_{i=1}^{n} i$

• Calculate $\sum_{i=1}^{n} i^2$

• Can we calculate $\sum_{i=1}^{n} i^5$?

Estimate $f(n) = \sum_{i=1}^{n} 1/i$
For $i \leftarrow 1$ to $n$ do
For $j \leftarrow 1$ to $n$ do
  $S$
EndFor
EndFor
End.

• How many times $S$ is executed? $n^2$

For $i \leftarrow 1$ to $n$ do
For $j \leftarrow 1$ to $i$ do
  $S$
EndFor
EndFor
End.

• How many times $S$ is executed? $(\sum_{j=1}^{n} j) = n(n + 1)/2$
Measuring time

For $i \leftarrow 1$ to $n$ do
For $j \leftarrow 1$ to $i$ do
For $k \leftarrow 1$ to $j$ do
$S$
EndFor
EndFor
EndFor
End.

• How many times $S$ is executed?

\[ \sum_{i=1}^{n} i(i + 1)/2 \] (what is the value of this sum?)
Measuring time

Binary Search

• We are searching for \( X \) in a sorted array \( A \).

\[
\begin{align*}
l & \leftarrow 0; \quad H \leftarrow n \\
\text{While } L \leq H \text{ do} \\
Mid & \leftarrow (L + H \text{ Div}2) \\
\text{if } A[Mid] < X \text{ then } L \leftarrow Mid + 1 \\
\text{else if } A[Mid] > X \text{ then } H \leftarrow Mid - 1 \\
\text{else return } Mid \\
\text{EndWhile} \\
\text{Return NOTFOUND} \\
\text{End.}
\end{align*}
\]

• Time complexity is \( O(\log(n)) \): at each loop iteration \( (H - L) \) is divided by 2.
What is time complexity of the following program

Read integer $N$

$X \leftarrow N$

While $X > 0$ do

For $i \leftarrow 1$ to $X$ do

$S$

EndFor

$X \leftarrow (X \text{ div } 2)$

Endwhile

End.
Recurrence relations

Merge Sort

Mergesort(L, H)
If \( L < H \) then
\( Mid \leftarrow (L + H) \div 2 \)
Mergesort(L, Mid)
Mergesort(Mid + 1, H)
Merge(L, Mid + 1, H)
EndIf
End.

Merging is done in linear time

What is the time complexity of Mergesort?

Recurrence relation is \( T(n) = 2T(n/2) + n \): Now Solve this!
### Lists
- Lists are ordered multi sets.

### Major Operations
- Insert $O(1)$ time, delete $O(1)$ time, traverse $O(n)$ time.

### Implementations
- Arrays, and Linked lists

### Syntax?
Stacks, and Queues

- Stack is a list so that $O(1)$ operations insert: push and delete: pop are always done on one end.
- Queue is a list so that $O(1)$ time operations insert: InQ and delete: DeQ are always done on opposite ends.
Trees
- What is a tree?
- What is a binary tree?

Major Operations on Trees
- Tree Traversals: Preorder, Inorder, Postorder, $O(n)$ time, on a tree with $n$ vertices.

Priority Queues
A Priority Queue is a ADT that supports 2 major operations: Insert, and DelMin.
- DelMin Finds minimum element, and deletes it, whereas, Insert is a typical Insertion.
- In a List implementation Insert (at the beginning of list) takes $O(1)$ time, and DelMin takes $O(n)$ time.
A famous model or implementation for priority queues is a heap: A Near Complete Binary Tree so that the values stored at the children are $\geq$ the value stored at parent. A nice property of near complete binary tree on $n$ nodes is that its depth is $O(\log n)$. Heaps (or Priority Queues) are one of simplest and most effective data structures ever invented.

Insert and DelMin in a Heap

Insert Operation

Assume that we want to insert $x$ in a heap. A new leaf in the heap is created, where $x$ is placed. We then, repeatedly swap (or exchange), the content of the parent of the node, where $x$ is stored, and content of node where $x$ is stored, until, the heap property is satisfied, or we hit the root of the heap. Note that Insert takes $O(\log n)$ time, since the depth of a heap is $O(\log(n))$.

DelMin

We first delete the last leaf in the heap and place its content, or $x$, at the root of the heap. We then, repeatedly swap (or exchange) the content of node containing $x$, and the content of a child of this node, where the smaller data is stored, until the heap property is satisfied, or $x$ is stored at a leaf. Note that DelMin also takes $O(\log(n))$ time.
To sort $n$ numbers we first place them in a heap, in $O(n \log(n))$ time. (or even $O(n)$ time.) We then execute DelMin, $n$ times, and each time store the content of the deleted root. This process takes $O(n \log(n))$ time. The interesting part of this process is that the same array that stores the heap can also be used to store sorted numbers. So sorting is done inplace.
More General Operations on a Heap

Change Operation: Making the value smaller

Assume we make the content of a node smaller. To reconstruct the heap, we follow a process similar to Insert, with the exception that, we start the process, at the node where the change is made. So Change operation is done in $O(\log(n))$ time.

Change Operation: Making the value larger

Assume we make the content of a node larger. To reconstruct the heap, we follow a process similar to DelMin, by moving down, with the exception that, we start the process, at the node where the change is made. So this operation is also done in $O(\log(n))$ time.

General Delete Operation

We first delete the last leaf in the heap and place its content in the node to be deleted. We now a process similar to Change operation. It is important to note that we do not get to a cyclic situation.
Basic Definitions

Let $G = (V, E)$, be a graph, $|V| = n$, $|E| = m$. For any $x \in V$, the degree of $x$, or $\text{deg}(x)$, is the number of vertices adjacent to $x$. Maximum and minimum degree in $G$, are denoted, respectively, by $\Delta(G)$ and $\delta(G)$.

Data Structures for Graphs

- **Adjacency Matrix**: an $n \times n$ matrix $M$, so that $M[i, j] = 1$ if and only if $ij \in E$, requiring $O(n^2)$ storage.
- **Adjacency List**: A Collection of $n$ linked lists, so that list $i$ contains all vertices adjacent to vertex $i$, requiring $O(n + m)$ storage.
- **Adjacency List**: In the implementation, a one dimension array $L$, can be used, so that, $L[i]$ is list containing vertices adjacent to $i$. Alternatively, $L$ can be linked list itself.
Which Data Structure is Better?

• How fast $\delta(G)$ and $\Delta(G)$ can be computed?

• How fast $G$ can be updated, if a node $x$ is deleted?

• Given $x, y \in V$, how fast one can determine, if $xy \in E$?

• How effective we can simulate a dynamic network?
\( \alpha(G) \) or Chromatic Number of a Graph

\( \alpha(G) \) is the smallest number of colors assigned to vertices, so that endpoints of any edge are assigned different colors.

**Theorem**

\[ \alpha(G) \leq \Delta(G) + 1 \]

**Proof.** We use induction on number of vertices. If \( n = 1 \), then the claim is true. Now assume that claim holds for all graphs with \( n = k \) vertices, and let \( G \) be graph with \( n = k + 1 \). Remove any vertex \( x \) from \( G \), to obtain a graph \( G' = G - x \) on \( k \) vertices. Then by inductive hypothesis \( G' \) can be colored with \( \Delta(G') + 1 \leq \Delta(G) + 1 \) colors. Therefore there exists one color, among \( \Delta(G) + 1 \) colors, that is not used to color vertices in \( G' \) that are adjacent to \( x \). To complete the proof, we assign this color to \( x \).
We convert our proof to an algorithm for the assignment of colors. The input to the algorithm is the adjacency matrix of graph $G = (V, E)$, $|V| = n$. The main data structure is a one dimensional array $color$.

**Algorithm COLOR**

Order the vertices of $G$ as $x_1, x_2, ..., x_n$, and set $color[x_1] \leftarrow 1$.

For $i \leftarrow 2$ to $n$ Do

Select the smallest color $j$ that is not used to color any vertices adjacent to $x_i$ in $\{x_1, x_2, ..., x_{i-1}\}$.

$color[x_i] \leftarrow j$.

EndFor

End
Theorem

*Algorithm COLOR has time complexity of $O(n^2)$ and uses $\Delta(G) + 1$ colors.*

**Proof.** Clearly the algorithm uses $\Delta(G) + 1$ colors, since it is based on the our inductive proof. To prove the claim concerning time complexity it is sufficient to show that we can find in $O(n)$ time the smallest color that is NOT used for coloring the vertices adjacent to $x_i$. To do this at iteration $i$ of the For Loop, we construct, using the adjacency matrix and Color array, an auxiliary array $A[1..n]$ so that $A[j]$ is marked 1 if color $j$ used to color any vertices adjacent to $x_i$ which is indexed lower than $i$. We then scan $A$ in $O(n)$ time to find the smallest color which is not used to color any vertices adjacent to $x_i$ in $O(n)$ time. □
Breadth First Search (BFS)

BFS(s)

1. $InQ(s); \ visit[s] \leftarrow 1$
2. While $Q \neq \emptyset$ do
3.   $x \leftarrow DeQ()$
4.   For all $y \in G[x]$ do
5.     If $visit[y] = 0$ then $visit[y] \leftarrow 1; \ parent[y] \leftarrow x; \ InQ(y)$ endif
6.   EndWhile
BFS

Theorem

BFS can be implemented in $O(|V| + |E|)$ time and requires $O(|V| + |E|)$ storage, provided that $G$ is stored in the adjacency list format.

Applications of BFS

Finding a spanning tree, finding shortest paths from a source to all other vertices, finding connected components, determining if $G$ is acyclic, determining if $G$ is bipartite, can be done in $O(|V| + |E|)$ time.
Depth First Search (DFS)

DFS(s)

1. \( \text{visit}[s] \leftarrow 1 \)
2. For all \( y \in G[s] \) do
3. If \( \text{visit}[y] = 0 \) then \( \text{DFS}(y) \)

Theorem

DFS can be implemented in \( O(|V| + |E|) \) time and requires \( O(|V| + |E|) \) storage, provided that \( G \) is stored in the adjacency list format. Moreover using DFS, finding a spanning tree, finding connected components, finding biconnected components, determining if \( G \) is acyclic, and determining if \( G \) is bipartite, can be done in \( O(|V| + |E|) \) time.
Topological Ordering

Directed Acyclic Graphs: DAG

Let $G = (V, E)$ be a directed graph, $G$ is a DAG, if it does not contain any cycles. A topological ordering of a DAG is an ordering $v_1, v_2, ... v_n$ of vertices so that for any $v_i v_j \in E$, we have $i < j$. The following simple algorithm that uses a Queue finds a topological ordering in a DAG.

TOPO(G)

1. For each $x \in V$ do if $\text{indeg}(x) = 0$ then InQ($x$)
2. $i \leftarrow 0$
3. While $Q \neq \emptyset$ do
4. $v \leftarrow \text{DeQ}(); i \leftarrow i + 1; \text{Num}[i] \leftarrow v$
5. For all vertices $y$ adjacent from $v$ do
   $\text{indeg}(y) = \text{indeg}(y) - 1$; If $\text{indeg}(y) = 0$ then InQ($y$) Endfor
6. EndWhile
Theorem

Let $G = (V, E)$ be a DAG, then the previous algorithm finds a topological ordering of $G$ in $O(|V| + |E|)$ time.