CIT596 Notes

Robert Ottogalli

September 2021

Contents

List of Algorithms

Asymptotics

Name: Big Oh O

Use: To define the tightest upper bound of a function's run time. (Worst case scenario)

Definition 1: $f(n) = O(g(n))$ if $f(n) \leq k \cdot g(n)$ for some constant k

Definition 2: $f(n) = O(g(n))$ if there exist some constants k and n_0 such that for all $n \geq n_0$, $f(n) \leq k \cdot g(n)$ where $k > 0$ and $n_0 \geq 0$

Name: Big Omega Ω

Use: To define the tightest lower bound of a function's run time. (Best case scenario)

Definition 1: $f(n) = \Omega(g(n))$ if $0 \leq k \cdot g(n) \leq f(n)$ for some constant k

Definition 2: $f(n) = \Omega(g(n))$ if there exist some constants k and n_0 such that for all $n \geq n_0$, $f(n) \geq k \cdot g(n)$ where $k > 0$ and $n_0 \geq 0$

Note: In some cases, may be the same as the best case scenario.

Name: Big Theta Θ

Use: To define the expected run time of a function. Expressed as a value between the best case and worst case scenario.

Definition: $f(n) \in \Theta(g(n))$ if $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$ for some distinct constants c_1, c_2.

Formula: $\Omega(g(n)) \leq \Theta(g(n)) \leq O(g(n))$

Notation	Inputs	Runtime
Big O	worst-case	worst-case
Big Ω	worst-case	best-case
Big Θ	worst-case	expected case

Name: Properties of Asymptotic Growth Rates

Property 1: Transitivity
If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Property 2: Sums of Two: Big Oh
If two distinct functions share an upper bound, then the sum of those two functions shares the same upper bound.
If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.

Property 3: Sums of Series: Big Oh
If there is a fixed constant number k of functions, where $k > 2$, then the sum of all those functions share the same upper bound.
If there is a constant k, and there are functions $f_1, f_2, \cdots, f_k$ and h such that $f_i = O(h)$ for all i, then $f_1 + f_2 + \cdots + f_n = O(h)$.

Property 3: Sums of Two: Big Theta
If there are 2 distinct functions f and g (taking non-negative values) such that $g = O(f)$, then $f + g = \Theta(f)$.

Note: In general, $O(f + g) = \max\{O(f),\ O(g)\}$

Logarithm Rules

Definition: $\log_b(n) = x,\ \text{where } b^x = n$

Product: $\log_b(xy) = \log_b(x) + \log_b(y)$

Quotient: $\log_b(\frac{x}{y}) = \log_b(x) - \log_b(y)$

Power: $\log_b(x^y) = y \log_b(x)$

Base Switch: $\log_b(c) = \dfrac{1}{\log_c(b)}$

Base Change: $\log_b(x) = \dfrac{\log_c(x)}{\log_c(b)} = \log_c(x) \cdot \dfrac{1}{\log_c(b)}$

Log of 0: $\log_b(0)$ is undefined

Log of 1: $\log_b(1) = 0$

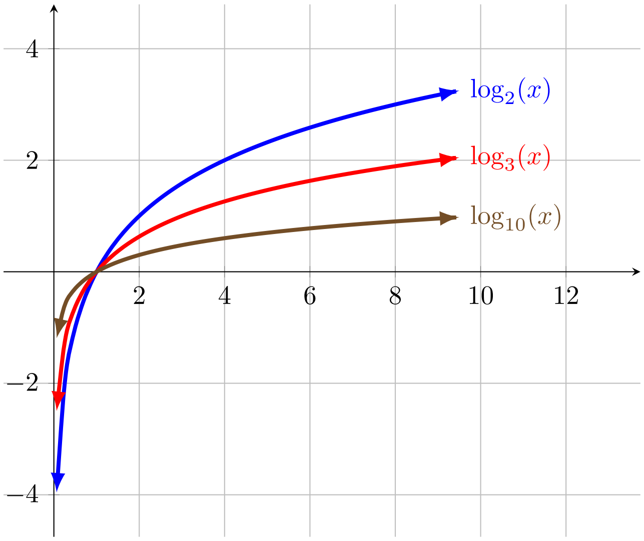
Log of base: $\log_b(b) = 1$

Identities:

1. $\log_b(b^a) = a$
2. $b^{\log_b(a)} = a$
3. $a^{\log_b n} = n^{\log_b a}$ for any $a, n > 1$

Comparisons: If $a < b$, then $\log_a(x) > \log_b(x)$
Example: $\log_2(x) > \log_3(x)$

Comparing Logarithm Bases



Logarithms of 2

n	1	2	3	4	5	6	7	8	16
$\log_2 n$	0	1	1.58	2	2.32	2.58	2.81	3	4

Notes:

1. A balanced recursion tree of n nodes has a depth of $\lceil \log_2(n) \rceil$.
2. The $\log_2(x)$ and $\log_3(x)$ functions in the plot above were represented using the base change principle, as the pgfplots package only accepts inputs in the form $\log_{10}(x)$. Thus, $\log_2 x$ was coded as $\dfrac{\log_{10} x}{\log_{10} 2}$.

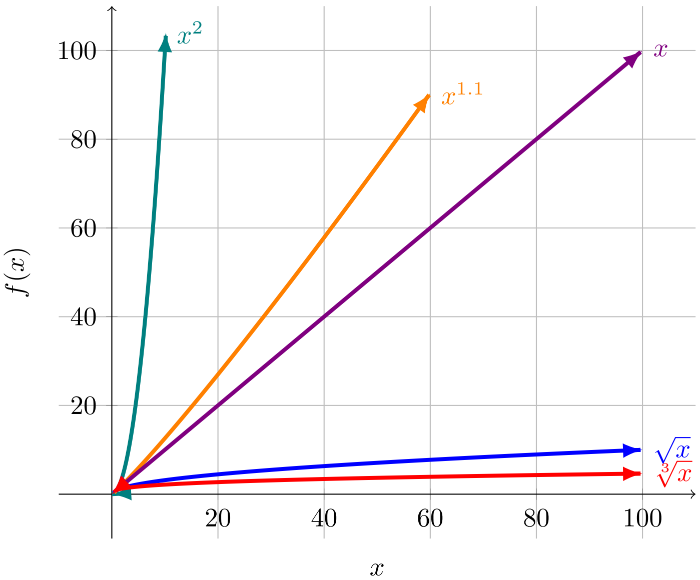
Name: **Asymptotic Bounds**

Polynomials: Time complexity functions of the form $O(n^x)$, $\Omega(n^x)$, or $\Theta(n^x)$ where $x > 0$ is some real number. Note that roots are polynomials, as given in the examples below:

$$\sqrt{2} = 2^{\frac{1}{2}}$$
$$\sqrt[3]{2} = 2^{\frac{1}{3}}$$
$$\sqrt[y]{x} = x^{\frac{1}{y}}$$

Note that:
1. Let $f(n) = a_0n^0 + a_1n^1 + a_2n^2 + \cdots + a_dn^d$. f is a polynomial of degree d. Thus, $f = O(n^d)$.
2. For $x > 1$, these functions grow relatively quickly. For $x < 1$, they grow more slowly than $x = 1$, and slightly more quickly than logarithmic functions of the form $\log(x)$.

Polynomials for Whole and Radical Values of x



Logarithms: Time complexity functions of the form $O(\log(n))$, $\Omega(\log(n))$, or $\Theta(\log(n))$. These functions grow very slowly.

Note that:
1. We can approximate how quickly $\log_b n$ grows with the formula: $\lfloor 1 + \log_2(n) \rfloor = x$ where x is the number of digits used to represent n.
2. For every $b > 1$ and every $x > 0$, we have $\log_b(n) = O(n^x)$. Thus, $\log_b(n)$ is bounded by $O(n^x)$.

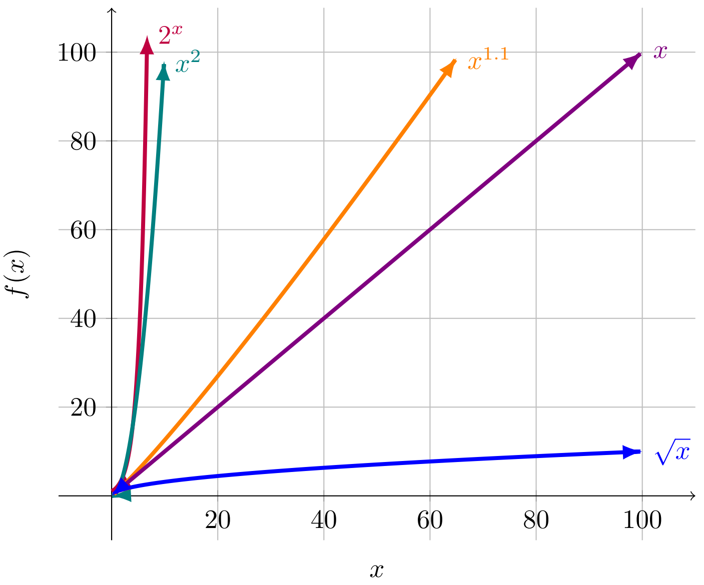
Exponents: Time complexity functions of the form $O(r^n)$, $\Omega(r^n)$, or $\Theta(r^n)$ where $r > 0$ is some real number.

Note that:
1. For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$.

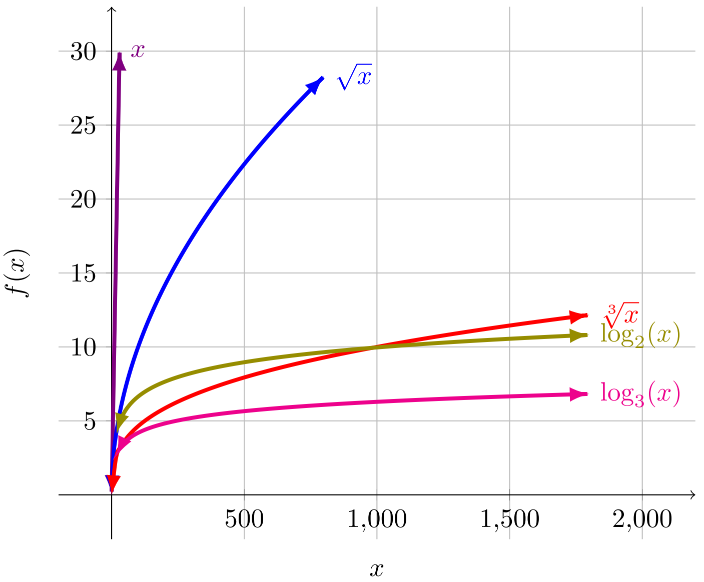
Note: Observe that for all exponential, polynomial, logarithmic, and constant functions, the run-time growth rate is:

exponents $>$ polynomials $(x > 1) >$ linear $>$ polynomials$(0 < x < 1) >$ logarithms $>$ constants

Exponential vs. Polynomial vs. Logarithmic



Linear vs. Radical Polynomial vs. Logarithmic



Important Algorithms

Fundamentals

Name: **Greatest Common Denominator (GCD)**

Definition: The greatest common factor between two different positive integers.

Facts: 1. Suppose $a > b$. If $d = \gcd(a, b)$, then $d = \gcd(b, a - b)$
2. Any common factor of a and b is also a factor of $a - b$.

Proof: If $b|x$ AND $a|x$, then $a = k_1x$ and $b = k_2x$ for some integers k_1, k_2.
Thus, $a - b = k_1x - k_2x = (k_1 - k_2)x$.
Therefore, the common factors of a and b are the same as the common factors of b and $a - b$.
Thus, the gcds of a, b, and $a - b$ are the same.

Name: **Euclid's Algorithm**

Use: To find the greatest common denominator (GCD) of two numbers.

Definition 1: Let $a = bq + r$, where q is the quotient of $\lfloor a/b \rfloor$ and r is the reminder of $a \mod b$.
Thus, $\gcd(a, b) = \gcd(b, r)$.

Definition 2: For any nonnegative integer $a \geq 0$ and any positive integer $b > 0$ $\gcd(a, b) = \gcd(a, a \bmod b)$

Algorithm: Euclid's Algorithm

Algorithm 1 Euclid's Algorithm

```
Runtime: $O(\log n)$
procedure EUCLID($a, b$)
    if $b = 0$ then
        return $a$
    else
        return EUCLID($b, a \bmod b$)
end procedure
```

Example:

$$\begin{aligned} \textsc{Euclid}(30, 21) &= \textsc{Euclid}(21, 9) \\ &= \textsc{Euclid}(9, 3) \\ &= \textsc{Euclid}(3, 0) \\ &= 3 \end{aligned}$$

3 is the GCF of 30 and 21.

Name: **Extended Euclid's Algorithm**

Use: Modifies Euclid's Algorithm to find integers x, y such that $ax + by = \gcd(a, b)$.

Algorithm: Euclid's Algorithm

Algorithm 2 Extended-Euclid's

```
Runtime: $O(\log n)$
procedure EXTENDED-EUCLID($a, b$)
    if $b = a$ then:
        return $(a, 1, 0)$
    else
        $(d', x', y') = $ EXTENDED-EUCLID($b, a \bmod b$)
        $(d, x, y) = (d', x', y' - \lfloor a/b \rfloor y')$
        return $(d, x, y)$
end procedure
```

Example: $\gcd(74, 44)$

$$\begin{aligned}
30 &= 74 - 44 = 1(74) + (-1)(44) \\
14 &= 44 - 30 = 1(44) - (1(74) + (-1)(44)) = 2(44) - 1(74) \\
2 &= 30 - 2(14) = (1(74) + (-1)(44)) - 2(2(44) + (-1)(74)) \\
&= 3(74) + (-5)(44) \\
\gcd &= 2; x = 3; y = -5
\end{aligned}$$

Name: **Divide and Conquer Strategy**

Use: Used to break a problem into smaller pieces, and solve each piece recursively.

Steps:

1. Divide: Break the input (n, array, etc.) into roughly equal halves.
2. Conquer: Solve the problem in both halves. (Recursively)
3. Combine: Merge the solutions of the 2 halves into the solution for the whole problem.

Example: The Merge Sort algorithm (see below) is a classic example.

Name: **Solving Recurrence Relations**

Explanation: The Divide and Conquer Strategy leads to solving sub-problems of decreasing size, resulting effectively in a tree where each node represents some n or fraction of n problems. The contributing factors to the solution of a recurrence can come predominantly from the root node or leaf nodes, or be balanced across all nodes.

Patterns: Solution contributions from levels have the following patterns:

Contribution from levels	**Solution**
Decreasing geometric series	Root level dominant
Increasing geometric series	Leaf level dominant
Equal for all levels	Number of levels dominant

Formula: This type of scenario can be solved using the **Master Theorem**, given in the formula below.

The Master Theorem

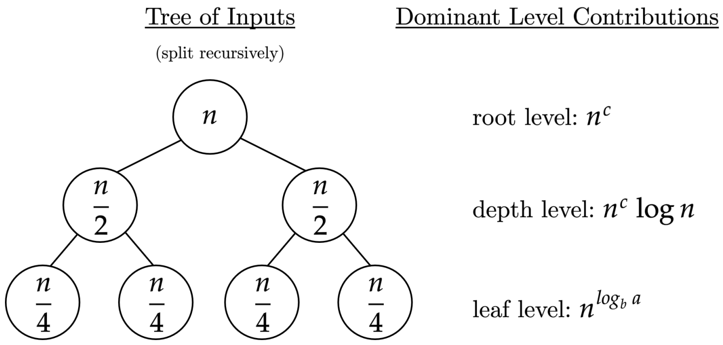
$$T(n) \leq aT\left(\frac{n}{b}\right) + n^c$$

has the following solution:

$$T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \text{ (root dominant)} \\ O(n^{\log_b a}) & \text{if } a > b^c \text{ (leaf dominant)} \\ O(n^c \log n) & \text{if } a = b^c \text{ (equal dominant)} \end{cases}$$

where $a \geq 1$ and $b > 1$ are constants and n^c is the asymptotic positive bound. In this equation, a is the number of subproblems, $\dfrac{n}{b}$ is the size of each subproblem, and n^c is the time/complexity cost of dividing the problem and combining the subproblems.

Recurrence Relations and Master Theorem


Tree of Inputs

(split recursively)

Dominant Level Contributions

Notes: 1. The Master Theorem cannot be applied for unbalanced trees (i.e. if we divide the problem into $T(\frac{n}{b})$ sub-problems of unequal sizes). Example: $T(n) = T(\frac{n}{b}) + T(\frac{2n}{b})$

2. There are $\log_2 n$ levels of recursion in the tree.

3. $T(n)$ and $T(\frac{n}{b})$ on the LHS and RHS must be of a ratio. If LHS=$T(n)$ and RHS=$T(\frac{\sqrt{n}}{2})$ or RHS=$T(n-1)$, the Master Theorem cannot be applied. ($\sqrt{n}$ and $n-1$ are not of the divide and conquer form.)

4. If the time complexity (n^c) is of form c^n, the Master Theorem cannot be applied.

See Logarithm identity 3.

Sorts

Name: **Insertion Sort**

Use: Efficient algorithm for sorting a small number of elements.

Algorithm Sorts elements in place. Input: Array A; Outputs: Sorted Array A. Note: the numbering below assumes that the Array is indexed starting at 1 rather than 0.

Runtime: $O(n^2)$

Pseudocode: 1. Insertion Sort

Algorithm 3 Insertion Sort

```
Runtime: $O(n^2)$
procedure INSERTION SORT(A)
    for $i = 1$ to $A.length - 1$ do:
        key = $A[i]$
        $j = i - 1$                    ▷ Insert $A[i]$ into the sorted sequence $A[i..i-1]$
        while $j > 0$ and $A[j] >$ key do
            $A[j+1] = A[j]$
            $j = j - 1$
end procedure
```

Name: **Merge Sort**

Use: To sort the elements in an unsorted list $C_{unsorted}$.

Description: The process is to split $C_{unsorted}$ into 2 equal lists $A_{unsorted}$ and $B_{unsorted}$, sort both lists recursively, and then combine 2 lists A_{sorted} and B_{sorted} into a single list C_{sorted}.

Runtime:

$$\text{For some constant } c,\ T(n) \leq 2T\left(\frac{n}{2}\right) + cn \text{ when } n > 2, \text{ and } T(2) \leq c. \tag{1}$$

$= O(n \log n)$

Name: **Quick Sort**

Use: Sorting all elements in an array in place. Divide and conquer strategy, randomized or deterministic algorithms possible.

Description: 1. Choose a pivot element x (randomly or deterministically).
2. Compare all array elements to the pivot.
3. Partition the array into 2 subarrays S and L.
4. Arrange elements so all elements $< x$ are in one subarray S, and all elements $> x$ are in the other subarray L.
5. Recursively sort subarrays S and L.

Runtime: 1. Worst case: $O(n^2)$ (in case of a bad partition on a low element)
2. Average case: $\Theta(n \log n)$

Pseudocode: 1.Quick Sort (Deterministic)

Algorithm 4 Partition (Deterministic)

```
Runtime: $O(n)$ (worst-case)
Runtime: $O(\log n)$ (best-case)
procedure Partition($A, start, end$)
    $pivot = A[end]$
    $i = start - 1$
    for $j = start$ to $end - 1$ do
        if $A[j] \leq pivot$ then
            $i = i + 1$
            exchange $A[i]$ with $A[j]$
    exchange $A[i+1]$ with $A[end]$
    return $i + 1$
end procedure
```

Algorithm 5 Quick Sort (Determinisitic)

```
Runtime: $O(n^2)$ (worst-case)
Runtime: $O(n \log n)$ (best-case)
procedure Quick Sort($A, start, end$)
    if $start < end$ then
        $pivot =$ Partition($A, start, end$)
        Quick Sort($A, start, pivot - 1$)
        Quick Sort($A, pivot + 1, end$)
end procedure
```

2. QuickSort (Randomized)

Algorithm 6 Partition (Randomized)

```
Runtime: $O(n)$ (worst-case)
Runtime: $O(\log n)$ (best-case)
procedure Randomized-Partition($A, start, end$)
    $i$ = Random($start, end$)
    exchange $A[end]$ with $A[i]$
    return Partition($A, start, end$)
end procedure
```

Algorithm 7 Quick Sort (Randomized)

```
Runtime: $O(n^2)$ (worst-case)
Runtime: $O(n \log n)$ (best-case)
procedure Randomized-Quick Sort($A, start, end$)
    if $start < end$ then
        $pivot$ = Randomized-Partition($A, start, end$)
        Randomized-Quick Sort($A, start, pivot - 1$)
        Randomized-Quick Sort($A, pivot + 1, end$)
end procedure
```

Name: **Quick Select**

Use: Select the kth smallest element in an array. (Like QuickSort, except we recurse only on half the array.)

Description: 1. Choose a pivot element x (randomly).
2. Compare all array elements to the pivot.
3. Partition the array into 2 subarrays S and L.
4. Arrange elements so all elements $< x$ are in one subarray S, and all elements $> x$ are in the other subarray L, leaving x in the ith position.
5. Apply the following cases:
  1. If $k = i$, return x.
  2. If $k < i$, recurse on the elements to the left of x.
  3. If $k > i$, recurse on the elements to the right of x.

Runtime: $O(n)$

Algorithm 8 Quick Select (Randomized)

```
Runtime: $O(n^2)$ (worst-case)
Runtime: $O(n)$ (expected)
procedure Randomized-Quick Select($A, start, end, k$)
    if $p == r$ then
        return $A[start]$
    $pivot$ = Randomized-Partition($A, start, end$)
    $i = pivot - start + 1$
    if $k == i$ then
        return $A[pivot]$
    else if $k < i$ then
        return Randomized-Quick Select($A, start, pivot - 1, k$)
    else
        return Randomized-Quick Select($A, pivot + 1, end, k - i$)
end procedure
```

Graphs and Trees

Name: **Graphs**

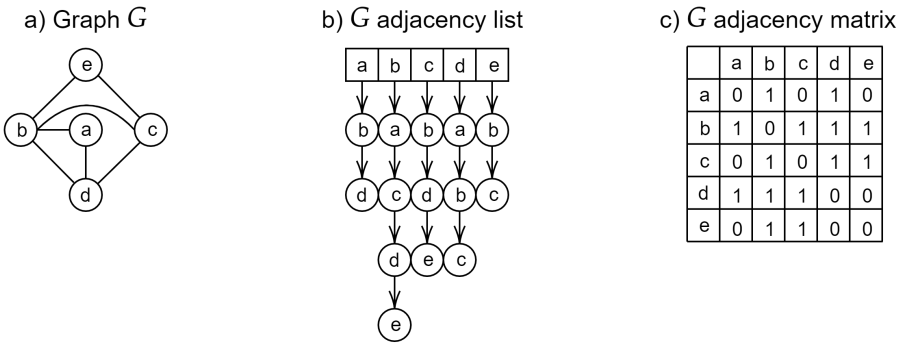
Definitions:

1. Graphs are representations of nodes or vertices connected by edges. Edges may be directed or undirected, weighted or unweighted.
2. Dense graphs are graphs in which there is a high ratio of edges to nodes.
3. Sparse graphs are graphs in which there is a low ratio of edges to nodes.

Representation:

1. Adjacency Matrix: Graphs may be represented as matrices in which all nodes are listed as column headers and row labels, and in which edges between two nodes are marked with 1 if present or 0 if not present. This representation requires a space of $O(n^2)$. It is best used for dense graphs.
2. Adjacency List: Graphs may also be represented as linked lists in which each node is the head of a list, and all nodes sharing an edge with that node are linked to that head node. This representation uses a space of $O(m + n)$, and is asymptotically more efficient than matrix representation in all cases.

Example: See examples of both kinds of representation in the diagram below.



a) Graph G

b) G adjacency list

c) G adjacency matrix

	a	b	c	d	e
a	0	1	0	1	0
b	1	0	1	1	1
c	0	1	0	1	1
d	1	1	1	0	0
e	0	1	1	0	0

Notes:

1. For all graphs G in this document, the runtime complexity assumes an implementation of G as an adjacency list.
2. The set of vertices in a graph G is represented as V. $n = |V|$ in asymptotic notation.
3. The set of edges in a graph G is represented as E. $m = |E|$ in asymptotic notation.

Name: Depth-First Search

Use: Finds all nodes v in a graph G, by exploring all descendant nodes u of a single node first, then exploring all other unvisited nodes in G.

Algorithm 9 Depth-First-Search

```
Runtime: $O(m + n)$
procedure DEPTH-FIRST-SEARCH($G$)
    while there is an unvisited vertex $v \in V$ do
        DF-SINGLE-NODE($v$)
end procedure
```

Algorithm 10 DF Single-Node

```
Runtime: $O(n)$
procedure DF-SINGLE-NODE($v$)
    Mark $v$ as visited.
    while there is an unvisited neighbor $u$ of $v$ do
        DF-SINGLE-NODE($u$)
    Mark $v$ as finished.
end procedure
```

Name: Breadth-First Search

Use: Finds all nodes v in a graph G, by exploring all neighbors u of a single node, then the neighbors of u, etc. until all nodes are discovered.

Algorithm 11 Breadth-First-Search

```
Runtime: $O(m + n)$
procedure BREADTH-FIRST-SEARCH($G, s$)
    Initialize and empty queue $Q$.
    Mark node $s$ as visited.
    ENQUEUE($Q, s$)
    while $Q$ is not empty do
        $x \leftarrow$ DEQUEUE($Q$)
        for each unvisited neighbor $y$ of $x$ do
            Mark $y$ as visited.
            ENQUEUE($Q, y$)
            Mark $x$ as finished.
end procedure
```

Greedy Algorithms

Name: Greedy Algorithms

Definition: For problems where an optimal solution is defined by a series of decisions, make the first decision "short-sightedly" based on what is the best solution now. Then, perform short-sighted solutions recursively on smaller sets.

Name: Progress Argument

Definition: Proves a greedy algorithm by showing that the algorithm progresses **at least** as fast as any other solution. The algorithm never falls behind another algorithm's progress. **Mnemonic:** "greedy stays ahead".

Steps:

1. **Define Your Solution**. Your algorithm will result in a solution X, and some alternative algorithm will result in an optimal solution X^*.
2. **Define Your Measure**. Find a series of measurements (taken at each decision step in the algorithm) to compare your solution X with the optimal solution X^*. Define the series as $m_1(X), m_2(X), ..., m_n(X)$ and $m_1(X^*), m_2(X^*), ..., m_k(X^*)$ such that both series are defined for some values m, n, and k. (Note that $k \neq n$ at this point in the proof, since we cannot yet assume that both series have the same number of elements or that X is optimal.)
3. **Prove Greedy Stays Ahead**. Prove that $m_i(X) \geq m_i(X^*)$ (for maximization problems) or $m_i(X) \leq m_i(X^*)$ (for minimization problems) for all reasonable values of i. Usually, this is proved by induction.
4. **Prove Optimality**. Use the fact "greedy stays ahead" to prove that your algorithm with solution X produces an optimal solution. Typically, this argument is made by contradiction by assuming that the greedy algorithm's solution is not optimal and then by using the "greedy stays ahead" fact to show a contradiction.

Name: Exchange Argument

Definition: Proves a greedy algorithm by iteratively switching an alternative optimal solution into the solution produced by the greedy algorithm without changing the cost of the optimal solution.

Steps:

1. **Define Your Solutions**. Your algorithm will result in a solution X, and some alternative algorithm will result in an optimal solution X^*.
2. **Compare Solutions**. Show that if $X \neq X^*$, they differ in some way. This may mean that a piece of X is not in X^* or that two elements in X are in X^* but in a different order.
3. **Exchange Pieces**. Show how to change X^* into X by exchanging some piece of X^* for some other piece of X. (Typically, you will use the piece identified in the previous step.) Prove that by doing so you did not increase or decrease the cost of X^* and thus have a different optimal solution. (In the rare occasion that there is only one optimal solution, you can directly prove that X^* is decreased and that your algorithm's solution is thus better.)
4. **Iterate**. Reason that you have decreased the number of differences between X and X^* by doing this exchange and that by iterating this process you can turn X^* into X without decreasing its efficiency.

Name: Compression

Definition: Converting **fixed-length strings** of text into **variable-length strings** of text. The purpose of this is to more efficiently store the encoded data.

Name: Huffman Coding Algorithm

Definition: The most efficient algorithm for creating prefix-free, variable-length codes. It returns a binary tree in which each leaf is a character. Characters of lower frequency have longer lengths.

Algorithm 12 Huffman Coding

```
Input: $C$ = a set of characters $c_i$, which each as a frequency.
Output: The root of the tree.
Attribute: $c_i.freq$ is the frequency of $c_i$.
Runtime: $O(n \log n)$
procedure HUFFMAN($C$)
    $n = |C|$
    $Q = C$                                  ▷ $Q$ is a min priority queue, sorted by $c_i.freq$
    for $i$ in range $[1..n-1]$ do
        Initialize a new node $z$
        $z.left = z =$ EXTRACT-MIN($Q$)
        $z.right = y =$ EXTRACT-MIN($Q$)
        $z.freq = x.freq + y.freq$
        INSERT($Q, z$)
    return EXTRACT-MIN($Q$)
end procedure
```

Name: Kruskal's Algorithm

Use: To find a minimum spanning tree (MST) from a connected directed graph.

Description: Starts with no edges. Builds a minimum spanning tree greedily, by adding edges of least weight as long as adding that edge does not create a cycle. Note that after sorting the edges, the nodes of edges e_i and e_{i+1} are not necessarily adjacent.

Algorithm: Kruskal's Algorithm

Algorithm 13 Kruskal's Algorithm

```
Runtime: $O(m+n)$ if $G$ is implemented as an adjacency list.
procedure Kruskal($G, w$)
    Initialize an empty array $A$.
    Sort the edges of $G.E$ into non-decreasing order by weight $w$: $w(e_1) < w(e_2) < \cdots w(e_m)$
    for each edge $(e) \in G.E$ (sorted) do
        if $A \bigcup \{e_i\}$ is acyclic then
            $A = A \bigcup \{e_i\}$
    return $A$
end procedure
```

Name: Prim's Algorithm

Use: To find a minimum spanning tree (MST) from a connected directed graph.

Description: Starts with a single node s. Greedily grows a tree outward from s by adding the node connected by the edge of least weight.

Algorithm: Prim's Algorithm

Algorithm 14 Prim's Algorithm

```
Runtime: $O(m+n)$ if $G$ is implemented as an adjacency list.
procedure Prim($G, s$)
    Initialize an empty set $S$.
    Initialize an empty tree $T$.
    while $S \neq G.V$ do
        Find lightest edge $e$ from $S$ to $G.V - S$. (Example: from $u \in S$ to $v \in G.V - S$)
        $S = S \bigcup \{v\}$
        $T = T \bigcup \{e\}$
    return $T$
end procedure
```

Name: **Dijkstra's Algorithm**

Use: To find the shortest path between node s and all other nodes in a weighted, directed graph.

Notes: 1. Dijkstra's Algorithm provides a solution to the SSSP (Single-Source Shortest Path problem).
2. Dijkstra's Algorithm only works when all weights in the graph are non-negative. (It fails when there are negative edges, or negative cycles. For algorithms that function in the presence of negative edges and can detect negative cycles, see the Bellman-Ford Algorithm.)

Description: Definitions:
$G = (V, E)$: The graph in question.
s: The start node in the graph. Assume s has a path to all other nodes in G.
e: An edge from s to another vertex.
ℓ_e: The length of edge e, or the time (or distance or cost) it takes to traverse e. Let $\ell_e \geq 0$
$\ell(P)$: A path in the graph G. $\ell(P) =$ the sum of the lengths of all edges in P. $\ell(P) = \sum_{k=1}^{e_n \in P} \ell_{e_k}$.
S: Set of nodes u for which we have determined the shortest-path distance.
u: A node in G that has been visited.
$d(u)$: The shortest-path distance between s and some node u.
$V - S$: The edges for which we have not yet discovered the shortest-path distance.
v: Some node in $V - S$.
$e = (u, v)$: An edge e between nodes u and v.
$\min_{e=(u,v):u \in S} d(u)$: The minimum distance between s and u for some $u \in S$. (The min is added because there may be multiple paths between s and u, and we want the shortest path.)
$min_{e=(u,v):u \in S} d(u) + \ell_e$: The minimum distance between s and $v \in V - S$ that passes through some $u \in S$. The path itself consists of nodes $s - u - v$, and the distance is computed by the minimum distance between s and u, added to the length of the single edge $e = (u, v)$.

Algorithm 15 Dijkstra's Algorithm

```
**Runtime:** $O(n^2)$
procedure DIJKSTRA'S ALGORITHM($G, \ell$)
    Let $S$ be the set of explored nodes
    Initially, $S = \{s\}$ and $d(u) = 0$
    for each $u \in S$ do:
        we store a distance $d(u)$
    while $S \neq V$ do:
        Select a node $v \notin S$ with at least one edge from $S$
           for which $d'(v) = min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible.
        Add $v$ to $S$ and define $d(v) = d'(v)$.
end procedure
```

Dynamic Programming

Name: Dynamic Programming

Use: An algorithm technique for solving optimization problems, where subproblems share overlapping subsubproblems and uses **recursion** with **memoization** to reduce runtime complexity. Solutions to the subsubproblems are stored in arrays or lookup tables, to be referenced again when shared subsubproblems are detected. This memoization strategy is more efficient than divide-and-conquer technique in terms of runtime because divide-and-conquer would require duplicate effort to calculate the sub-subproblems.. It is called "programming" not in terms computer programming but in terms of programming solutions into a solutions table.

The purpose of dynamic programming is to reduce solutions that would take exponential time such as $O(2^n)$ if implemented in an iterative form into solutions that run in polynomial time such as $O(n^3), O(n^2)$, or better.

Name: **Weighted Interval Scheduling**

Use: Finding a schedule of jobs that have start time, end time, and a weight (reward) so as to maximize the reward.

Definitions:

1. i and j are 2 specific, distinct requests in the set of jobs $R = \{1..n\}$.
2. Each request i has a start time s_i, a finish time f_i, and a value/weight v_i.
3. $f_1 \leq f_2 \leq \cdots \leq f_n$ is the set of R jobs sorted by non-decreasing order of finish times.
4. A request/job i comes before a request j if $i < j$.
5. $p(j)$ is the largest index where $i < j$ such that jobs i, j 's finish times do not overlap.

Algorithm 16 Compute Optimum (Slow)

```
Runtime: $O(2^n)$
procedure Compute-Opt($j$)
    if $j = 0$ then
        return 0
    else
        return max[$v_j$ + Compute-Opt($p(j)$), Compute-Opt($j - 1$)]
end procedure
```

Algorithm 17 Compute Optimum (Memoized)

```
Runtime: $O(n)$
procedure M-Compute-Opt($j$)
    if $j = 0$ then
        return 0
    else if $M[j] \neq \emptyset$ then
        return $M[j]$
    else
        $M[j]$ = max[$v_j$ + M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$)]
        return $M[j]$
end procedure
```

Algorithm 18 Compute Optimum Solution With Value

```
Runtime: $O(n)$
procedure Find-Solution($j$)
    if $j = 0$ then
        return nothing
    else
        if $v_j + M[p(j)] \geq M[j - 1]$ then
            return $j$ and result of Find-Solution($j$)
        else
            return the result of Find-Solution($j - 1$)
end procedure
```

Name: **Sequence Alignment**

Use: Comparing two sequences of characters (for example, strings of text or DNA sequences) to determine the edit distance between them.

Definitions:

1. X and Y: 2 strings consisting of the characters $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.
2. M: an alignment between 2 strings X and Y.
3. $\delta > 0$: a gap penalty, for 2 characters that represent a "blank", or unaligned place, across 2 strings. Example: In "stop_" and "_tops", "top" is aligned, but the "s" in both strings is unaligned because of the "_" character and would be assigned a gap penalty δ.
4. $\alpha_{x_i y_j}$: a mismatch cost applied for 2 characters $x \in X$ and $y \in Y$ that are in the same position in an alignment of M. The mismatch cost may be of two forms:
   (a) $\alpha_{pq} = 1$: is a penalty of 1, for 2 characters $p \in X$ and $q \in Y$ that are aligned but are different letters.
   (b) $\alpha_{pp} = 0$: is a penalty of 0, for 2 characters $p \in X$ and $p \in Y$ that are aligned and are identical letters.

   Example: In "bop" and "top", the "o" characters are correctly aligned and would receive a penalty of $\alpha_{oo} = 0$. However, the "b" and "t" are aligned to the same place but mismatched and so this position would receive a mismatch cost of $\alpha_{bt} = 1$.
5. $\textsc{opt}(i, j)$: the minimum cost of an alignment M.

Principles:

1. An optimal alignment of M has the following properties:
   (a) It consists of the gap penalty δ and the mismatch cost $\alpha_{x_i y_j}$ of each $(i, j) \in M$.
   (b) The cost of M is the sum of the gap and mismatch penalties, and we desire the minimum-cost alignment.
2. In the optimal alignment of M, at least one of the following is true:
   (a) $(m, n) \in M$
   (b) the mth position of X is not matched
   (c) the nth position of Y is not matched
3. The minimum alignment costs can be resolved by the following recurrence for $i \geq 1$ and $j \geq 1$:
   $\textsc{opt}(i, j) = \min[\alpha_{x_i y_j} + \textsc{opt}(i-1, j-1), \delta + \textsc{opt}(i-1, j), \delta + \textsc{opt}(i, j-1)]$
   Symbols:
   (a) $\alpha_{x_i y_j} + \textsc{opt}(i-1, j-1)$: Cost to replace i with j.
   (b) $\delta + \textsc{opt}(i-1, j)$: Cost to insert j.
   (c) $\delta + \textsc{opt}(i, j-1)$: Cost to delete i.

Algorithm 19 Sequence Alignment

```
**Runtime:** $O(mn)$
procedure ALIGNMENT($X, Y$)
    $A[0...m, 0...n]$                    ▷ Initialize an empty 2-D array to hold the minimum costs
    Initialize $A[i, 0] = i\delta$ for each $i$
    Initialize $A[0, j] = j\delta$ for each $j$
    for each $i \in [1..m]$ do
        for each $j \in [1..n]$ do
            $\textsc{opt}(i, j) = \min[\alpha_{x_i y_j} + \textsc{opt}(i-1, j-1), \delta + \textsc{opt}(i-1, j), \delta + \textsc{opt}(i, j-1)]$
            Store $\textsc{opt}(i, j)$ in $A[i, j]$.
    return $A[m, n]$
end procedure
```

Name: Shortest Paths

Definitions: 1. The shortest path in a weighted directed graph is the path of shortest weight from a source node s to a target node t.

2. A negative edge is an edge of negative weight between two nodes i and j.

3. A negative cycle is a cycle that contains at least one negative edge.

4. A shortest path cannot contain any cycles. Negative cycles will result in an infinite loop that approaches $-\infty$. Positive cycles can be optimized by removing the heaviest edge from the cycle.

5. Some algorithms (such as Dijkstra's) can handle only positive-weight edges. Bellman-Ford algorithm can detect negative edges. Floyd-Warshall's algorithm can detect negative cycles.

6. Relaxing an edge in a dynamic programming algorithm means finding a shorter (lighter) path between two nodes s and t by going through an intermediate node k.

Algorithm 20 Counting Shortest Path

```
Runtime: $O(n^3)$
procedure SHORTEST-PATH($G, s, t$)
    $n$ = number of nodes in $G$
    Initialize array $M[0 \cdots n-1, V]$
    $M[0, t] = 0$
    $M[0, v] = \infty$ for all other $v \in V$
    for $i = 1, \cdots, n-1$ do
        for $v \in V$ in any order do
            $M[i, v] = \text{OPT}(i, v) = \min[\text{OPT}(i-1, v)], \min_{w \in V}[\text{OPT}(i-1, w) + c_{vw}]$
    return $M[n-1, s]$
end procedure
```

Algorithm 21 Relax

```
Runtime: $O(1)$
Input: $u$ = node 1
Input: $v$ = node 2
Input: $w$ = weight function
Attribute $v.d$ = shortest path of $v$
Attribute $v.\pi$ = predecessor node of $v$ in a directed path
procedure RELAX($u, v, w$)
    if $v.d > (u.d + w(u, v))$ then
        $v.d = u.d + w(u, v)$
        $v.\pi = u$
end procedure
```

Name: Bellman-Ford Algorithm

Use: Detecting negative edges in a directed weighted graph.

Algorithm 22 Initialize-Empty-Source

```
Runtime: $O(n)$
Input: $G$ = directed weighted graph
Input: $s$ = source node
Attribute $v.d$ = shortest path of $v$
Attribute $v.\pi$ = predecessor node of $v$ in a directed path
procedure INITIALIZE-EMPTY-SOURCE($G, s$)
    for each node $v \in V$ do
        $v.d = \infty$
        $v.\pi$ = NIL
    $s.d = 0$
end procedure
```

Algorithm 23 Bellman-Ford

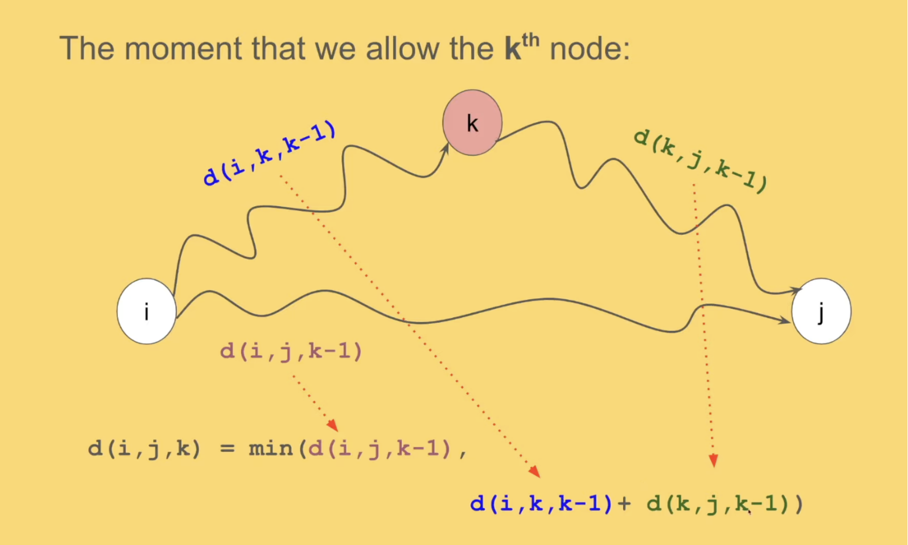
```
Runtime: $O(mn)$
Input: $G$ = directed weighted graph
Input: $w$ = a weight function
Input: $s$ = source node
procedure BELLMAN-FORD($G, w, s$)
    INITIALIZE-EMPTY-SOURCE($G, s$)
    for $i = 1$ to $|V| - 1$ do
        for each edge $(u, v) \in E$ do
            RELAX($u, v, w$)
    for each edge $(u, v) \in E$ do
        if $v.d > (u.d + w(u, v))$ then
            return False
    return True
end procedure
```

Name: **Floyd-Warshall Algorithm**

Use: Finding the shortest path between all nodes in a graph. This solves the APSP (all-pairs shortest path) problem.

Notes: 1. The Floyd-Warshall Algorithm cannot handle negative cycles.
2. However, Floyd-Warshall can detect and report the presence of negative edges and negative cycles.

Concepts: 1. Find the shortest path from i to j, by comparing the weights of paths $i - j$ and $i - k - j$.

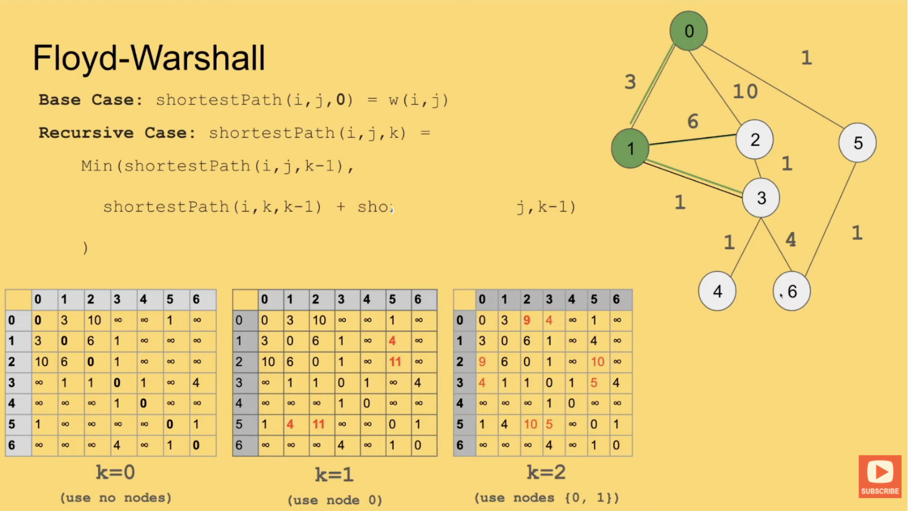


The moment that we allow the **k**th node:

d(i,j,k) = min(d(i,j,k-1),

d(i,k,k-1)+ d(k,j,k-1))

2. Iteratively calculate the shortest path $i - k - j$ for $k = 0, 1, 2$ intermediate nodes respectively for all n nodes.



Floyd-Warshall

```
Base Case: shortestPath(i,j,0) = w(i,j)
Recursive Case: shortestPath(i,j,k) =
      Min(shortestPath(i,j,k-1),
        shortestPath(i,k,k-1) + sho          j,k-1)
     )
```

	0	1	2	3	4	5	6
0	0	3	10	∞	∞	1	∞
1	3	0	6	1	∞	∞	∞
2	10	6	0	1	∞	∞	∞
3	∞	1	1	0	1	∞	4
4	∞	∞	∞	1	0	∞	∞
5	1	∞	∞	∞	∞	0	1
6	∞	∞	∞	4	∞	1	0

k=0

(use no nodes)

	0	1	2	3	4	5	6
0	0	3	10	∞	∞	1	∞
1	3	0	6	1	∞	4	∞
2	10	6	0	1	∞	11	∞
3	∞	1	1	0	1	∞	4
4	∞	∞	∞	1	0	∞	∞
5	1	4	11	∞	∞	0	1
6	∞	∞	∞	4	∞	1	0

k=1

(use node 0)

	0	1	2	3	4	5	6
0	0	3	9	4	∞	1	∞
1	3	0	6	1	∞	4	∞
2	9	6	0	1	∞	10	∞
3	4	1	1	0	1	5	4
4	∞	∞	∞	1	0	∞	∞
5	1	4	10	5	∞	0	1
6	∞	∞	∞	4	∞	1	0

k=2

(use nodes {0, 1})

Images taken from arisaif's YouTube video.

Algorithm 24 Floyd-Warshall

```
    **Runtime:** $O(n^3)$
    **Input**: $W = n \times n$ matrix of edges.
    **Input**: $d_{ij}^{(k)}$ = weight of the shortest path from nodes $i$ to $j$, where all intermediate nodes in the set $\{1, 2, \cdots, k\}$ are included.
    **Output**: $D^{(n)}$ = matrix of shortest paths.
procedure Floyd-Warshall(W)
    $n = W.rows$
    $D^{(n)} = W$
    for $k = 1$ to $n$ do
        Let $D^{(k)} = d_{ij}^{(k)}$ be a new $n \times n$ matrix.
        for $i = 1$ to $n$ do
            for $j = 1$ to $n$ do $d_{ij}^{(k)} = \min\left[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right]$
    return $D^{(n)}$
end procedure
```

Recap

Name: Harmonic Number

Use: The Harmonic Number (H_n or $H(n)$) is the solution to the probability problem: What is the expected number of cards you will guess correctly in the **Guessing Cards with Memory** game? (Game rules: flip a card, guess the card, if you get it wrong, remove the card from the deck and remember all cards previously drawn).

Formula: $\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

Derivation:

$$\begin{aligned} \mathrm{E}[X_i] &= \Pr[X_i = 1] = \frac{1}{n-i+1} \\ \Pr[X] &= \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \frac{1}{n-i+1} \\ &= \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \end{aligned}$$

Bound: $H(n) = \Theta(\log n)$

Name: Maximum/Minimum Notation

Note:

1. The max or min of a set $A = \{1, 2, 3, \cdots, n\}$ can be represented as $\max\{1, 2, 3, \cdots, n\}$.
2. The max or min of functions $f(x)$ applied to a set S is represented as $\max_{x \in S} f(x)$.
3. For functions, the max or min object goes before the subscript in LaTeX inline math mode. The max or min object goes above the subscript in LaTeX display math mode.

Examples: Inline mode: $\min_{e \in E} r(e)$
Display mode:

$$\min_{e \in E} r(e)$$

Appendix

My LaTeX Commands

Commands and Outputs

Output Function	**LaTeX Command Input**
$O(n^2)$	`\BigOh{n^2}`
$\Omega(n^2)$	`\BigOmega{n^2}`
$\Theta(n^2)$	`\BigTheta{n^2}`
$f(n)$	`\FofN`
$f(g(n))$	`\FofGofN`
$f(x)$	`\FofX{f}{x}`
$n_1, n_2, \cdots, n_k$	`\Series{n}`
$f_1 + f_2 + \cdots + f_n$	`\SeriesSum{f}{n}`
$n_1 + n_2 + \cdots + n_k$	`\SeriesSumK{n}`
$a_0n^0 + a_1n^1 + a_2n^2 + \cdots + a_kn^k$	`\PolynomialSum{a}{k}`
$\max\{5, 2\} = 5$	`\mathmax{5, 2} = 5`
$\lfloor 5.2 \rfloor = 5$	`\floor{5.2} = 5`
$\lceil 5.2 \rceil = 6$	`\ceiling{5.2} = 6`
`my code`	`\Code{my code }`
`i`	`\CodeI`
`n`	`\CodeN`
red text	`\red{red text}`
blue text	`\blue{blue text}`
violet text	`\violet{violet text}`
$\{1, 2, 3\}$	`\set{1, 2, 3}`
$\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$	`\harmonic`

LaTeX Command Definitions

```
\newcommand{\BigOh}[1]{O(#1)}
\newcommand{\BigOmega}[1]{\Omega(#1)}
\newcommand{\BigTheta}[1]{\Theta(#1)}

\newcommand{\FofX}[2]{#1(#2)}
\newcommand{\FofN}{f(n)}
\newcommand{\GofN}{g(n)}
\newcommand{\FofGofN}{f(g(n))}
\newcommand{\Series}[1]{#1_1, #1_2, \cdots , #1_k}
\newcommand{\SeriesSumK}[1]{#1_1 + #1_2 + \cdots + #1_k}
\newcommand{\SeriesSum}[2]{#1_1 + #1_2 + \cdots + #1_#2}
\newcommand{\PolynomialSum}[2]{#1_0n^0 + #1_1n^1 + #1_2n^2 + \cdots + #1_#2n^{#2}}
\newcommand{\mathmax}[1]{\text{max}\{#1\}}
\newcommand{\floor}[1]{\lfloor #1 \rfloor}
\newcommand{\ceiling}[1]{\lceil #1 \rceil}
\newcommand{\Code}[1]{\texttt{#1} }
\newcommand{\CodeI}{\texttt{i} }
\newcommand{\CodeN}{\texttt{n} }
\newcommand{\red}[1]{\textcolor{red}{#1}}
\newcommand{\blue}[1]{\textcolor{blue}{#1}}
\newcommand{\violet}[1]{\textcolor{violet}{#1}}
\newcommand{\set}[1]{\{#1\}}
\newcommand{\harmonic}{\sum_{i=1}^n \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}}
\renewcommand{\Pr}[1]{{\text{Pr}\left[ #1 \right]}}
\newcommand{\E}[1]{{\text{E}\left[ #1 \right]}}
\newcommand{\Var}[1]{{\text{Var}\left( #1 \right)}}
```

Index