## Properties of Summations

$$\sum_{k=0}^{n} a_1 x^k = \frac{a_1(1-r^n)}{1-r}$$

$$\sum_{k=1}^{n} ca_k + b_k = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$$

**Arithmetic** $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

**Geometric**

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 = \frac{x^{n+1}-1}{x-1}$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}, |x| < 1$$

**Harmonic**

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

## Asymptotic Notation

$O(g) = \{f \mid \exists c, n_0 > 0$
$\forall n \ge n_0, 0 \le f(n) \le c \cdot g(n)\}$

$\Omega(g) = \{f \mid \exists c, n_0 > 0$
$\forall n \ge n_0, 0 \le c \cdot g(n) \le f(n)\}$

$(n) \in \Theta(g) \Leftrightarrow \exists c\_1, c\_2, n\_0$
$\forall n \ge n\_0,$
$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$

$f \in O(g) \Rightarrow g + f \in \Theta(g)$
$k \cdot n^a \in \Theta(n^a), \quad k > 0$
$f \in O(g) \Leftrightarrow g \in \Omega(f)$
$f \in \Theta(g) \Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$

## Logarithm Laws

$\log_b b = 1 \qquad \log_b(1) = 0$
$\log(ab) = \log(a) + \log(b)$
$b^{\log_b y} = y$
$\log\left(\frac{1}{a}\right) = -\log(a)$
$\log_b(M^k) = k \log_b M$
$\log_b(b^k) = k$
$x = \log_b(a) \to b^x = a$
$\log x < x \quad \forall x > 0$

## Recurrences - Master Method

**Case 1**: f *Asymptotically Smaller*
$T(n) \in \Theta(n^{\lg_b(a)})$
$\Rightarrow f(n) \in O(n^{\lg_b a - \epsilon}), \epsilon > 0$

**Case 2:** f *Asymptotically Same*
$T(n) \in \Theta(n^{\log_b a} \lg n)$
$\Rightarrow f(n) \in \Theta(n^{\log_b a})$

**Case 3:** f *Asymptotically Larger*
$T(n) \in \Theta(f(n))$
$\Rightarrow f(n) \in \Theta(n^{\lg_b a + \epsilon})$

If $af\left(\frac{n}{b}\right) \le cf(n)\ for\ c < 1$

**Case2+:** f larger, not asymptoti $f \in (n^{\lg_b a} \lg^k n) \to \Theta(n^{\lg_b a} \lg^{k+1} n)$

## Substitution Method

*Guess solution, prove by induction*

$$T(n) = \begin{cases} 2, & n = 2 \\ 2T\left(\frac{n}{2}\right) + n, & n = 2^k\ \forall\ k > 1 \end{cases}$$

=> $T(n) = n \lg n, n = 2^k, k \ge 1$

Assume $T\left(\frac{n}{2}\right) = \frac{n}{2} \lg\left(\frac{n}{2}\right)$

Prove for $n = 2^k$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$= 2\left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) + n$$
$$= n(\lg n - \lg 2) + 2 = n \lg n$$

## Graph Data Structures

**Adjacency List:** + sparse graphs
**Space** $\Theta(V + \sum_{v \in G.V} outDeg(v)) = \Theta(V + E)$
**Time isAdjacentTo** $O(V)$
**Time hasEdge** $\Theta(1)$

**Adjacency Matrix**: + dense graphs
**Space** $\Theta(V^2)$
**Time isAdjacentTo** $O(1)$

## Undirected Graphs

Must not have any self-loops.

$|E| \le |V|^2$

**Path**: Length $k$ from $v_0 \to v_k$ $\langle v_0, \cdots, v_k \rangle, (v_{i-1}, v_i) \in G.E$ $\forall i \in \{1, \cdots, k\}$

$u$ is **reachable** from $v$ if ∃ path from $u$ to $v$

**Simple Path** if all vertices distinct
**Path is Cycle** if $v_0 = v_k \wedge k \ge 1$
**Simple Cycle** If all v and e unique
**Connected** if every vertex is reachable from all other vertices

$$|V| - 1 \le |E| \le \frac{n(n-1)}{2}$$

**Forest** if it is acyclic

$$0 \le |E| \le |V| - 1$$

**Tree** if it is a forest with one connected component

$$|E| = |V| - 1$$

## Directed Graphs

**Strongly Connected** Every two vertices reachable

$G' = (V', E')$ **Subgraph** if $V' \subseteq V, E' \subseteq E$

$G'$ **Spanning Subgraph** if $V' = V$

## Breadth-First Search $O(V + E)$

*Unwt Single-Source Shortest Path*

```
bfs(G, v)
  for u in G.V – {v}:
    u.d = ∞; u.color=WHITE
  v.d = 0; v.color = GREY
  Q = {v} // Uses a Queue.
  while Q.size != 0
    curr = Q.dequeue()
    for u in G.adj[curr]
      if u.color == WHITE
        u.d = curr.d + 1
        u.color = GREY
        Q.enqueue(u)
     curr.color = BLACK // done
```

$$\Theta(V + 1 + V \times 1 + deg(V) \times 1) = \Theta(V + E)$$

## Depth-First Search O(V+E)

Not guaranteed shortest path.

```
dfs(G)
  t=0
  for u in G.V: {u.c = WH; u.pi=-1}
  for u in G.V:
    if u.color = white {visit(G, u)}
visit(G, v)
  v.color=grey; t++;  v.disc=t
  for u in G.adj[v]
    if u.color = white:
      u.pi = v
      visit(G, u)}
    v.color=black; t++; v.fin = t
```

## Directed Acyclic Graphs

**TopoSort**: Linear ordering of $V$ s.t. if $(u, v) \in E$ then u before v.
All vertices with directed edges to v appear before v.

Topo-sort(G) $O(V + E)$
  Init empty linked list of vertices
  Call DFS(G)
  As each vertex finished, add to front of list
  Return list of vertices in descending order of DFS finishing time.

## Prim's Algorithm

*Finds MST of weighted, undirected graph using least-weight edges.*

Base Case: T if tree is spanning
Recursive $T \cup \{(u, v)\}$ where (u,v) is the least-weight edge leaving T

$O(V \times T_{ext-min} + E \times T_{dec-key})$

**Array** $O(V^2)$
**Fibonacci Heap** $O(E + V \lg V)$
**Binary Heap** $O(E + \lg V)$

```
mst-prim(G, w, r)
  for each v in V {v.k = ∞, v.π=-1}
  r.key = 0
  Q = G.V
  while Q.size() != 0
    u = Q.remove-min()
    for each v in G.adj[v]
      if v in Q and q(u,v) < v.key
        v.key = w(u,v)
        // decrease the key
        v.π = u
```

## Kruskal's Algorithm Greedy $E \lg E$

*Finds minimum spanning forest. If graph is connected, finds MST.*

Create graph with |V| forests.
Add least-weighted edge connecting any two forests together. Terminate when T is connected.
Uses a **disjoint forest** data structure.
Make-set O(1)
Find-set O(1) normally, O(lg n)
Union(x, y) runs in almost O(1)

mst-kruskal(G, w) $\Theta(E \lg E)$

```
  T = {}
  For v ∈ G.V { make-set(v) }
  Sort G.E in non-decreasing
    order by weight
  for (u, v) ∈ G.E (sorted)
    if find-set(u) ≠ find-set(v)
      T = T ∪ {(u, v)}
      union(u,v)
  return T
```

## Dijkstra's Algorithm Greedy

*Single-Source Shortest Path*

Finds shortest path tree of weighted graph G with no negative weight edges.

$\Theta(V \times T_{e-min} + E \times T_{dec-key})$

**Array** $O(V^2 + E * 1) = O(V^2)$
**Binary Heap** $O(V \lg V + E \lg V)$
**Fibonacci Heap** $O(E + V \lg V)$

Not guaranteed for graphs with negative weight edges, as it will not re-explore paths (via newly explored vertices).

Let A=source, B, C.
e(A,B)=5, e(A,C)=6, e(C,B)=-3
(1) Set d(A)=0 as source
(2) Set d(B)=5 d(B) > d(A) + w(A,B)
(3) Set d(C)=6 d(C) > d(A) + w(A,B)
->no check d(B)=d(C)+w(C,B)=3

**Dijkstra(G, weight, source):**

```
  Init-single-source(G, s)
  Visited = {}; Q = G.V;
  While Q.size != 0
    u = Q.extract-min()
    visited.add(u)
    for v ∈ G.adj[u] relax(u, v, w)
```

**init-single-source(G, s)**

```
  for v ∈ G.V {v.d=∞, v.π=NIL}
  s.d = 0
```

**relax(u, v, w):**

```
  if v.d > u.d + w(u, v)
    v.d = u.d + w(u,v); v.π = u
```

## Bellman Ford

*Single-Source Shortest Path in Weighted Directed Graph*

```
Bellman-Ford(G, w, s)
  init-single-source(G, s)
  // Relax each edge |V|-1 times
  For (u, v) ∈ G.E
    If v.d > u.d + w(u,v)
      return False
      // Negative wt cycle
  return True
```

## Floyd Warshall [Dynamic]

*All-Pairs Shortest Paths* $O(|V|^3)$

Number vertices from 0..|V|-1
Let $sp(i, j, k)$ be the shortest path from $i \to j$ using k

Floyd-warshall
 N = W.rows  $D^{(0)} = W$
 For k = 1..n
  Let $D^{(k)} = (d_{ij}^{(k)})$ be nxn mat
  For i=1..n { for j=1..n {
   $d_{ij}^{(k)} = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}$
  }}
return $D^{(n)}$ – $T(n) \in \Theta(n^3)$

## Greedy Problems

Greedy problems exist when problem has optimal substructure and greedy choice property.

*Solve problems by making greedy (locally optimal) choice and solving only chosen sub-problem.*

**Greedy Choice Pr** Given problem, know which sub-problem's solution will yield optimal solution without having to solve all other sub-problems

## Dynamic Programming

*Efficiency from avoiding re-computing sub-problems*

May apply to problems with optimal substructure in which optimal solution can express as optimal solutions to subproblems.

**Bottom-Up** Solve base case first

**Memoisation** Solve top-down like recursive algo – worse constant fs

$LCS(\langle\rangle, S_2) = LCS(S_1, \langle\rangle) = 0$

LCS($S_1$.X, $S_2$.X)= LCS($S_1$, $S_2$)+1

LCS($S_1$.X, $S_2$.Y)=MAX(

LCS($S_1$, $S_2$.Y), LCS($S_1$.X, $S_2$)), $X \neq Y$

## Amortised - Aggregate Method

Argue that a series of $n$ operations is completed in $T(n)$ – each operation has amortised cost $\frac{T(n)}{n}$.

## Amortised – Accounting Method

*Focus on data structure operation*

1 Calculate the actual cost $c_i$ of each type of operation

2 Assign an amortised cost $\hat{c_i}$ to each operation

**For any sequence of operations, amortised cost must be an upper bound on actual cost**

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Credit stored is difference between amortised cost and actual cost.

## Potential Method

1 Determine the cost of each operation

2 Define a potential function on data structure

Amortised cost of an operation:

$$\hat{c_i} = c_i + \left(\Phi(D_i) - \Phi(D_{i-1})\right)$$

**For any sequence of operations, amortised cost must be an upper bound on actual cost.**

Amortised cost = sum of ci

Obliged to show $\Phi(D_i) \geq \Phi(D_0)$

True if $\Phi(D_0) = 0, \Phi(D_i) \geq 0$

Push / Pop / Multipop

Actual: 1, 1, min(|S|, k)

ΔΦ 1, -1, -k'

Amortised: 2, 0, 0

## Complexity Theory - Polynomial

P: Class of problems that can be solved in polynomial time on a serial random-access machine.

Same as the class of problems that can be solved in polynomial time on an abstract Turing machine, and the class of problems that can be solved in polynomial time on a parallel compute where the number of processors grow polynomially with input size.

---

Closed under addition, multiplication, and composition:

**Addition**: Run one polynomial time algorithm after another is still $O(n^k)$

**Multiplication**: Run a polynomial time algorithm a polynomial number of times is still polynomial time

**Composition**: Feeding the output of a polynomial time algorithm (which is at most polynomial in size) to another polynomial time algorithm is still polynomial.

Alternatively, run a polynomial-time algorithm on an input of polynomial size.

## Complexity Theory – NP

**Non-Deterministically Polynomial**

Set of concrete problems for which a solution (certificate) can be checked/verified in polynomial time.

Problems for which we can't even verify a solution in polynomial time are unlikely to have a polynomial time solution

---

Trivially, we know that $P \subset NP$

**NP Hard** A concrete decision problem $B$ is NP-hard when every problem $A \in NP$ is polynomial-time reducible to $B$

**NP Complete** It is NP-H and its solution can be verified in polynomial time

## Complexity Theory – Classes

Focus on concrete decision probs:

**Decision** Output 1 or 0 as sol'n

Optimisation problem usually have closely related decision problems.

If the optimisation problem is easy, the related decision problem is also easy.

**Abstract**: Problem that takes any input and maps to a solution.

Binary relation as there may be multiple solutions for a given problem instance.

**Concrete**: Problem that has set of binary strings as input.

**Encodings**: Translate abstract problems to concrete problems

Reductions

Let X be a problem. If you can show that a known NP-Hard problem is reducible to X then X is NPH.

**Reduce HAM-CYCLE to TSP**

**Hamiltonian Cycle:** Simple path through unweighted graph containing very vertex, starting, and ending at the same vertex.

**Use TSP to solve HAM** by set weight to 0 if edge exists, else 1.

*For a problem to be NP Hard, we must show that every problem in NP is polynomial-time reducible to B.*

*To do this, we can show there exists a mapping between the inputs of a known NP-Hard problem and the problem X (Reduce Known NPH to X).*

*For a problem to be NP-Complete, we must show that it is NP Hard, and that a certificate (solution) to the problem can be verified in polynomial time.*

- **Master Method** – added the final Θ(…) answer.
- **Dynamic Programming** – Check that using max or min in the right way – trying to max/min a particular value.

Pseudocode – populate the base case first before computing the further cases

Return a value

*Joe biden*