BFS O(V+E)

Procedure bfs(G,s)
Input: Graph G=(V,E), directed or undirected; vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.

```
for all u ∈ V:
    dist(u) = ∞          } initialization
dist(s) = 0
Q = [s]  # Q is a queue containing s
while Q is not empty:
    u = eject(Q)
    for all edges (u,v) ∈ E:
        if dist(v) = ∞:
            inject(Q,v)  ← dist(v) = dist(u)+1
```

Asymptotic linear function

$\frac{T(n)}{n} \to \begin{cases} \infty & T \in \omega(n), T > n \\ \text{constant} > 0 & T \in \theta(n), T = n \\ 0 & T \in o(n), T < n \end{cases}$

$T \in O(f)$ $T \leq f$ | $T \in \theta$ $T = f$ | $T \in \Omega(f)$ $T \geq f$

Master theorem

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

a = # of branching

b = is reduced size of input at each step

d = exponent of complexity of the merge/combine step

DFS O(V+E)

```
Procedure dfs(G)
for all v ∈ V:
    visited(v) = false
for all v ∈ V:
    if not visited: explore(v)
```

```
Procedure explore(G,v):
Input: G=(V,E) is a graph; v ∈ V
Output: visited(u) is a set to true for all nodes u reachable from v
visited(v) = true
previsit(v)
for each edge(v,u) ∈ E:
    if not visited(u): explore(u)
postvisit(v)
```

Sparse Graph:
|E| is very close to |V|
Dense Graph:
|E| is very close to $O(|V^2|)$
Complete graph:
$|E| = \frac{n(n-1)}{2}$

Dijkstra shortest path $O(V^2)$ use vector, $O((V+E)\log V)$ use a queue

Input: Graph G=(V,E), directed or undirected; positive edge lengths $\{l_e : e \in E\}$
vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u

```
for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0
H = make queue(V)  # Vector or queue
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
        if dist(v) > dist(u) + l(u,v):
            dist(v) = dist(u) + l(u,v)
            prev(v) = u
            decreasekey(H,v)
```

Divide and Conquer

```
function merge(x[1...k], y[1...l])
if k = 0:  return y[1...l]
if l = 0:  return x[1...k]
if x[1] ≤ y[1]:
    return x[1] ∘ merge(x[2...k], y[1...l])
else:
    return y[1] ∘ merge(x[1...k], y[2...l])
```

```
function iterative-mergesort(a[1...n])
Input: elements a_1, a_2, ..., a_n to be sorted

Q = [ ] (empty queue)
for i = 1 to n:
    inject(Q, [a_i])
while |Q| > 1:
    inject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
```

Chord: An edge that connects a pair of vertices in a cycle, but is not actually in a cycle.
Induced cycles contains no chords forms an induced subgraph
Induced subgraphs needs to contain all the edges, except the ones connected to a removed node
Simple cycles:
A cycle that is broken down to its most simple form
A cycle in a graph that has no repeat vertexs

Edge Disjoint Path
Disjoint sub graphs
Do not share common vertices or edges

It can be Edge or vertices disjoint path

Bellman Ford O(V·E)

Resulting path is not negative

```
procedure shortest-paths(G, l, s)
Input:  Directed graph G = (V, E);
        edge lengths {l_e : e ∈ E} with no negative cycles;
        vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
    for all e ∈ E:
        update(e)
```

```
procedure update((u,v) ∈ E)
dist(v) = min{dist(v), dist(u) + l(u,v)}
```

```
procedure dag-shortest-paths(G, l, s)
Input:  Dag G = (V, E);
        edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
    for all edges (u,v) ∈ E:
        update(u,v)
```

O(?)

Adjacency list look up time O(|V|) worst case for dense graph

Finds numbers using divide and conquer

```
def random_number(arr, k):
    if len(arr) == 1:
        return arr[0]
    if len(arr) == 2:
        if k == 0:
            return min(arr)
        else:
            return max(arr)
    if len(arr) == 0:
        print("Error: empty array")
        return 0
    rand = arr[random.randint(0, len(arr)-1)]
    s1 = []
    s2 = []
    s3 = []
    for i in range(len(arr)):
        if arr[i] < rand:
            s1.append(arr[i])
        elif arr[i] == rand:
            s2.append(arr[i])
        else:
            s3.append(arr[i])

    if k < len(s1):
        return random_number(s1, k)
    elif k < len(s1) + len(s2):
        return rand
    else:
        return random_number(s3, k - len(s1) - len(s2))
```

Spanning tree:
A subgraph that contains all the vertices and only remove edges, such that the graph is a tree.

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

Dijkstra's algorithm

Binary heap implementation

preferable use when $|E| < \frac{|V|^2}{\log(|V|)}$

Binary Heap $O((|V|+|E|)\log(|V|))$

```
function makeheap(S)
h = empty array of size |S|
for x ∈ S:
    h(|h|+1) = x
for i = |S| downto 1:
    siftdown(h, h(i), i)
return h

procedure bubbleup(h, x, i)
(place element x in position i of h, and let it bubble up)
p = ⌈i/2⌉
while i ≠ 1 and key(h(p)) > key(x):
    h(i) = h(p);  i = p;  p = ⌈i/2⌉
h(i) = x

procedure siftdown(h, x, i)
(place element x in position i of h, and let it sift down)
c = minchild(h, i)
while c ≠ 0 and key(h(c)) < key(x):
    h(i) = h(c);  i = c;  c = minchild(h, i)
h(i) = x
```

topological sort

```
function linearization(G)
Input: directed acyclic graph (DAG) G = (V,E)
Output: linearization of G

for all v in V:                          // O(|V|)
   in-degree(v) = 0

run DFS and for each edge (u,v):         // O(|V|+|E|)
   increment in-degree(v) by 1

S = []  (empty queue of source vertices)

for all v in V:                          // O(|V|)
   if in-degree(v) is 0:
      add v to S

while S is not empty:                    // O(|V|+|E|)
   u = de-queue(S)
   print u
   for each edge (u,v):
      if in-degree(v) is 0:
         add v to S
```

```
DAG Properties:
        - a directed graph has a cycle if and only if its
                DFS reveals a back edge.
        - In a DAG, every edge leads to a vertex with a
                lower post number.
        - Every DAG has at least one source and one sink.
        - Every graph is a DAG of its SCC's
```

```
Kosarahu's SCC Algorithm

1.) Find the Transpose of graph G
2.) perform DFS on GT defining clock of each vertex
3.) perform DFS on G in order of decreasing post # from 1.)
        - each invocation finds one SCC
```

Algorithm 1: Constructing an Induced Subgraph

Data: $G = (V, E)$ - the input graph with nodes V and edges E, $S \subseteq V$ - the inducing set of nodes
Result: $G_S = (S, E_S)$ - the subgraph of G induced by S

```
E_S ← ∅
for u ∈ S do
    for v ∈ ADJACENT-NODES(u) do
        E_S ← E_S ∪ {(u, v)}
    end
end
G_S ← (S, E_S)
return G_S
```

Algorithm 2: Checking if a Subgraph Is Induced

Data: $G_1 = (V_1, E_1)$ - the input graph with nodes V_1 and edges E_1, $G_2 = (V_2, E_2)$ - a subgraph of G_1 ($V_2 \subseteq V_1$ and $E_2 \subseteq E_1$)
Result: *true*, if G_2 is an induced subgraph; *false*, otherwise

```
for u ∈ V_2 do
    // check all the nodes adjacent to u in G_1
    for v ∈ ADJACENT-NODES(u) do
        if (u, v) ∉ E_2 then
            return false
        end
    end
end
return true
```

Algorithm 3: Checking if a Graph Is an Induced Subgraph of Another Graph

Data: $G_1 = (V_1, E_1)$ - the input graph with nodes V_1 and edges E_1, $G_2 = (V_2, E_2)$ - a graph of nodes in V_1 (so $V_2 \subseteq V_1$, but not necessarily $E_2 \subseteq E_1$))
Result: *true*, if $G2$ is an induced subgraph of G_1; *false*, otherwise

```
// First check if G_2 is a subgraph of G_1
for (u, v) ∈ E_2 do
    if (u, v) ∉ E_1 then
        return false
    end
end
// If yes, check if it's induced by V_2
for u ∈ V_2 do
    // check all the nodes adjacent to u in G_1
    for v ∈ ADJACENT-NODES(u) do
        if (u, v) ∉ E_2 then
            return false
        end
    end
end
return true
```

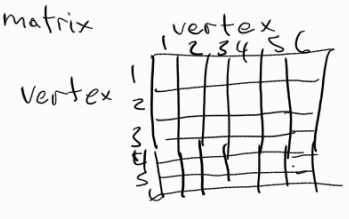

SUMS — COMMON SUMS

Prove using induction…!

Common Sums. Prove these sums by induction on n. Please do it!

1. $\sum_{i=k}^{n} 1 = n+1-k$
2. $\sum_{i=1}^{n} f(x) = nf(x)$
3. $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$ $(r \neq 1)$
4. $\sum_{i=1}^{n} i = n(n+1)/2$
5. $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$
6. $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$
7. $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$
8. $\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$
9. $\sum_{i=1}^{n} \log i = \log n!$

Adjacency matrix



$G(V, E) = (\{\text{set of vertices}\}, \{\text{set of edges}\})$