CS170 Review Session

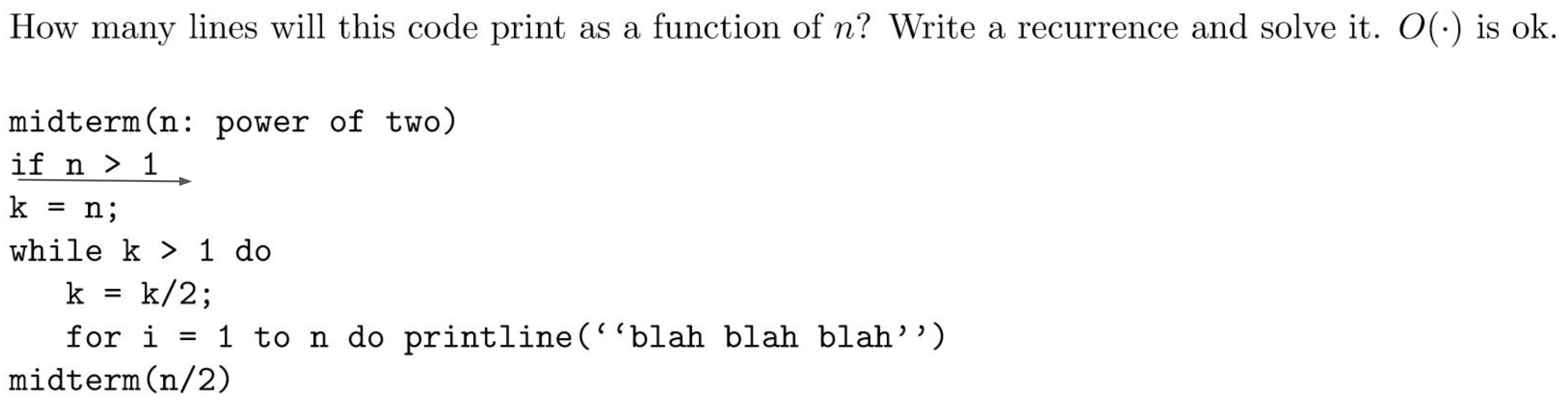
7:00-9:00, Hearst Annex A1

Ray and Chris

Plan for the night

- Practice problems

Recurrence Analysis (Sp 15, MT1)

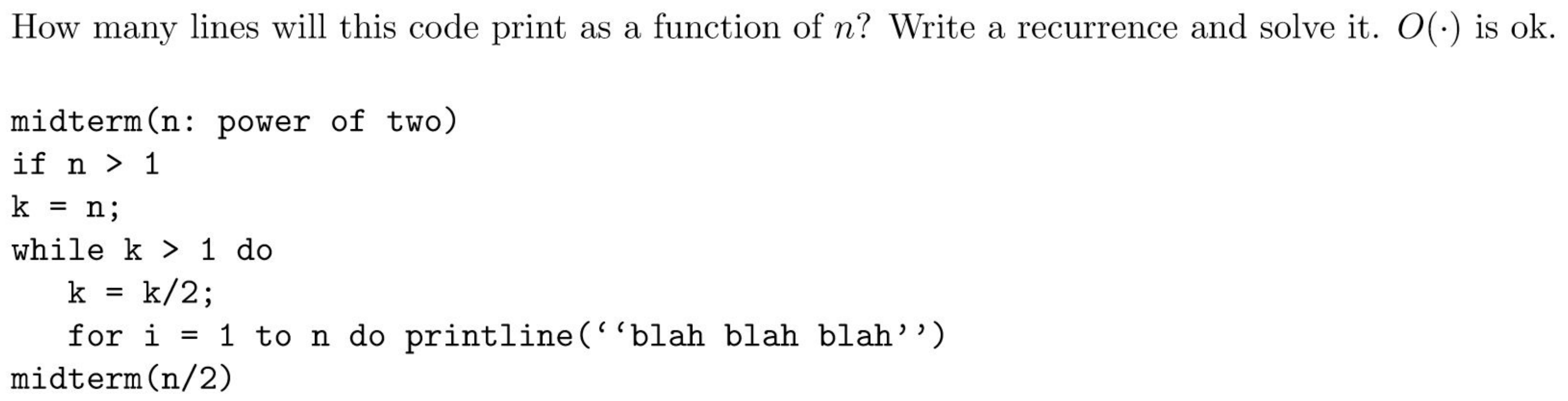


How many lines will this code print as a function of n? Write a recurrence and solve it. $O(\cdot)$ is ok.

```
midterm(n: power of two)
if n > 1
k = n;
while k > 1 do
   k = k/2;
   for i = 1 to n do println(‘‘blah blah blah’’)
midterm(n/2)
```

Indentation error: Everything should be encapsulated by the if statement

Recurrence Analysis (Sp 15, MT1)



How many lines will this code print as a function of n? Write a recurrence and solve it. $O(\cdot)$ is ok.

```
midterm(n: power of two)
if n > 1
k = n;
while k > 1 do
   k = k/2;
   for i = 1 to n do println(``blah blah blah'')
midterm(n/2)
```

The while loop will print n * log(k) times. k=n. So the recurrence relation is:

T(n) = T(n/2) + O(n * log(n)). This is O(n * log(n)) (after expansion and bounding above and below). The bounding above and below is long, so not included here. Justification is available on a follow-up on Piazza’s MT1 Review thread.

Recurrence Relations (Sp13 Mt1)

You can solve a problem three different ways.

- Algorithm A has recurrence $T(n) = 3T(n/2) + n^2$.
- Algorithm B $T(n) = 8T(n/3) + n$.
- Algorithm C $T(n) = T(n-2) + n$.

Which one is the fastest in $O(\cdot)$? The slowest? Explain.

Recurrence Relations (Sp13 Mt1)

Algorithm A: $T(n) = 3 * T(n/2) + n^2$

Using Master's Theorem: $\log_2 3 < 2$. Thus $T(n) = O(n^2)$

Algorithm B: $T(n) = 8 * T(n/3) + n$

Using Master's Theorem: $\log_3 8 > 1$. Thus $T(n) = O(n^{log\_3\ (8)})$

Recurrence Relations (Sp13 Mt1)

Algorithm C: $T(n) = T(n - 2) + n$

Recurrence Relations (Sp13 Mt1)

Algorithm C: $T(n) = T(n - 2) + n$

$$\begin{tikzcd}
T(n) \arrow[d] & \text{Work: } O(n) \\
T(n - 2) \arrow[d] & O(n - 2) \\
T(n - 4) & O(n - 4)
\end{tikzcd}$$

Recurrence Relations (Sp13 Mt1)

Algorithm C: $T(n) = T(n - 2) + n$

$$\begin{tikzcd}
T(n) \arrow[d] & \text{Work: } O(n) \\
T(n-2) \arrow[d] & O(n-2) \\
T(n-4) & O(n-4)
\end{tikzcd}$$

There are n/2 levels. Summing up the work at every level:

$\sum_{i=0}^{n/2}(n-2i) \leq n * n = n^2$

$\sum_{i=0}^{n/2}(n-2i) \geq \overline{\sum_{i=0}^{n/4} n/2} = n^2/8$

$\Theta(n^2)$

Asymptotically, algorithm B is fastest at $\Theta(n)$ and algorithms A and B are slowest at $\Theta(n^2)$

Some True/False (Sp 15, MT1)

$$n^{\log n} = O(2^n)$$

If $f(n)$ and $g(n)$ are functions from the positive integers to the positive integers, then either $f(n) = O(g(n))$ or $g(n) = O(f(n))$ or both.

Some True/False (Sp 15, MT1)

$$n^{\log n} = O(2^n)$$

True

$$n^{\log n} = (2^{\log n})^{\log n} = 2^{(\log n)^2}$$

If $f(n)$ and $g(n)$ are functions from the positive integers to the positive integers, then either $f(n) = O(g(n))$ or $g(n) = O(f(n))$ or both.

False, consider f(n) = (n^3) when n is odd, (n) when n is even. And g(n) = n^2

Topological Sort

Which of the following will give you a guaranteed valid topological sort for a DAG?

A) Order the nodes by prenumber
B) Order the nodes by reverse postnumber
C) Order the nodes by in-degree
D) None of the above

Topological Sort

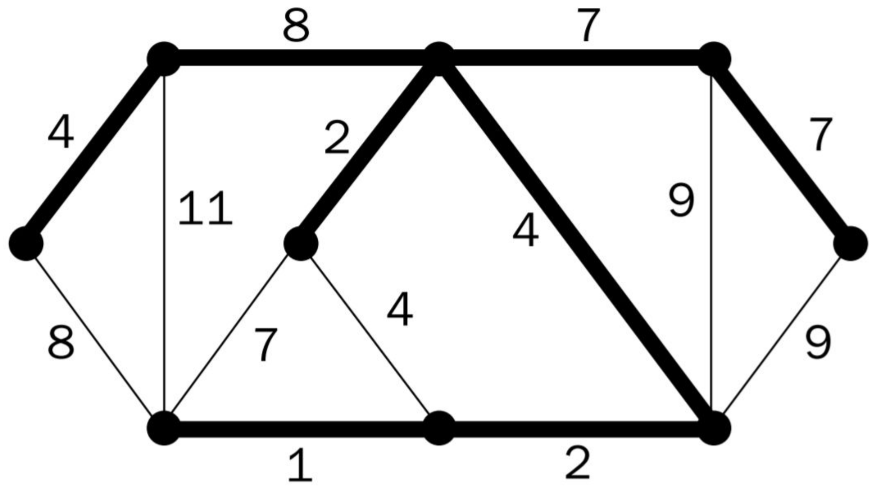
Which of the following will give you a valid topological sort for a DAG?

A) Order the nodes by prenumber
B) Order the nodes by reverse postnumber
C) Order the nodes by in-degree
D) None of the above

B. Can you think of a counter-example for A and C?

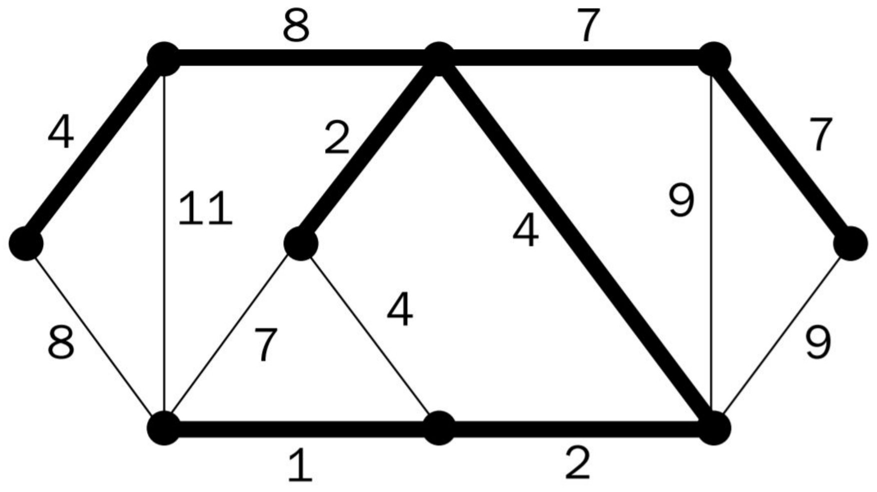
Kruskal's MST (Fa 00, MT1)

The bolded edges are the MST edges as returned by Kruskal's. Which edge must have been added last?



Kruskal’s MST (Fa 00, MT1)

The bolded edges are the MST edges as returned by Kruskal’s. Which edge must have been added last?



Kruskal’s adds edges from smallest to biggest, so the answer is 8.

MST vs. Shortest Path Tree (Fa 07, MT1)

True or false: We can find a connected undirected weighted graph G and a vertex V such that the minimum spanning tree and the shortest path tree rooted at V are disjoint. (Assume that the graph has a unique MST)

MST vs. Shortest Path Tree (Fa 07, MT1)

True or false: We can find a connected undirected weighted graph G and a vertex V such that the minimum spanning tree and the shortest path tree rooted at V are disjoint. (Assume that the graph has a unique MST)

False. For any graph, if we were to run Prim's algorithm starting at V, we would add the shortest edge touching V to the MST. Similarly, if we run Dijkstra's from V, we would add the shortest edge touching V to the shortest path tree. So in all cases, they would share at least one edge.

Dijkstra Modification (Fa 07, MT1)

We are given a directed graph with positive edge weights. We wish to find a shortest path from s to t, and among all shortest paths, we want the one in which the longest edge is as short as possible. How would you modify Dijkstra’s algorithm to this end?

Dijkstra Modification (Fa 07, MT1)

We are given a directed graph with positive edge weights. We wish to find a shortest path from s to t, and among all shortest paths, we want the one in which the longest edge is as short as possible. How would you modify Dijkstra's algorithm to this end?

Dijkstra's keeps a variable dist(u) to store the current shortest path distance to a node u, and prev(u) to store the last node needed to reach u in the shortest path found. Create a new variable long(u), which keeps track of the length of the longest edge.

More T/F (Sp 15, MT1)

- If you increase the weights of all edges in a graph by 4, then the MST will stay the same.
- If you increase the weights of all edges in a graph by 4, then the shortest path tree will stay the same.

More T/F (Sp 15, MT1)

- If you increase the weights of all edges in a graph by 4, then the MST will stay the same.
- If you increase the weights of all edges in a graph by 4, then the shortest path tree will stay the same.

True, the cost of every spanning tree goes up by exactly 4*(|V|-1)

False. Can you think of a counter-example?

DFS of a DAG

In depth-first search of a directed graph, the endpoints of edge (A, B) have these pre and post values: $A : [45, 74]; B : [39, 112]$. Can the graph be a DAG? Explain very briefly.

DFS of a DAG

In depth-first search of a directed graph, the endpoints of edge (A, B) have these pre and post values: $A : [45, 74]; B : [39, 112]$. Can the graph be a DAG? Explain very briefly.

Solution: No. Because $[\text{pre}(A), \text{post}(A)] \subset [\text{pre}(B), \text{post}(B)]$, there must be a path from B to A. This, together with the edge (A, B), means that there is a cycle in the graph.

T/F with a DAG

TRUE or FALSE: In a DAG, the number of distinct paths between two vertices is at most $|V|^2$.

T/F with a DAG

TRUE or FALSE: In a DAG, the number of distinct paths between two vertices is at most $|V|^2$.

False. There can be exponentially many distinct paths. Consider the linear ordering of a maximally connected DAG (every vertex has edges pointing to all vertices to the right of it). Then any vertex in between the source and the destination vertex can be in the path or not. This is 2^(|V| -2) paths.

T/F with a DAG

True or False: Every DAG has at least one source.

T/F with a DAG

TRUE OR FALSE: Every DAG has at least one source.

True, pick a vertex and replace it with a parent vertex until you reach a vertex with no parent.
is guaranteed to happen because otherwise there is a cycle.

T/F with a DAG

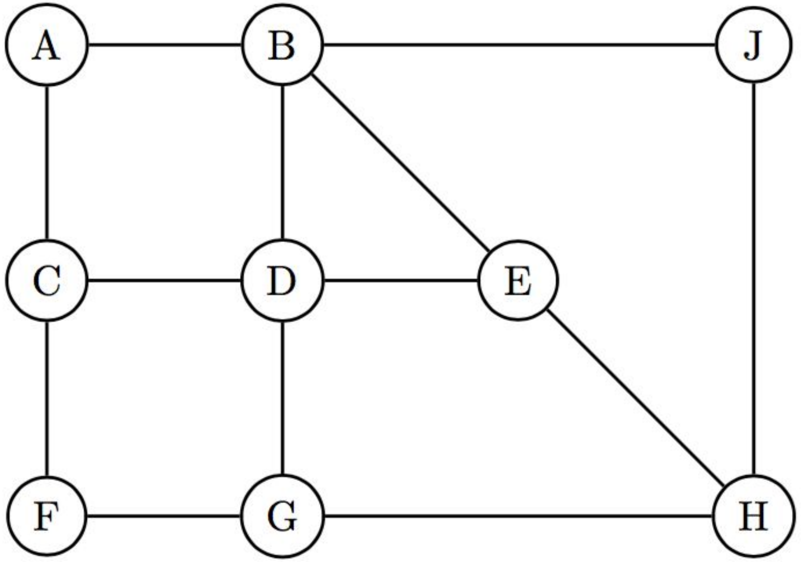
Every DAG has exactly one topological ordering

T/F with a DAG

Every DAG has exactly one topological ordering

False. Every DAG has at least one topological ordering. Can you think of an example?

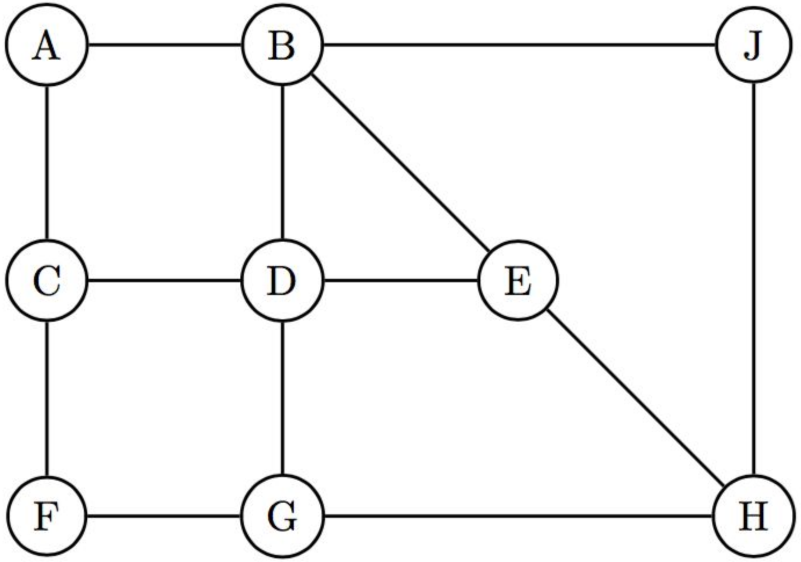
DFS (Fa15 Mt1)



Start at D and break ties alphabetically.

Node	Pre	Post
A		
B		
C		
D		
E		
F		
G		
H		
J		

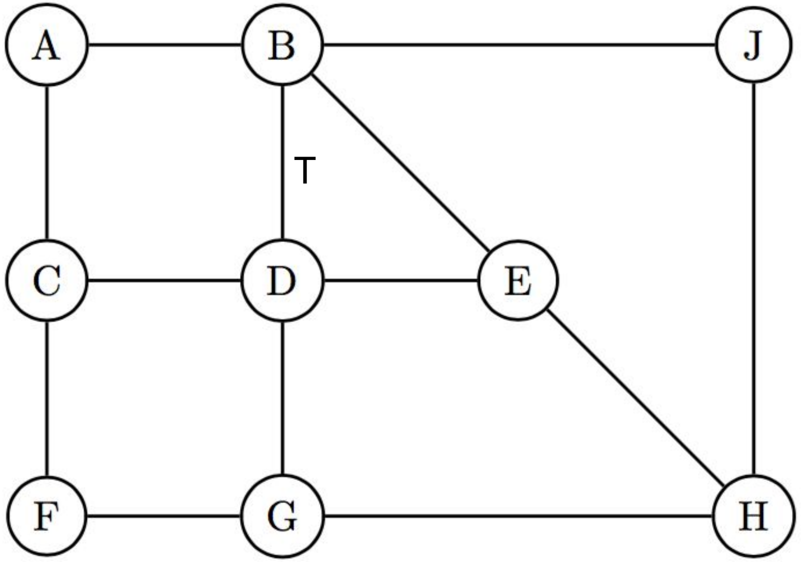
DFS (Fa15 Mt1)



Recursive Stack: D

Node	Pre	Post
A		
B		
C		
D	1	
E		
F		
G		
H		
J		

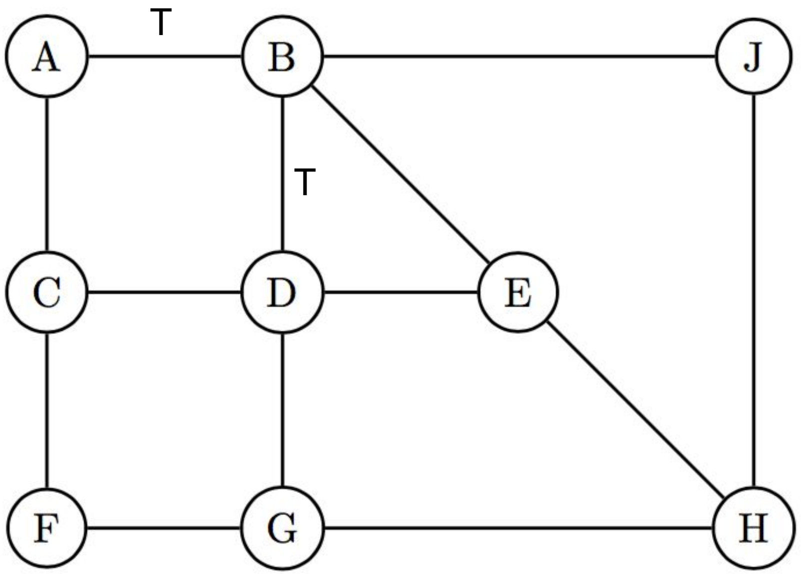
DFS (Fa15 Mt1)



Recursive Stack: D, B

Node	Pre	Post
A		
B	2	
C		
D	1	
E		
F		
G		
H		
J		

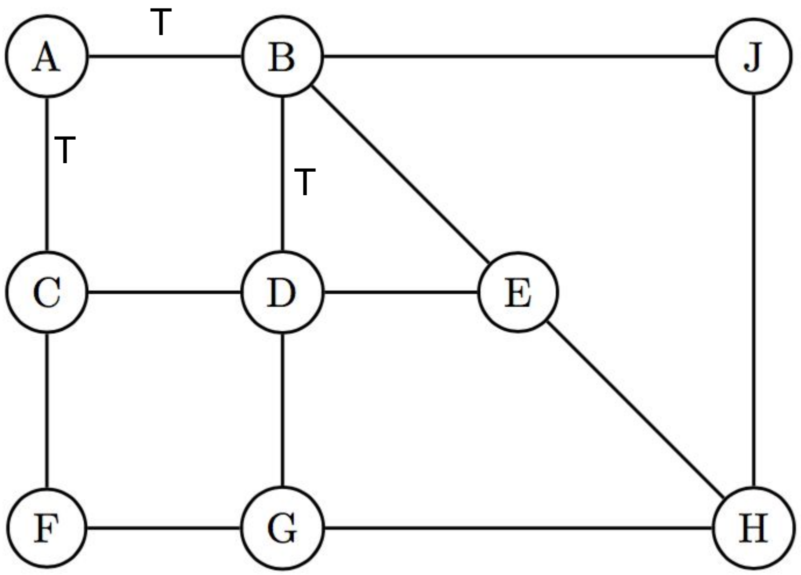
DFS (Fa15 Mt1)



Recursive Stack: D, B, A

Node	Pre	Post
A	3	
B	2	
C		
D	1	
E		
F		
G		
H		
J		

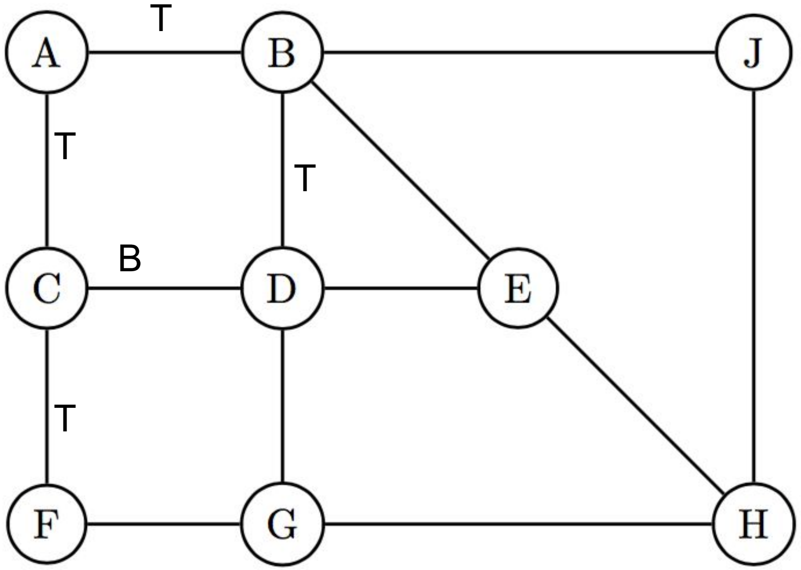
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E		
F		
G		
H		
J		

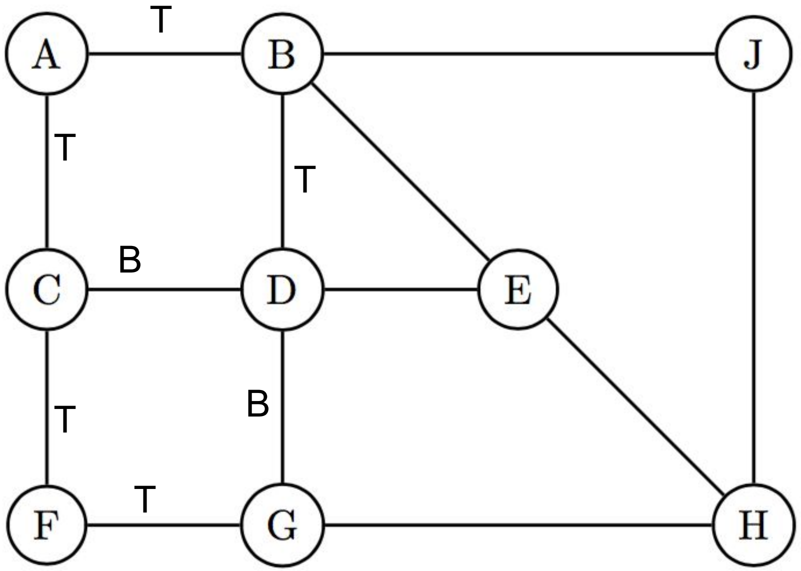
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E		
F	5	
G		
H		
J		

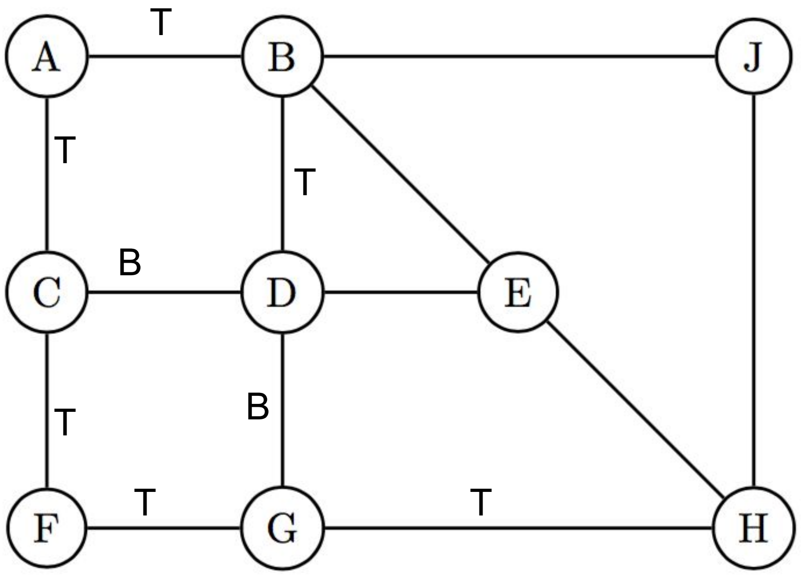
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E		
F	5	
G	6	
H		
J		

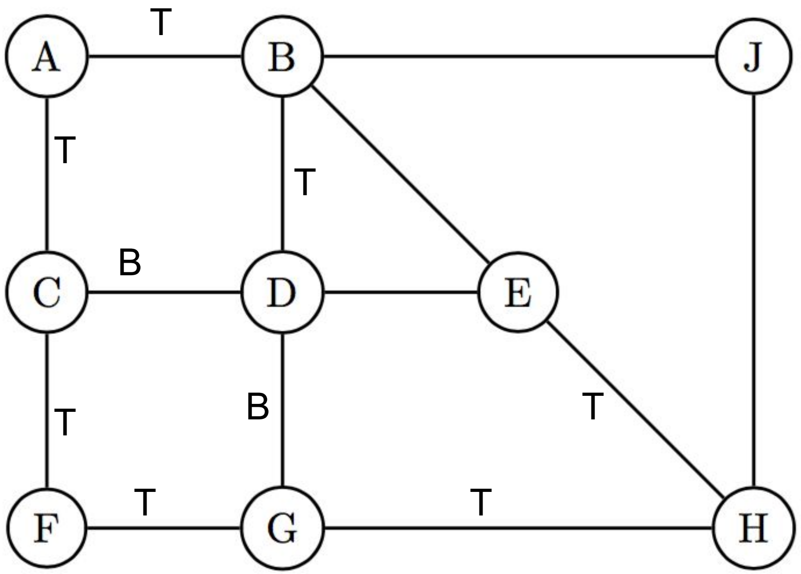
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G, H

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E		
F	5	
G	6	
H	7	
J		

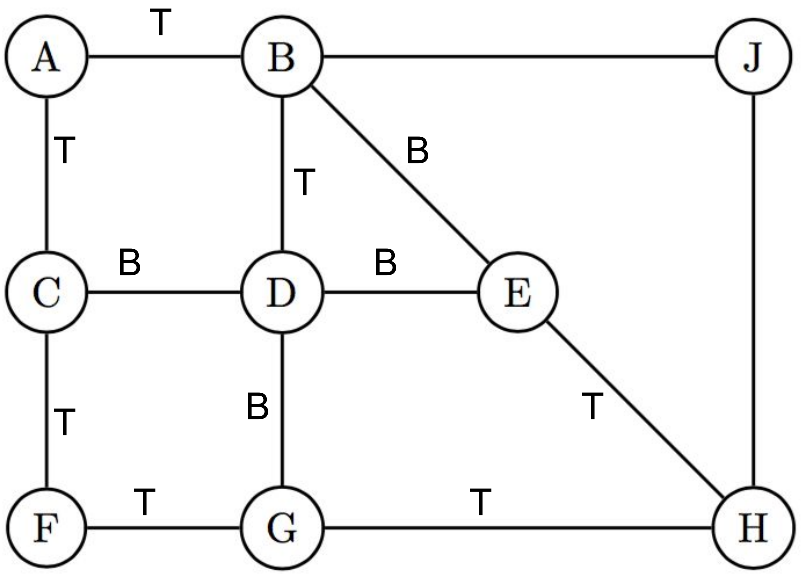
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G, H, E

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	
F	5	
G	6	
H	7	
J		

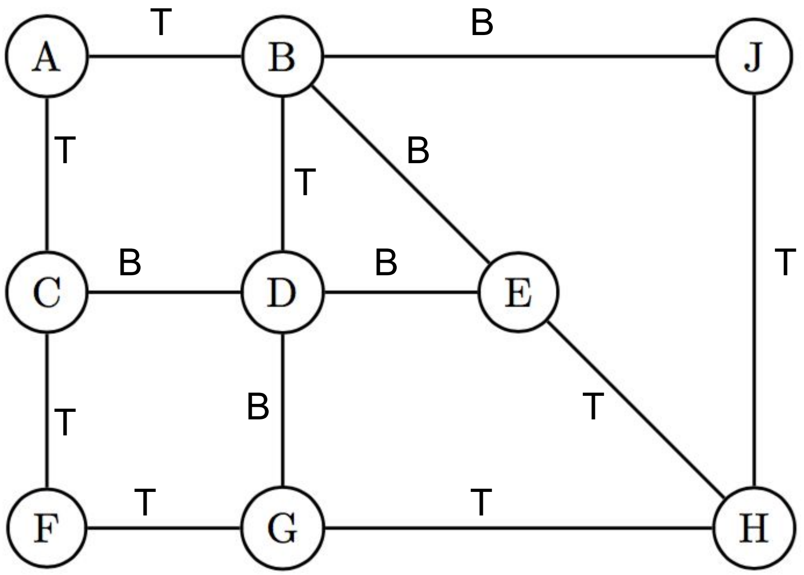
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G, H

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	9
F	5	
G	6	
H	7	
J		

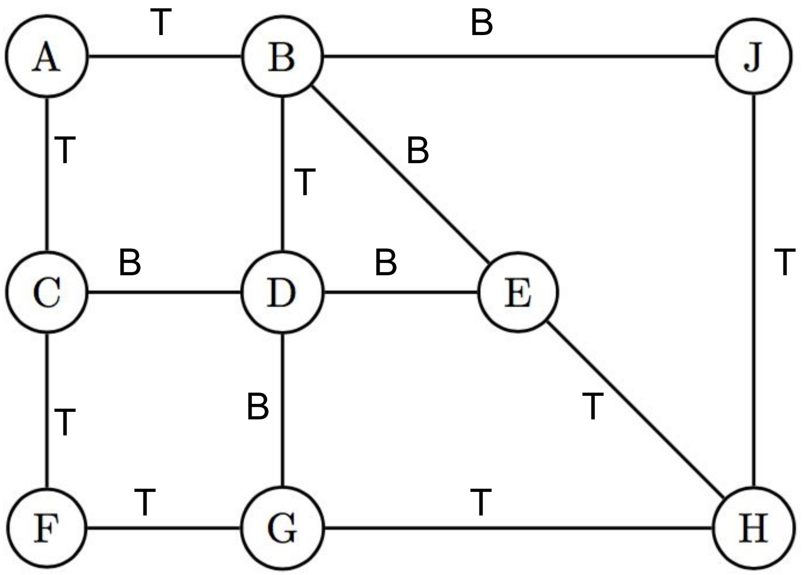
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G, H, J

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	9
F	5	
G	6	
H	7	
J	10	

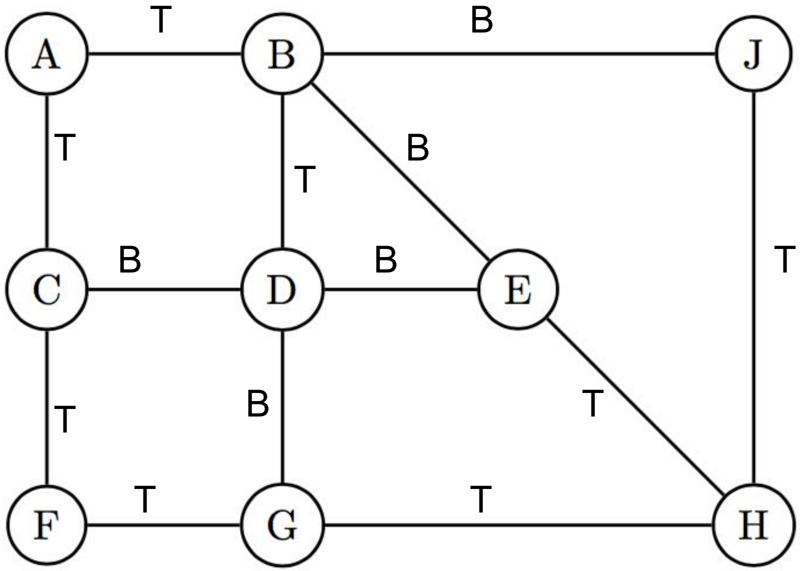
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G, H

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	9
F	5	
G	6	
H	7	
J	10	11

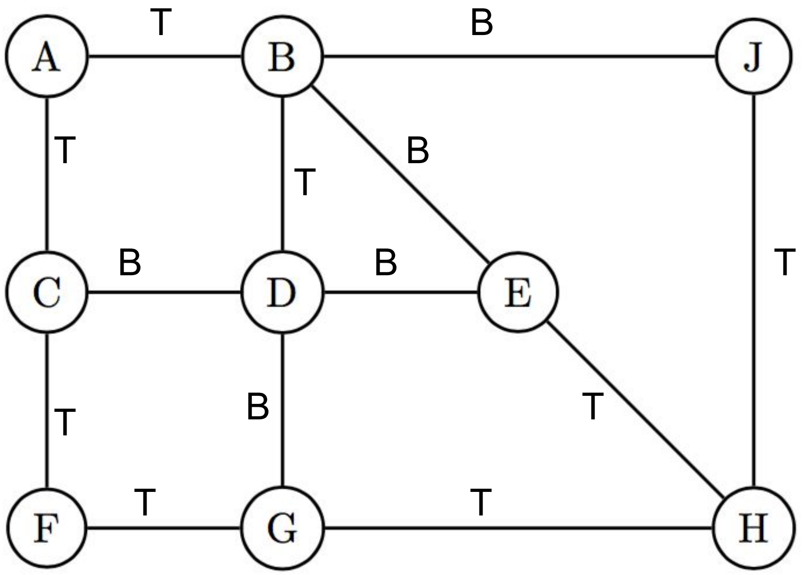
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F, G

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	9
F	5	
G	6	
H	7	12
J	10	11

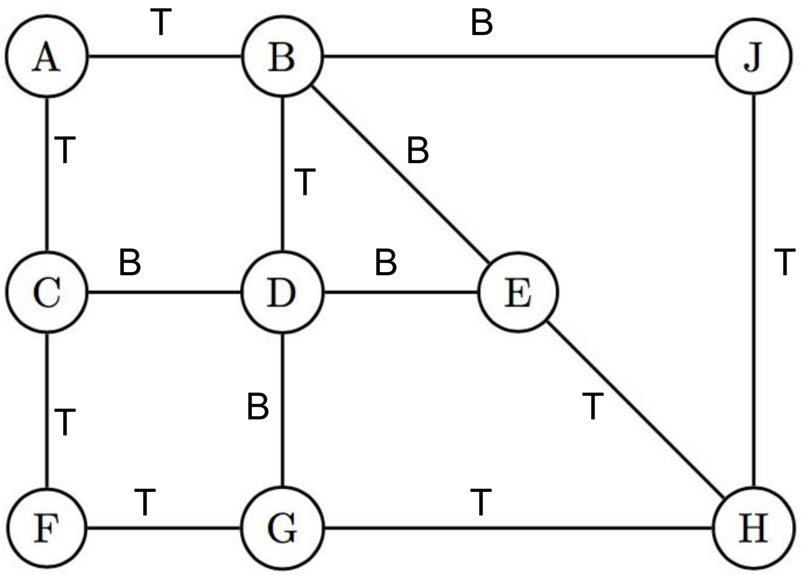
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C, F

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	9
F	5	
G	6	13
H	7	12
J	10	11

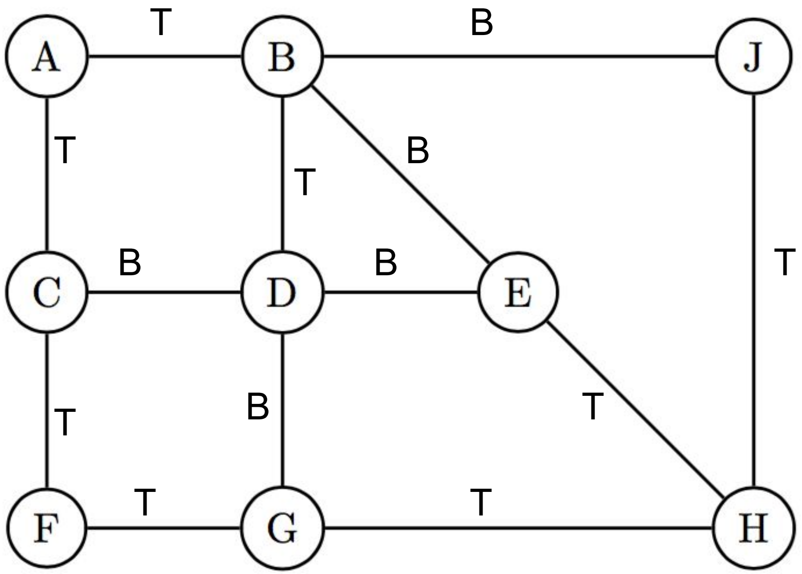
DFS (Fa15 Mt1)



Recursive Stack: D, B, A, C

Node	Pre	Post
A	3	
B	2	
C	4	
D	1	
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

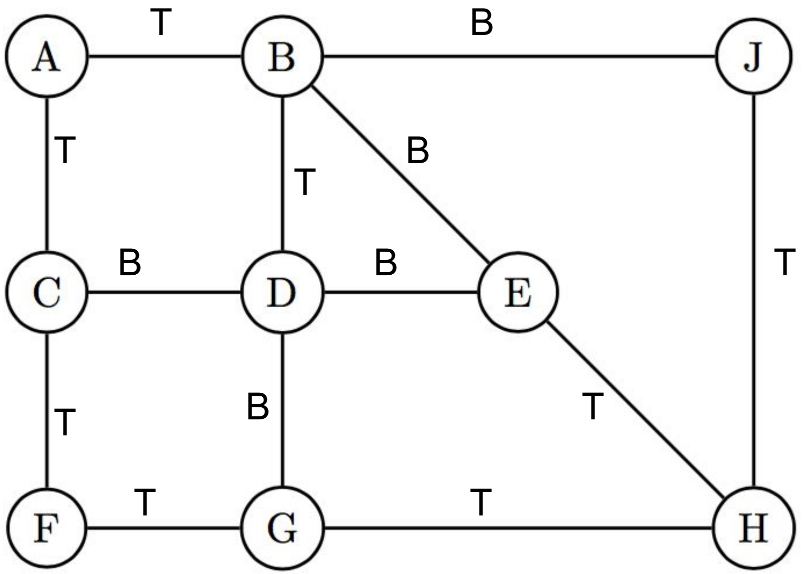
DFS (Fa15 Mt1)



Recursive Stack: D, B, A

Node	Pre	Post
A	3	
B	2	
C	4	15
D	1	
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

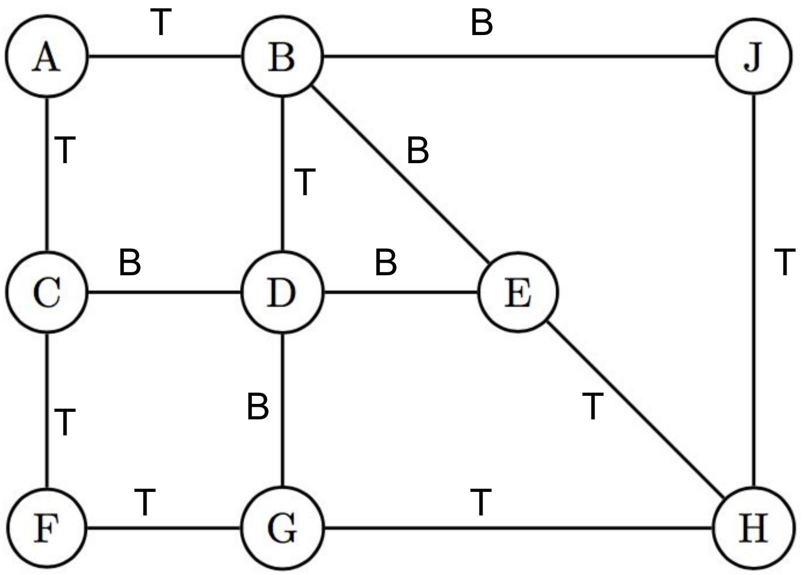
DFS (Fa15 Mt1)



Recursive Stack: D, B

Node	Pre	Post
A	3	16
B	2	
C	4	15
D	1	
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

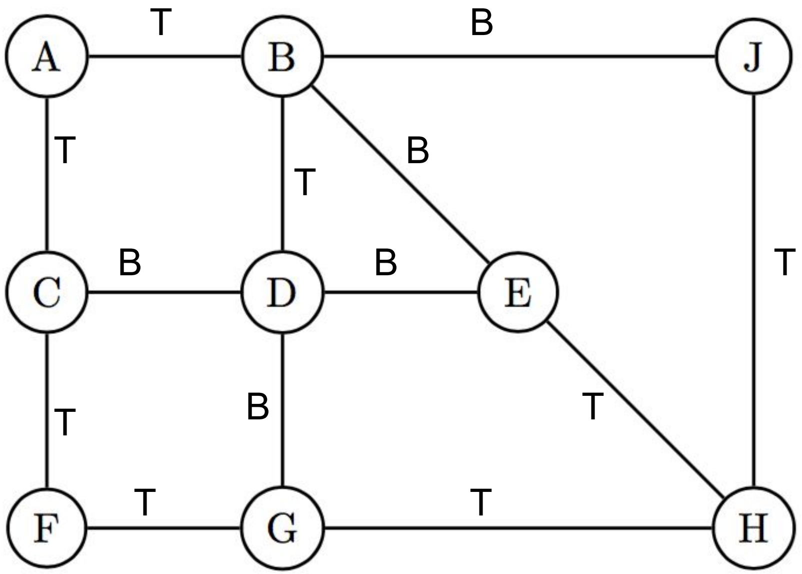
DFS (Fa15 Mt1)



Recursive Stack: D

Node	Pre	Post
A	3	16
B	2	17
C	4	15
D	1	
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

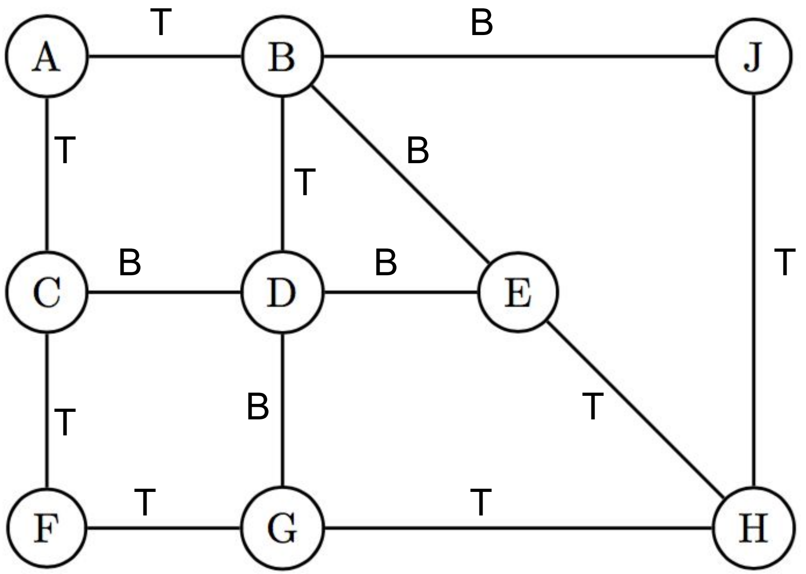
DFS (Fa15 Mt1)



Recursive Stack:

Node	Pre	Post
A	3	16
B	2	17
C	4	15
D	1	18
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

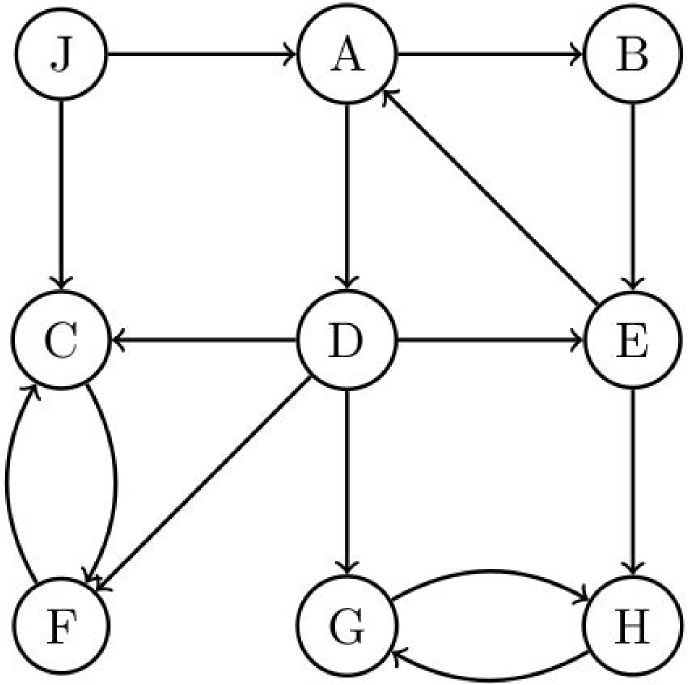
DFS (Fa15 Mt1)



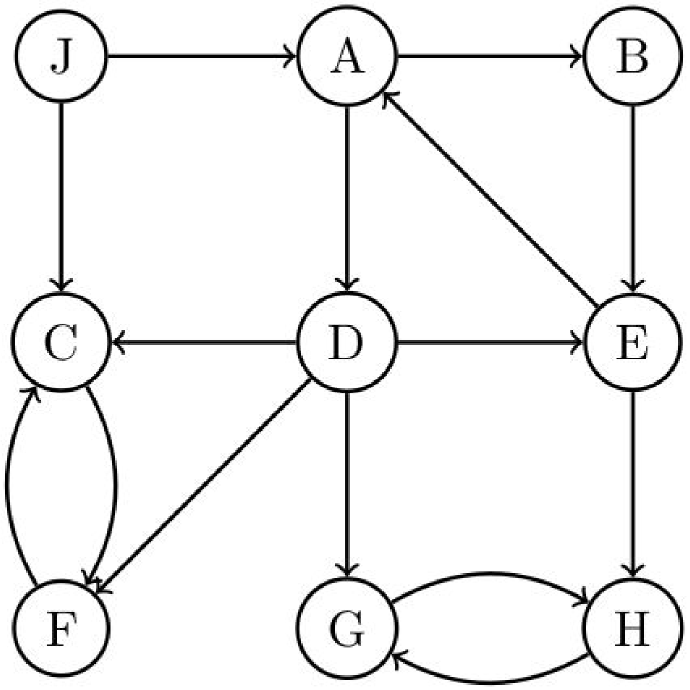
Since pre(D) is the smallest and post(D) is the largest, the graph is connected.

Node	Pre	Post
A	3	16
B	2	17
C	4	15
D	1	18
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

Find the SCC's

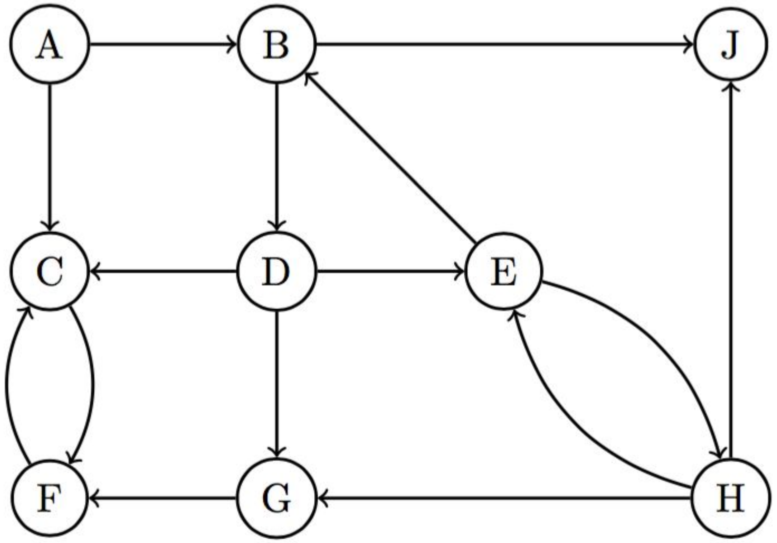


Find the SCC’s

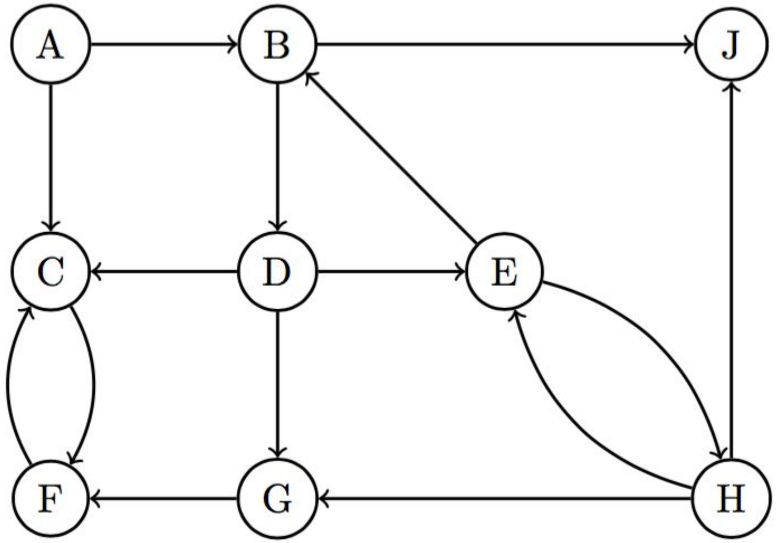


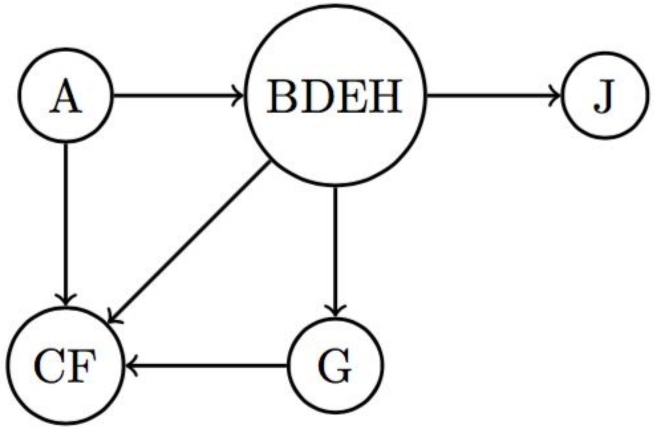
Solution: The SCC’s are (A, B, D, E), (C, F), (G, H), (J).

DAG of SCC



DAG of SCC





The strongly connected components are $\{A\}, \{B, D, E, H\}, \{C, F\}, \{G\}, \{J\}$

FFT

Suppose we want to evaluate a polynomial

$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$

Need $n \geq d + 1$ points to evaluate.

Choose the next power of 2. In this case we want 4 points, which will be the 4 roots of unity (1, -1, i, -i). Evaluate A(1), A(-1), A(i), A(-i).

Split $A(x) = A_e(x^2) + x * A_o(x^2)$.

$A(x) = (a_0 + a_2(x^2)) + x * (a_1 + a_3(x^2))$

$A_e(x) = a_0 + a_2x^2$ and $A_0(x) = a_1 + a_3x^2$

FFT

$A(1) = A_e(1) + A_o(1)$

$A(-1) = A_e(1) - A_o(1)$

$A(i) = A_e(-1) + i * A_o(-1)$

$A(-i) = A_e(-1) - i * A_o(-1)$

Only need to evaluate $A_e(1)$, $A_e(-1)$, $A_o(1)$, $A_o(-1)$.

We went from evaluat 4 points of $A(x)$ to 2 points of $A_e(x)$ and 2 points of $A_o(x)$.

That’s 2 recursive calls to problems of half the size. At each level, arithmetic takes $O(n)$.

FFT Recurrence Relation: $T(n) = 2 * T(n/2) + O(n) = O(n * \log n)$

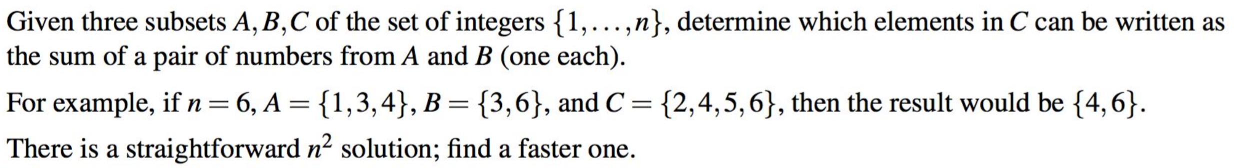
FFT (Sp16 Mt1)

Given three subsets A, B, C of the set of integers $\{1, \ldots, n\}$, determine which elements in C can be written as the sum of a pair of numbers from A and B (one each).

For example, if $n = 6$, $A = \{1,3,4\}$, $B = \{3,6\}$, and $C = \{2,4,5,6\}$, then the result would be $\{4,6\}$.

There is a straightforward n^2 solution; find a faster one.

FFT (Sp16 Mt1)



Given three subsets A, B, C of the set of integers $\{1, \ldots, n\}$, determine which elements in C can be written as the sum of a pair of numbers from A and B (one each).

For example, if $n = 6$, $A = \{1,3,4\}$, $B = \{3,6\}$, and $C = \{2,4,5,6\}$, then the result would be $\{4,6\}$.

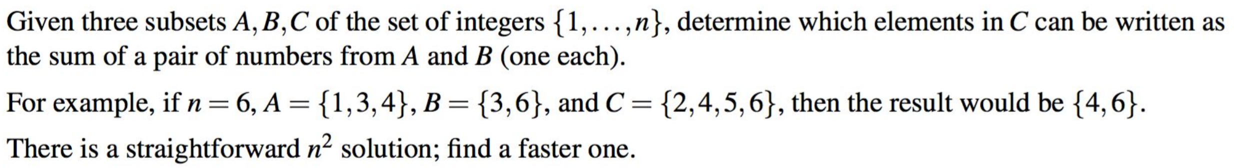
There is a straightforward n^2 solution; find a faster one.

Multiplying polynomial adds exponents.

$A(x) = \Sigma_a x^a$, $B(x) = \Sigma_b x^b$

Use FFT to obtain $A(x) * B(x) = \Sigma_{a,b} x^{a + b}$. For every nonzero coefficient term in this new polynomial, we check the power against set C and add it to our result if it is in C.

FFT (Sp16 Mt1)



Given three subsets A, B, C of the set of integers $\{1, \ldots, n\}$, determine which elements in C can be written as the sum of a pair of numbers from A and B (one each).

For example, if $n = 6$, $A = \{1,3,4\}$, $B = \{3,6\}$, and $C = \{2,4,5,6\}$, then the result would be $\{4,6\}$.

There is a straightforward n^2 solution; find a faster one.

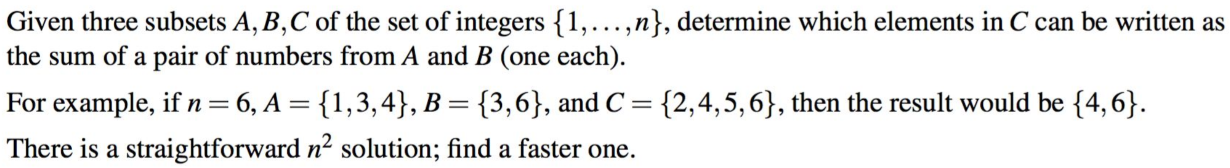
Multiplying polynomial adds exponents.

$A(x) = \Sigma_a x^a$, $B(x) = \Sigma_b x^b$

Use FFT to obtain $A(x) * B(x) = \Sigma_{a,b} x^{a + b}$. For every nonzero coefficient term in this new polynomial, we check the power against set C and add it to our result if it is in C.

Both A(x) and B(x) are n degree polynomials. A(x) * B(x) are 2n degree polynomials. Creating the polynomials takes O(n) time. Multiplication via FFT takes O(n log n). Checking the values against C takes O(n). Thus final runtime is O(n log n).

FFT (Sp16 Mt1)



Given three subsets A, B, C of the set of integers $\{1, \ldots, n\}$, determine which elements in C can be written as the sum of a pair of numbers from A and B (one each).

For example, if $n = 6$, $A = \{1,3,4\}$, $B = \{3,6\}$, and $C = \{2,4,5,6\}$, then the result would be $\{4,6\}$.

There is a straightforward n^2 solution; find a faster one.

Multiplying polynomial adds exponents.

$A(x) = \Sigma_a x^a$, $B(x) = \Sigma_b x^b$

Use FFT to obtain $A(x) * B(x) = \Sigma_{a,b} x^{a + b}$. For every nonzero coefficient term in this new polynomial, we check the power against set C and add it to our result if it is in C.

Both A(x) and B(x) are n degree polynomials. A(x) * B(x) are 2n degree polynomials. Creating the polynomials takes O(n) time. Multiplication via FFT takes O(n log n). Checking the values against C takes O(n). Thus final runtime is O(n log n).

Power of nonzero coefficient terms are some combination of $a_i + b_j$. All the nonzero coefficient terms together forms all the combinations. So we can just check it

Algorithm Design: Divide and Conquer

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Algorithm Design: Divide and Conquer

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

We use divide-and-conquer to split the array into two halves and solve the problem on each of the halves. If A_1 and A_2 denote the two halves, then either the optimal $i^*, j^* \in A_1$, $i^*, j^* \in A_2$, or $i^* \in A_1$ and $i^* \in A_2$. The values x, y, and z each handle these cases separately.

```
FindMax($A[0..n-1]$):
1. If $n \leq 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.

3. Set $x :=$ FindMax($A[0 \dots k-1]$).

4. Set $y :=$ FindMax($A[k \dots n]$).

5. Set $z := \max(A[0], \dots, A[k-1]) + (\max(A[k], \dots, A[n]))^2$ .
6. Return $\max(x, y, z)$.
```

Algorithm Design: No rules, go faster

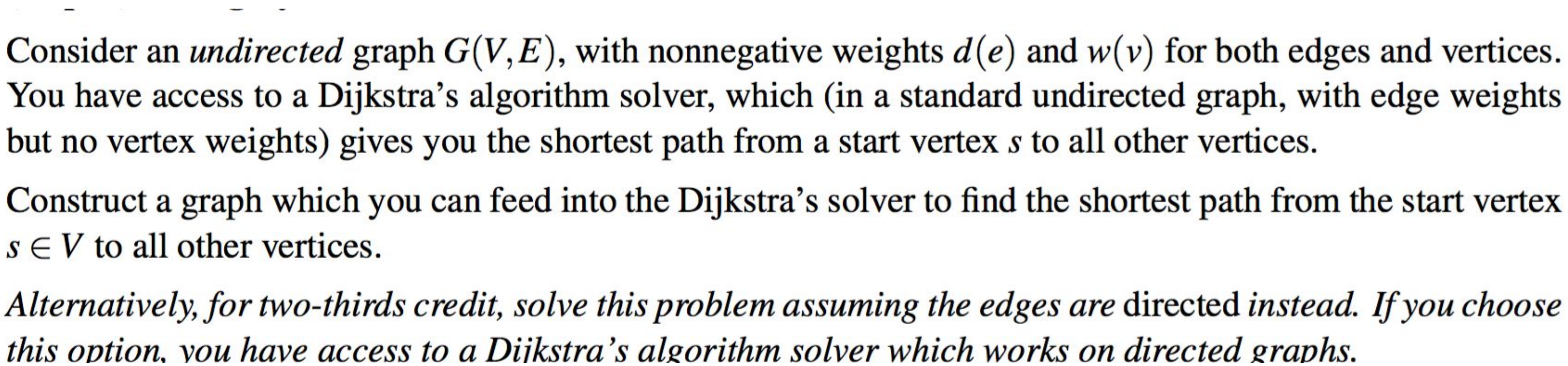
We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Algorithm Design: No rules, go faster

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

```
procedure FindMax($A[0..n-1]$)
    Set $m$ := index of the maximum element in $A$.
    Set $bestmax, i, j := -\infty, -1, -1$.
    for $l \in 0..m-1$ do
        if $A[l] + A[m]^2 > bestmax$ then
            Set $bestmax, i, j := A[l] + A[m]^2, l, m$
    for $l \in m+1..n-1$ do
        if $A[m] + A[l]^2 > bestmax$ then
            Set $bestmax, i, j := A[m] + A[l]^2, m, l$
    Output $bestmax$
```

Weighty Vertices (Sp16 Mt1)

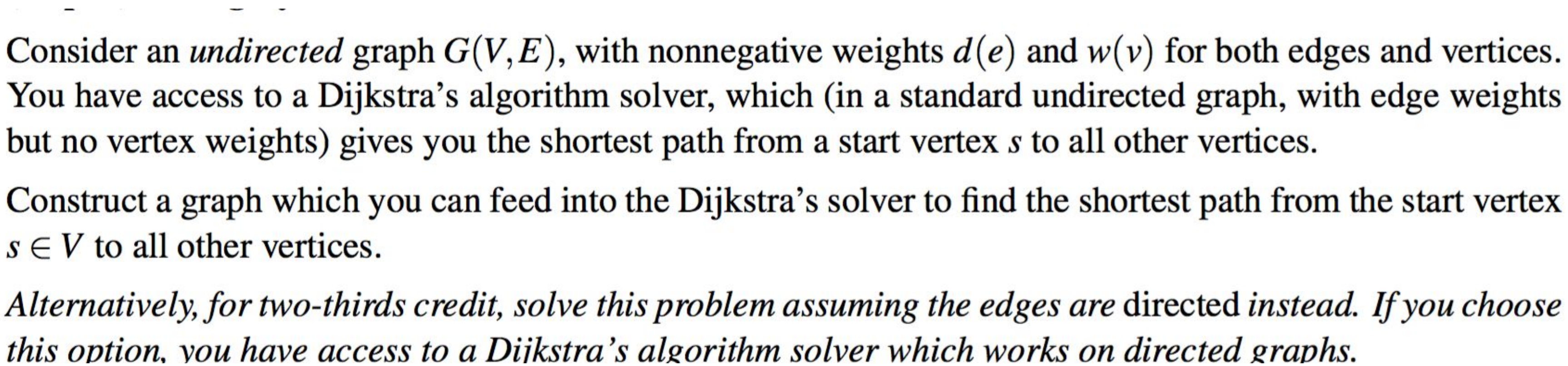


Consider an *undirected* graph $G(V,E)$, with nonnegative weights $d(e)$ and $w(v)$ for both edges and vertices. You have access to a Dijkstra's algorithm solver, which (in a standard undirected graph, with edge weights but no vertex weights) gives you the shortest path from a start vertex s to all other vertices.

Construct a graph which you can feed into the Dijkstra's solver to find the shortest path from the start vertex $s \in V$ to all other vertices.

Alternatively, for two-thirds credit, solve this problem assuming the edges are directed *instead. If you choose this option, you have access to a Dijkstra's algorithm solver which works on directed graphs.*

Weighty Vertices (Sp16 Mt1)



Consider an *undirected* graph $G(V,E)$, with nonnegative weights $d(e)$ and $w(v)$ for both edges and vertices. You have access to a Dijkstra's algorithm solver, which (in a standard undirected graph, with edge weights but no vertex weights) gives you the shortest path from a start vertex s to all other vertices.

Construct a graph which you can feed into the Dijkstra's solver to find the shortest path from the start vertex $s \in V$ to all other vertices.

Alternatively, for two-thirds credit, solve this problem assuming the edges are directed *instead. If you choose this option, you have access to a Dijkstra's algorithm solver which works on directed graphs.*

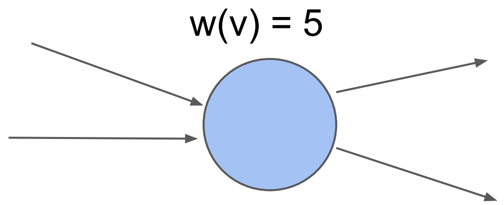
Hint: Think about what if the edges are directed instead?

How can we build off the directed solution for an undirected graph?

Weighty Vertices (Sp16 Mt1) - Directed

Force shortest path to add weight *w(v)* to distances if we use the vertex.

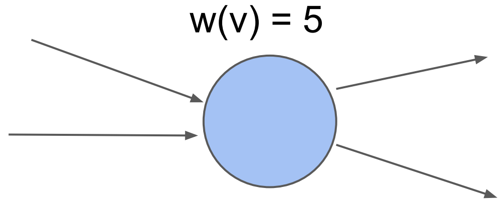
How to tell whether we have gone through a vertex?

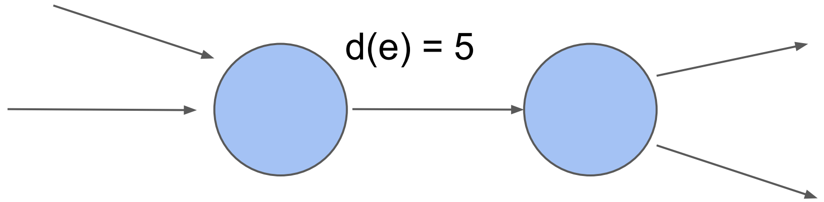


Weighty Vertices (Sp16 Mt1) - Directed

Add weight *w(v)* if we use the vertex.

How to tell whether we have gone through a vertex?

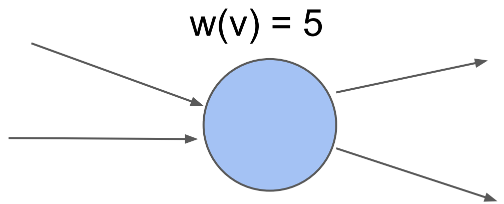


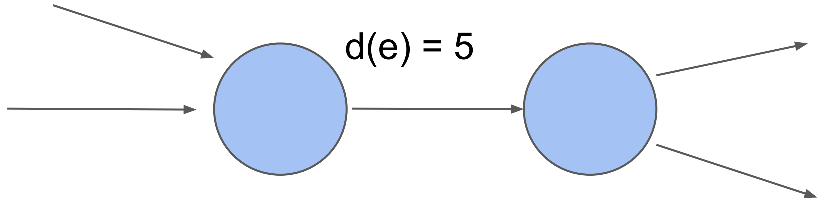


Weighty Vertices (Sp16 Mt1) - Directed

Add weight *w(v)* if we use the vertex.

How to tell whether we have gone through a vertex?





For every vertex v, we split it into 2 vertices, v_1 and v_2.

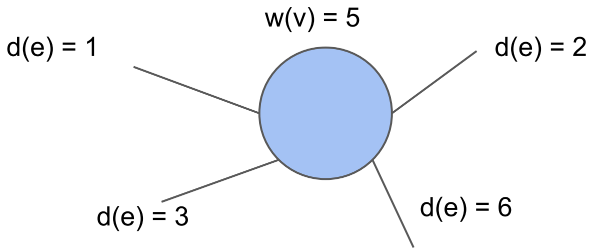
Add edge (v_1, v_2) with weight *w(v)*.

Run Dijkstra's shortest path algorithm on the new graph.

If a shortest path goes through v_1 and v_2, then it must have gone through v in the original graph.

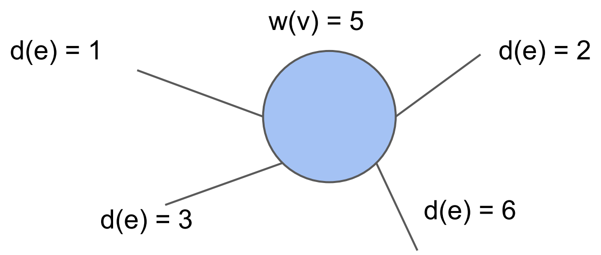
Weighty Vertices (Sp16 Mt1) - Undirected

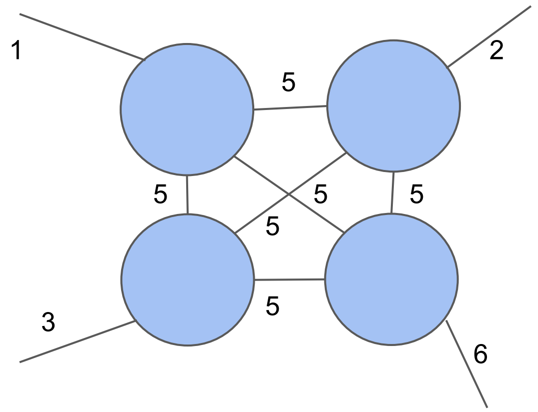
What can we do for an undirected graph?



Weighty Vertices (Sp16 Mt1) - Undirected

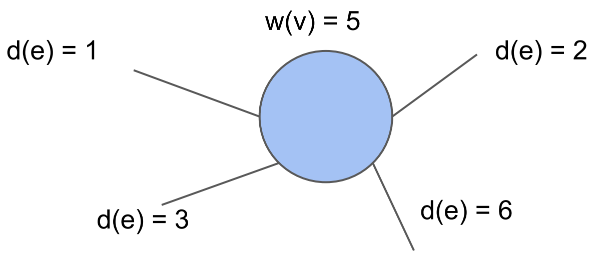
What can we do for an undirected graph?

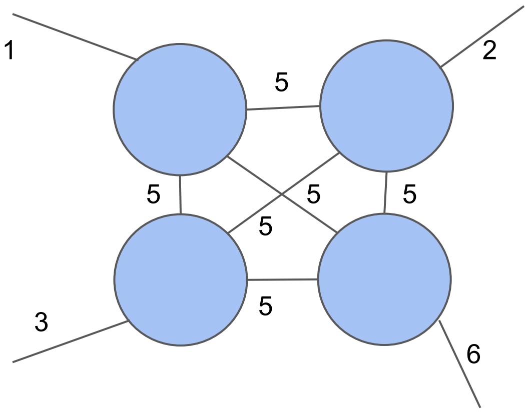




Weighty Vertices (Sp16 Mt1) - Undirected

What can we do for an undirected graph?





Create a vertex v_i for every edge incident to vertex v. Add an edge between every pair of v_i, v_j with edge weight w(v).

Run Dijkstra's algorithm on this new graph.

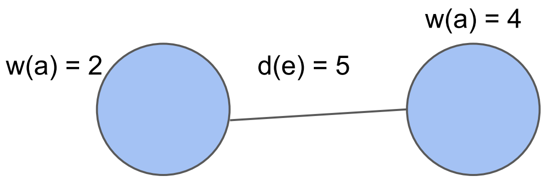
Using any of the newly created edges means we're using the vertex and taking into account the vertex cost.

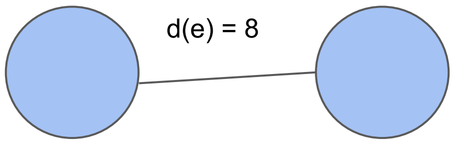
Each vertex becomes a complete sub-graph to replicate the behavior of entering and leave the vertex using any undirected edge.

Weighty Vertices (Sp16 Mt1)

Alternative solution (undirected):

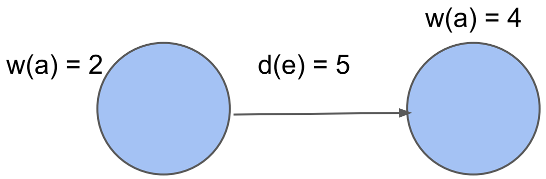
For every edge, add half of the vertex weights that the edge connects to the edge weight. d((a, b)) = w(a)/2 + w(b)/2 + d((a, b))

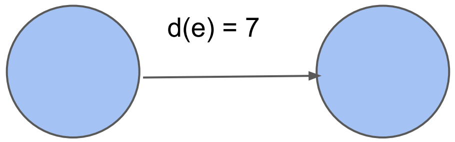




Alternative solution (directed):

For every edge, add the weight of the income vertex to the distance. d((a, b)) = d((a, b)) + w(b)/2





Super Mario (Hard)

You wish to use your knowledge of CS 170 to get a leg up on playing Super Mario Kart. In Level 17, you must drive through Bowser's Castle, which you can model as a directed graph. Traveling through each passage in the castle (directed edge) adds a positive or negative number of points to your score, depending upon what type of monster/object/power-up is on it. (Call $w(e)$ to find the number of points (which can be negative) each edge adds.) To successfully traverse the level, you must find a path, *perhaps with repeated edges*, from the castle entrance s to the exit t, such that your score never goes negative *at any point in time*. Assume that every vertex is reachable from s and every vertex is able to reach t. Use your knowledge of CS 170 to design an efficient algorithm to find such a path if it exists, or to determine that there is no such path.

Super Mario (Hard)

Since the number of points are edge weights, we want some path such that the accumulated points don't become negative. We want some positive cycle.

Let's modify Bellman Ford algorithm as this can find shortest (longest) paths with negative edge weights and deals with positive/negative cycles.

At each update, verify that the value of the path never becomes negative.

If there is a positive cycle from *s* such that the path does not become negative, there will be some legal path from *s* to *t*

Super Mario (Hard)

Pseudocode:

```
procedure MARIO($G = (V, E); w : E \to \mathbb{R}; s, t \in V$)                ▷ Initialize:
    for $v \in V$ do
        $Value[v] \leftarrow -\infty$
    $Value[s] \leftarrow 0$
                                                                              ▷ Main Loop:
    for $i = 1 \dots |V| - 1$ do
        for $(u \to v) \in E$ do
            if $Value[v] < Value[u] + w(e)$ AND $Value[u] \geq 0$ then          ▷ Idea # 2
                $Value[v] \leftarrow Value[u] + w(e)$
    if $Value[t] \geq 0$ then return True
                                                                     ▷ Find positive cycles
    for $(u \to v) \in E$ do
        if $Value[v] < Value[u] + w(e)$ AND $Value[u] \geq 0$ then return True
    return False
```

Questions

Any topic that we should talk about now?