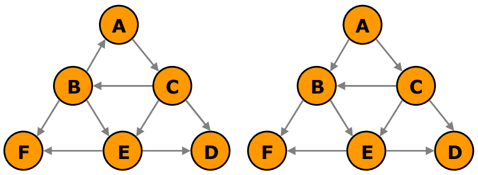
Week 12: Graphs

Continue from Last Week

Topological Sort

Definition

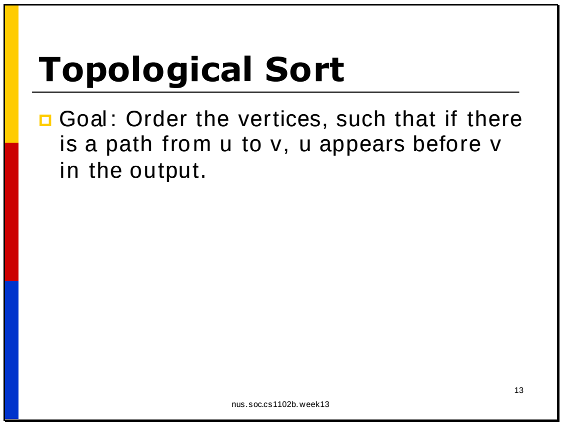
- Directed Acyclic Graph (dag): A directed graph with no cycle.



Define an acyclic graph to be a graph without cycle. An undirected acyclic graph is thus simply a tree. A directed acyclic graph is also called a "dag" for short. We also define in-degree of a vertex to be the number of incoming edges, and out-degree of a vertex to be the number of outgoing edges.

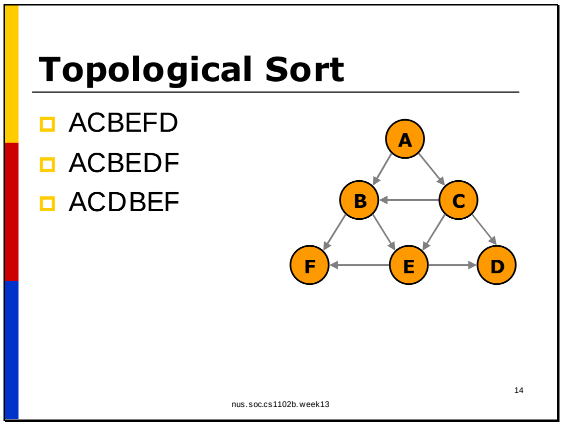
Definitions

- **in-degree** of a vertex
  - number of incoming edges
- **out-degree** of a vertex
  - number of outgoing edges



Topological Sort

- Goal: Order the vertices, such that if there is a path from u to v, u appears before v in the output.



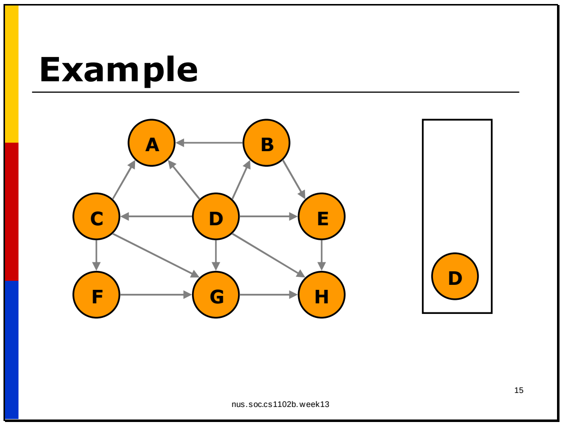
Topological Sort

- ACBEFD
- ACBEDF
- ACDBEF

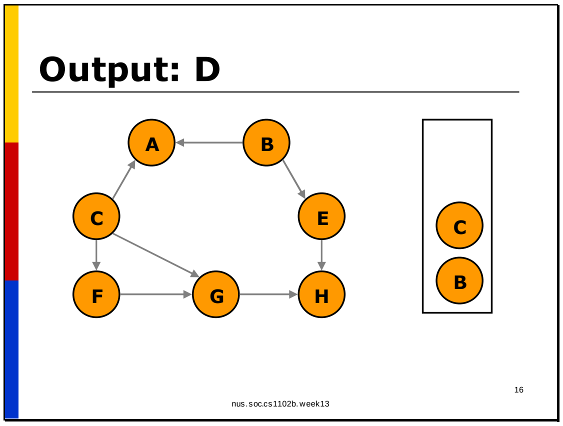
We are interested in solving this problem: Given a dag, we want to order the vertices such that if there is a path from u to v, u appears before v in the output. This is useful when vertices represents items with dependencies (such as course prerequisite) and we want to order the items without violating the dependencies.

Topological sort is not unique. In the graph above, ACBEFD and ACBEDF are both valid topological sorted orders. ACDBEF is NOT topologically sorted because D appears before B and there is a path from B to D.

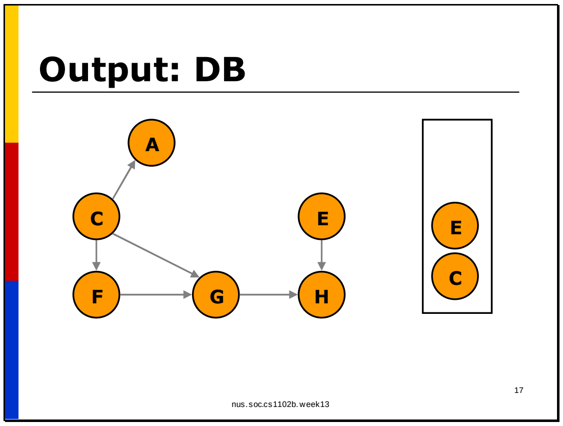
We perform topological sort by repeatedly en-queueing vertices with in-degree 0 into a queue, output the vertex de-queued from the queue and remove the edges from that vertex. Since the order where we en-queued vertices with 0 in-degree into the queue is not unique, the output is not unique.



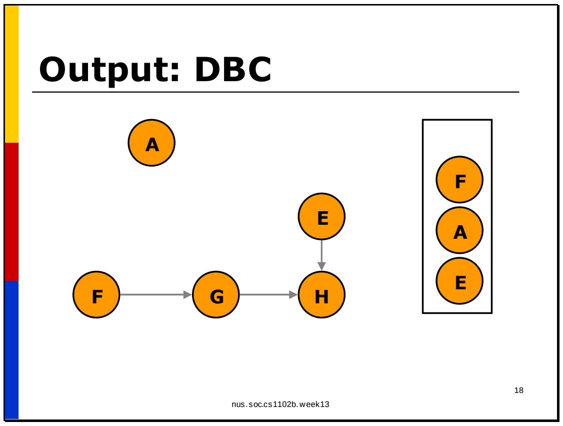
Example



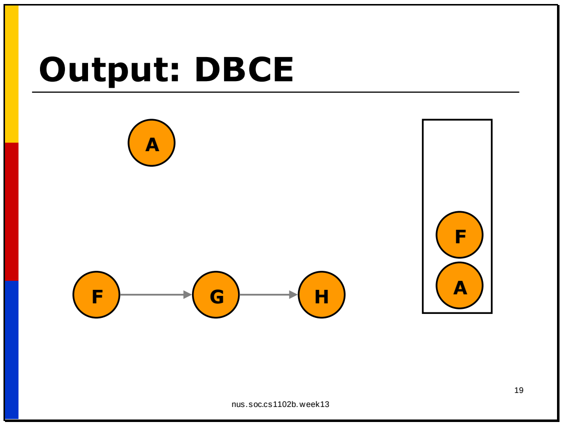
Output: D



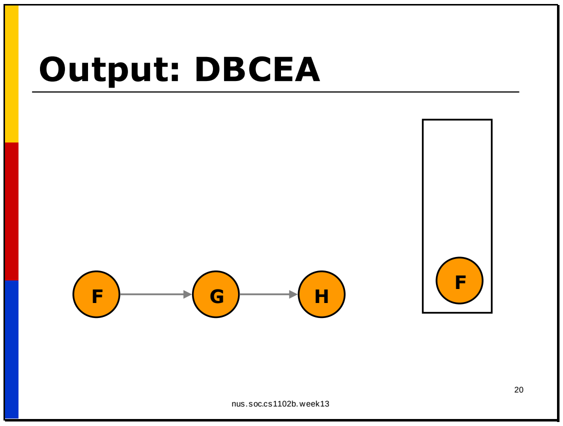
Output: DB



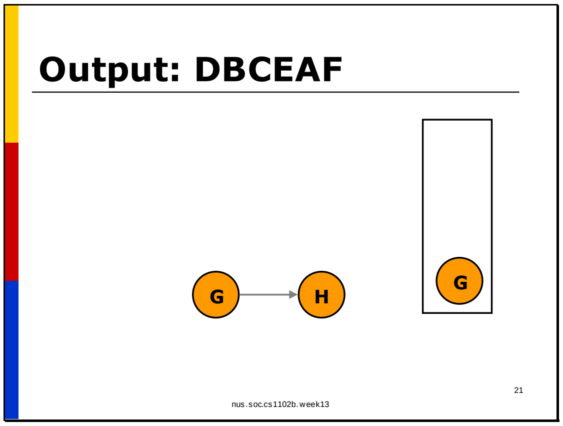
Output: DBC



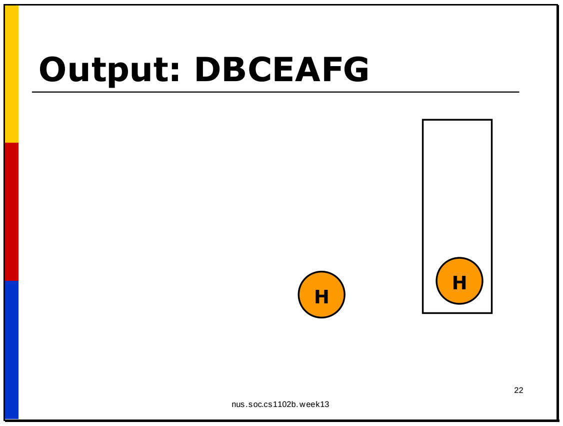
Output: DBCE



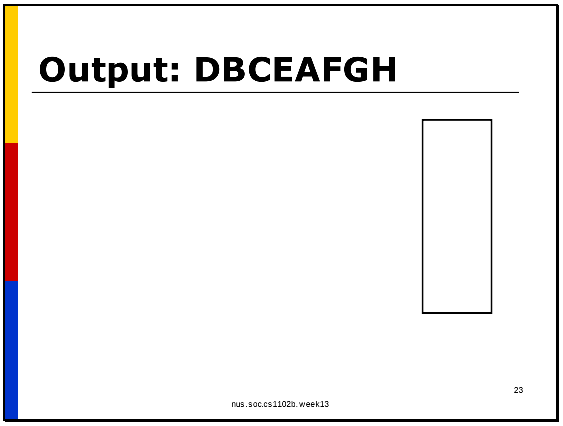
Output: DBCEA



Output: DBCEAF



Output: DBCEAFG



Output: DBCEAFGH

Pseudo code for Toposort

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
  v = q.deq()
  print v
  remove v from G
  put all vertices with in-degree 0 into q
```

Week 13:
Algorithm Design Techniques

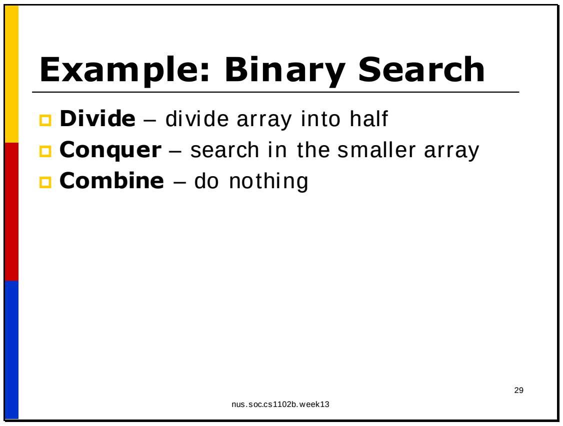
Review of Techniques

- Divide-and-Conquer Algorithm
- Dynamic Programming
- Greedy Algorithm

Divide-and-Conquer Algorithm

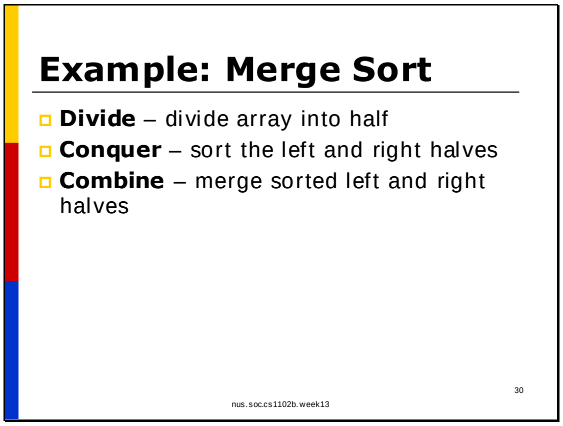
3 Steps

- **Divide** – divide problem into subproblems
- **Conquer** – solve the subproblems
- **Combine** – the solutions to the subproblems into the solution for the original problem.



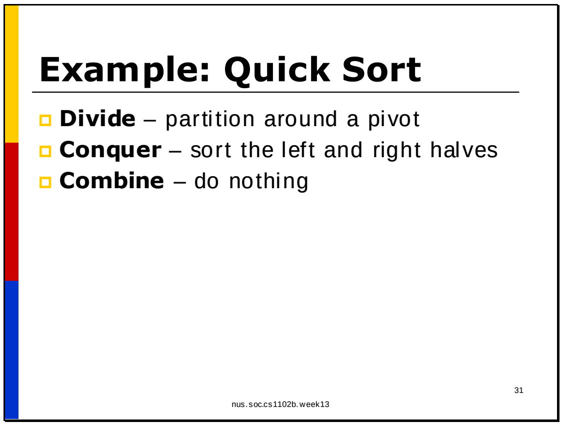
Example: Binary Search

- **Divide** – divide array into half
- **Conquer** – search in the smaller array
- **Combine** – do nothing



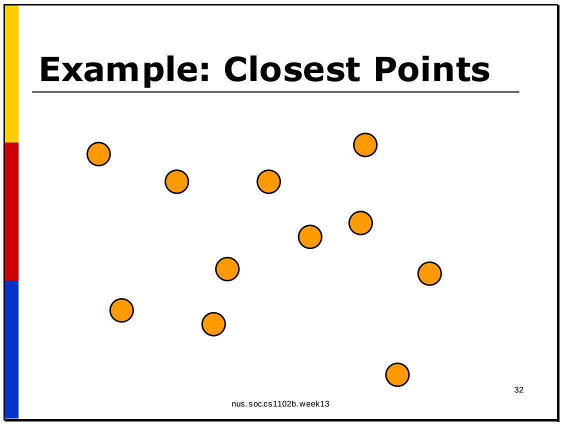
Example: Merge Sort

- **Divide** – divide array into half
- **Conquer** – sort the left and right halves
- **Combine** – merge sorted left and right halves



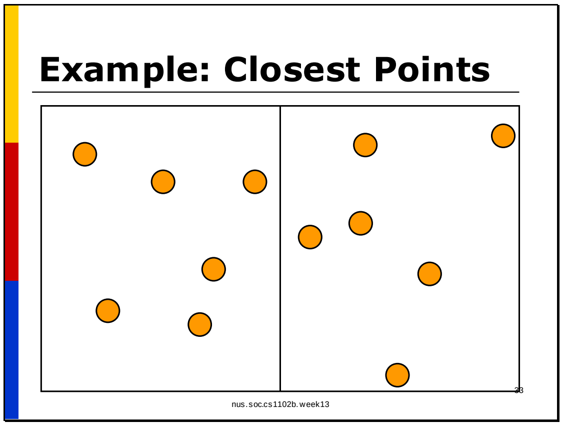
Example: Quick Sort

- **Divide** – partition around a pivot
- **Conquer** – sort the left and right halves
- **Combine** – do nothing



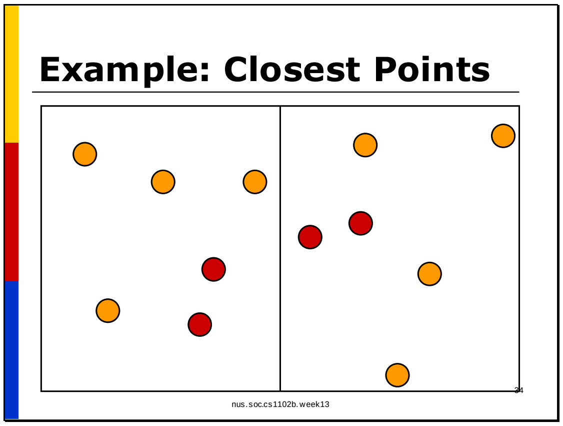
Example: Closest Points

Divide-and-conquer has many other applications besides sorting. One application of this technique is to find two points that are the closest, among a given set of points. A straight forward method of comparing every pairs would give an O(N) running time. By using divide-and-conquer, we can achieve O(NlogN) running time. The details of this algorithm is out of the scope of this course, therefore, only a sketch of the algorithm will be presented.



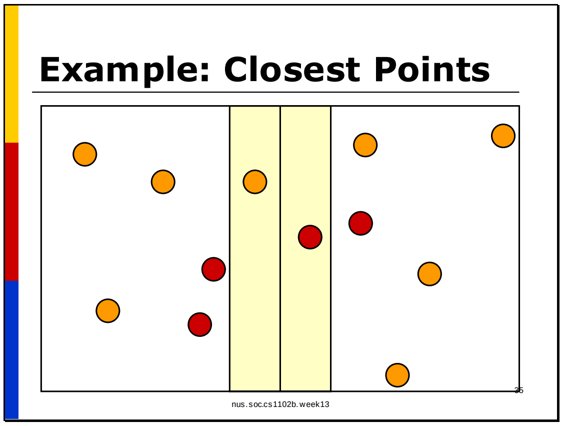
Example: Closest Points

DIVIDE: Given a set of points on a 2D plane, divide the points into two sets L and R such that they have equal number of points. (If the number of given points is odd, then one set will have one point more than the other.)



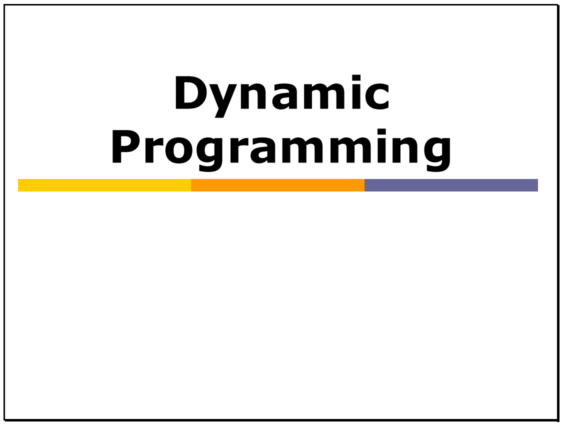
Example: Closest Points

CONQUER: Recursively find the closest points in L and R.



Example: Closest Points

COMBINE: The closest pair must be either one of the closest pairs in the two sets, or consists of one point in each L and R. We then check for the points in a strip of distance d, where d is the smaller distance of the closest pairs in L and R.

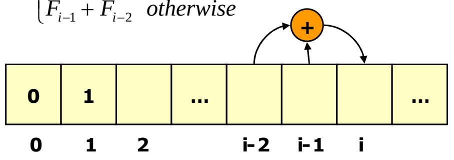


Dynamic Programming

The next algorithm design paradigm is dynamic programming. The idea of dynamic programming is that you solve the program by filling up a table. The problem is normally recursive in nature.

Fibonacci Numbers

$$F_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ F_{i-1} + F_{i-2} & otherwise \end{cases}$$



You have seen this in the calculation of fibonacci numbers.

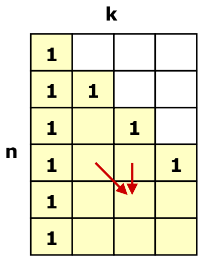
Fibonacci Numbers

fib(n)

```
x[0] = 0
x[1] = 1
for (i = 2; i <= n; i++)
    x[i] = x[i-1] + x[i-2]
return x[n]
```

Binomial Coefficient

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \\ 1 & \text{if } k = 0 \text{ or n} \end{cases}$$



1			
1	1		
1		1	
1			1
1			
1			

Change-Making Problem

- [Weiss] 7.6

For a currency with coin C1, C2, .. Cn (cents), what is the min number of coins needed to make K cents of change?

Example

- C = {1, 5, 10, 20, 50}
- K = 76 cents
- Give 4 coins 50 + 20 + 5 + 1 = 76

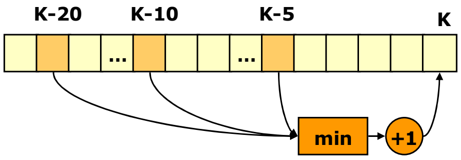
Some students have noticed that greedy method works on this example. They are correct. However, greedy does not work on all cases. Suppose C = {1,5,10,21,50} and K = 63, then greedy will give 5 coins, but the optimal solution is 3 coins.

Formulation

- To make a change of K cents, either
  - make a change of (K-50) cents, or
  - make a change of (K-20) cents, or
  - make a change of (K-10) cents, or
  - make a change of (K-5) cents, or
  - make a change of (K-1) cents
- Number of coins for K =
  1 + minimum of all the above choices

Dynamic Programming

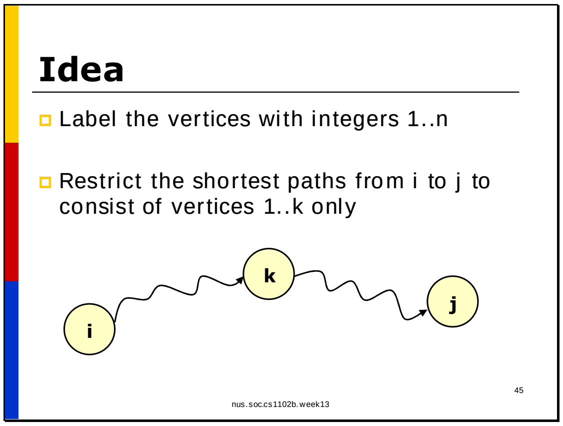
$$coinUsed(K) = 1 + \min_i \{coinUsed(K - C_i)\}$$



All Pair Shortest Path

- Execute Dijkstra's Algorithm |V| times
- Running Time: O(V(V+E)log V)
- Floyd-Warshall Algorithm: O(V^3)

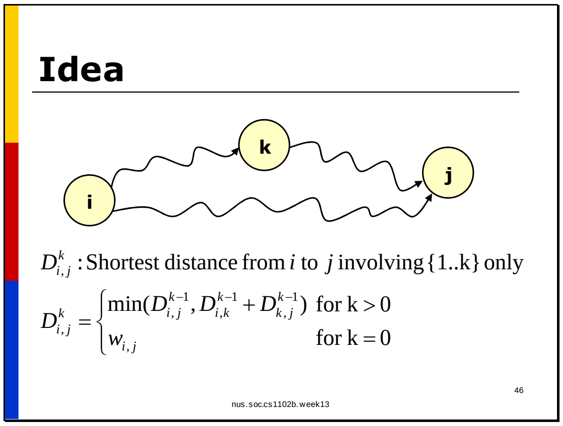
Another example of dynamic programming is all-pair shortest path where we are interested in calculating the shortest path between every pair of vertices. One obvious solution is to execute Dijkstra's algorithm from different source vertices. But the running time would be O((V+E)V log V). A dynamic programming solution runs in only O(V^3) time.



Idea

- Label the vertices with integers 1..n
- Restrict the shortest paths from i to j to consist of vertices 1..k only

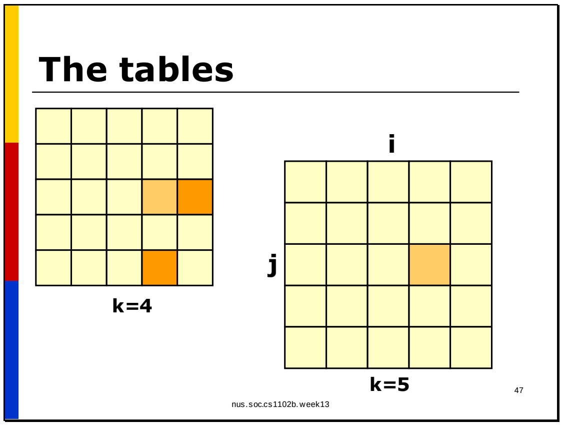
The idea behind Floyd-Warshall algorithm is this: Let the vertices be labeled from 1 to n. We find shortest paths between any two vertices with the restriction that the vertices on the shortest path (excluding the end points) can only consists of vertices from 1 to k. We then relax (nothing to do with Dijkstra's relax() operation!) the restriction so that the shortest paths can include vertices from 1 to (k+1).



Idea

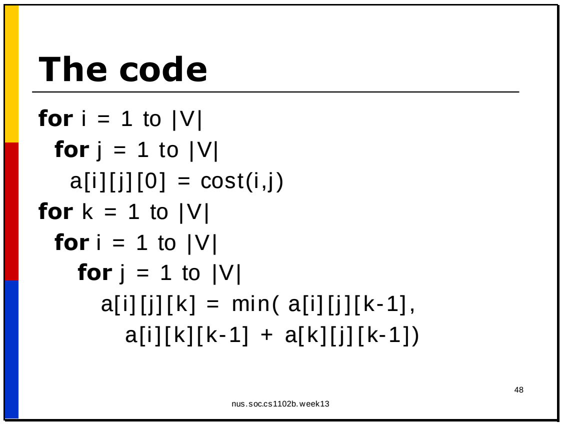
$D^k_{i,j}$: Shortest distance from i to j involving {1..k} only

$$D^k_{i,j} = \begin{cases} \min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}) & \text{for k} > 0 \\ w_{i,j} & \text{for k} = 0 \end{cases}$$



The tables

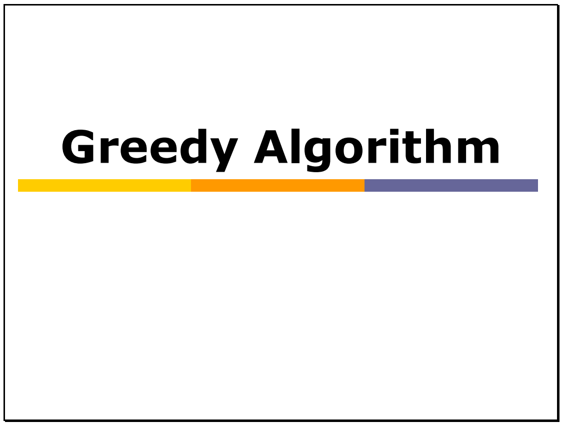
To fill out an entry in the table k, we make use of entries for table k-1, For example, to calculate $D^5_{4,3,}$ (column 4 row 3 in table 5), we look at $D^4_{4,3}$, and the sum of $D^4_{4,5}$ and $D^4_{5,3.}$ We take the smaller of the two values and fill in $D^5_{4,3.}$



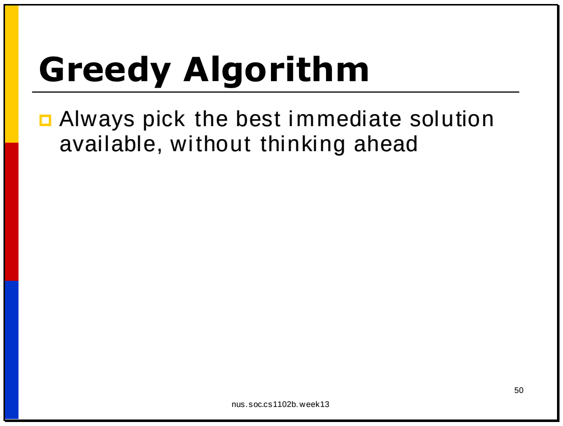
The code

```
for i = 1 to |V|
  for j = 1 to |V|
    a[i][j][0] = cost(i,j)
for k = 1 to |V|
  for i = 1 to |V|
    for j = 1 to |V|
      a[i][j][k] = min( a[i][j][k-1],
        a[i][k][k-1] + a[k][j][k-1])
```

The pseudo code above only gives us the distances of the shortest path? How can you modify the code so that we can recover the shortest paths?



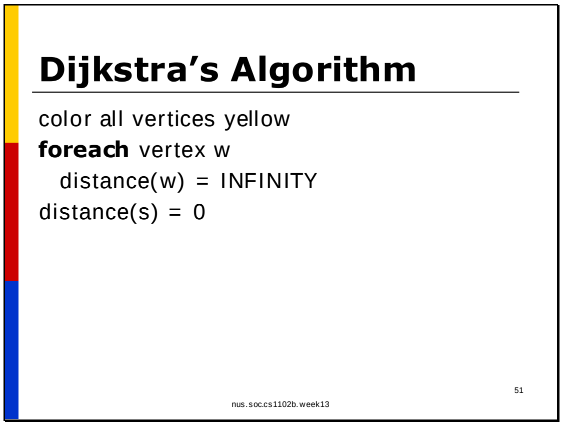
Greedy Algorithm



Greedy Algorithm

- Always pick the best immediate solution available, without thinking ahead

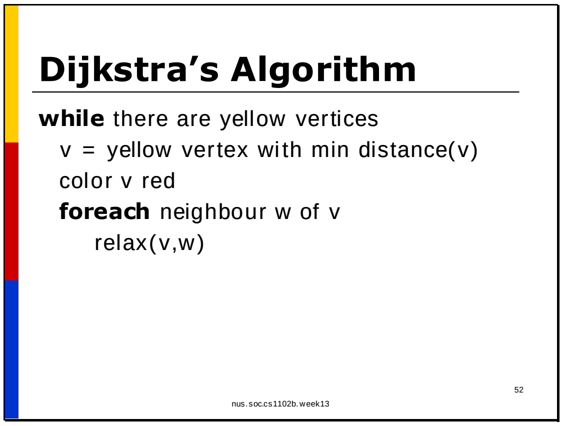
The last algorithm design paradigm is greedy algorithm. Greedy Algorithm always pick the best immediate solution available, without looking ahead.



Dijkstra’s Algorithm

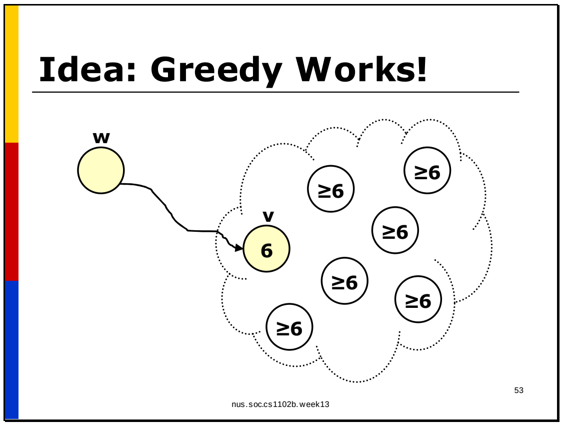
```
color all vertices yellow
foreach vertex w
  distance(w) = INFINITY
distance(s) = 0
```

One example is Dijkstra’s algorithm. We always pick the vertex with the shortest distance so far and conclude that we have found our shortest path.



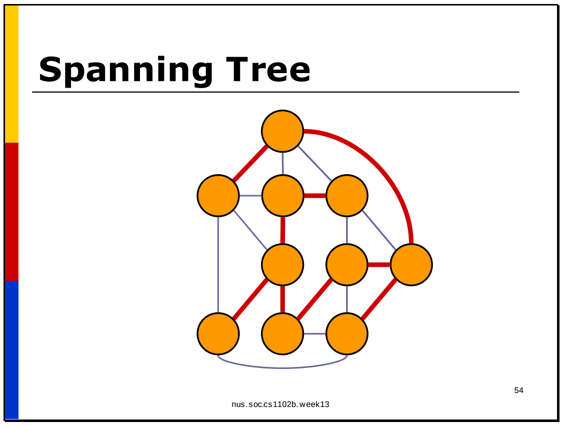
Dijkstra’s Algorithm

```
while there are yellow vertices
  v = yellow vertex with min distance(v)
  color v red
  foreach neighbour w of v
     relax(v,w)
```



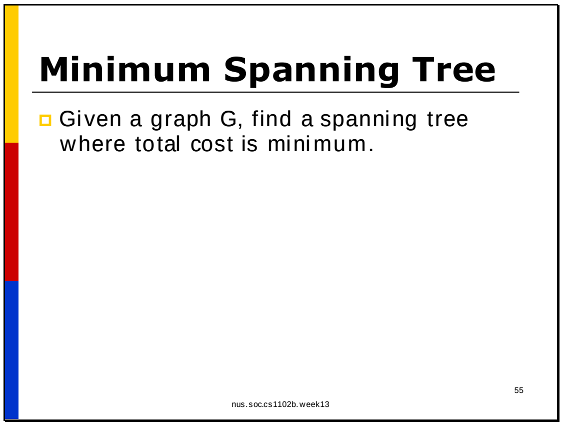
Idea: Greedy Works!

Greedy works in this case.



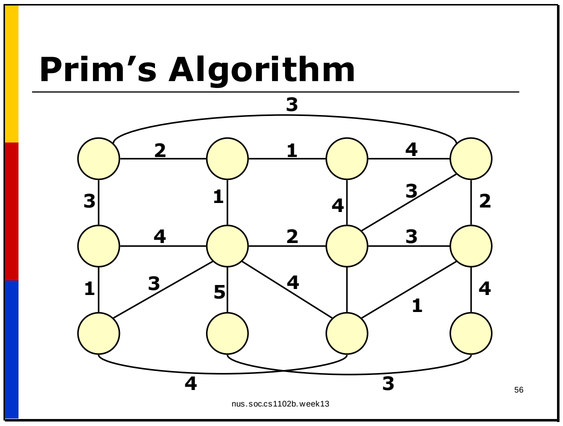
Spanning Tree

Another classic example of greedy graph algorithm is Prim's algorithm for finding Minimum Spanning Tree. A spanning tree is a set of edges that connects every vertex but yet does not form a cycle. The minimum spanning tree problem (or MST) is the problem of finding a spanning tree where the total cost of the edges is minimal.



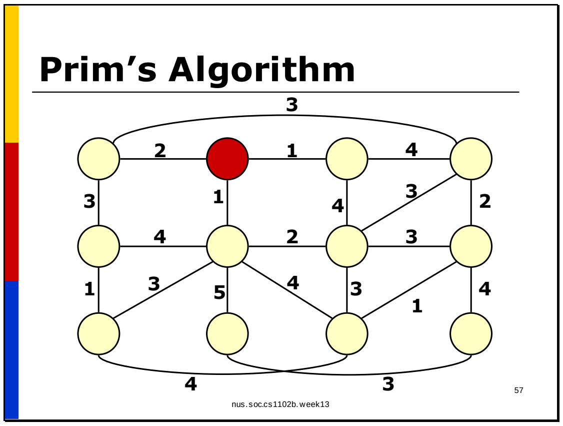
Minimum Spanning Tree

- Given a graph G, find a spanning tree where total cost is minimum.

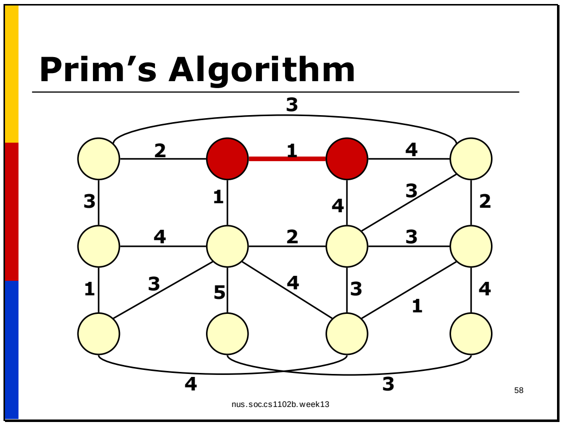


Prim's Algorithm

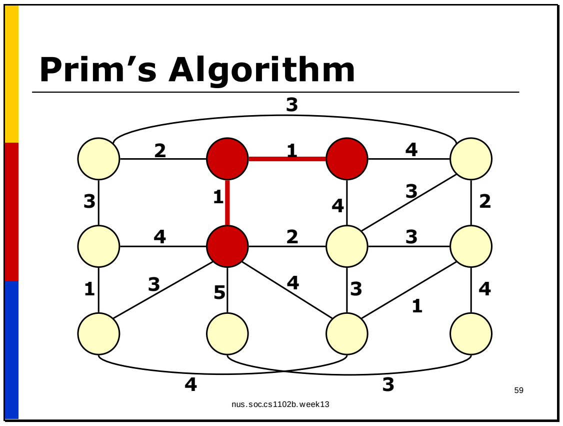
Prim's algorithm is greedy because at every iteration, it chooses an edge with minimum cost that does not form a cycle.



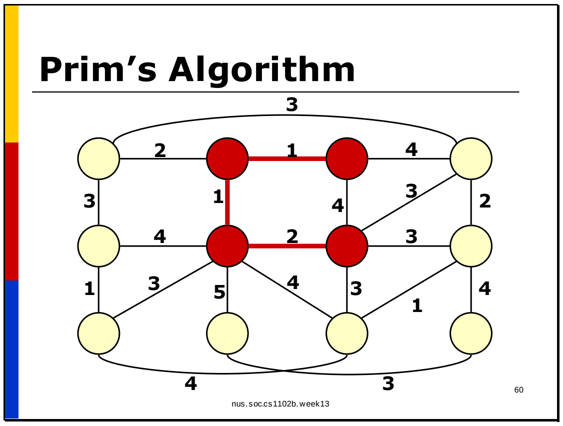
Prim’s Algorithm



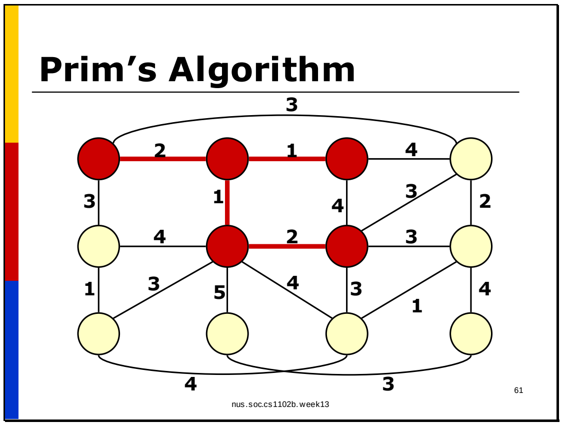
Prim’s Algorithm



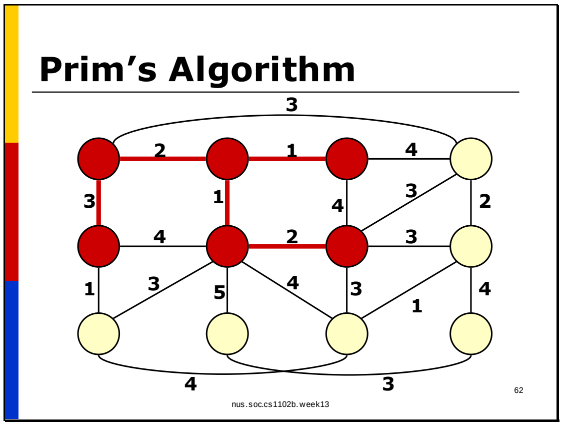
Prim’s Algorithm



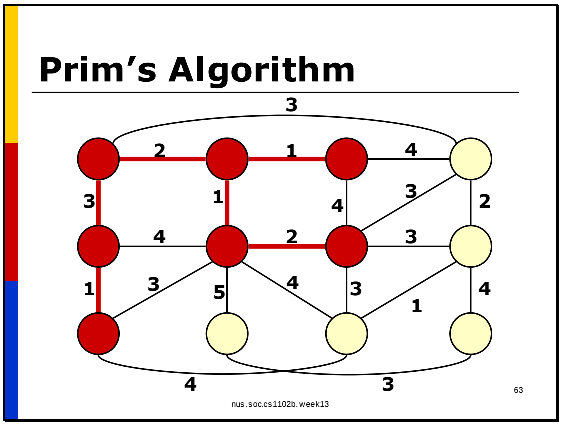
Prim’s Algorithm



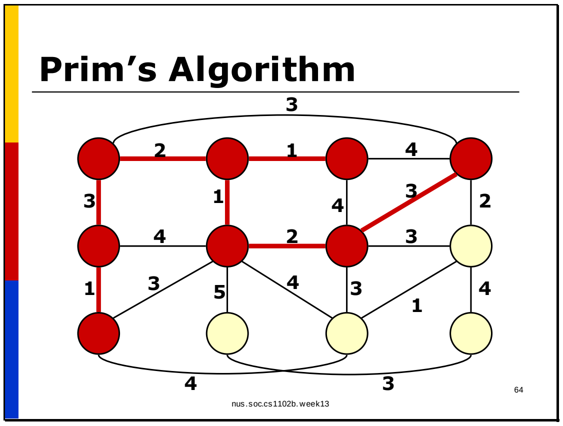
Prim’s Algorithm



Prim’s Algorithm

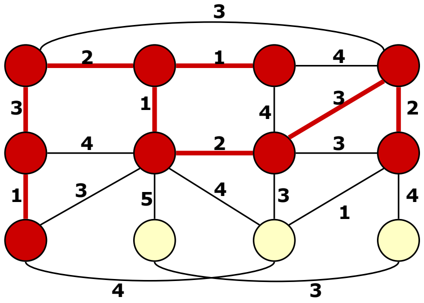


Prim’s Algorithm

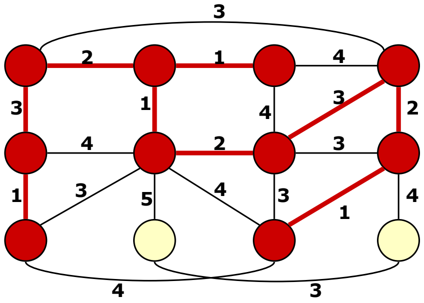


Prim’s Algorithm

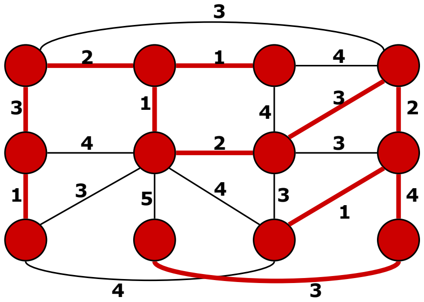
Prim’s Algorithm



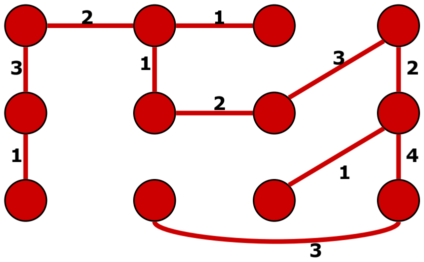
Prim’s Algorithm

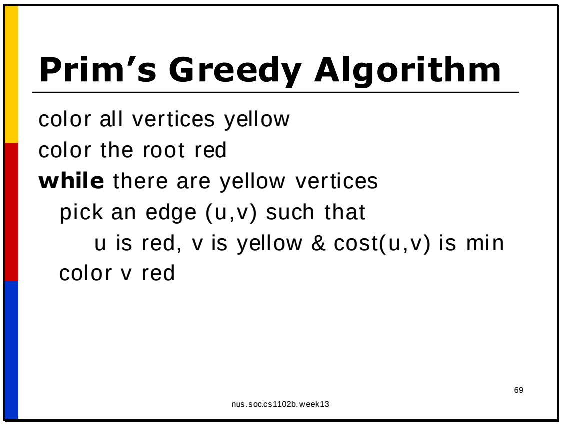


Prim’s Algorithm



Prim’s Algorithm

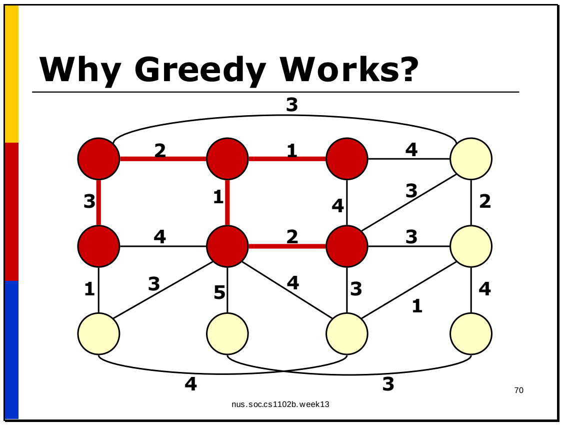




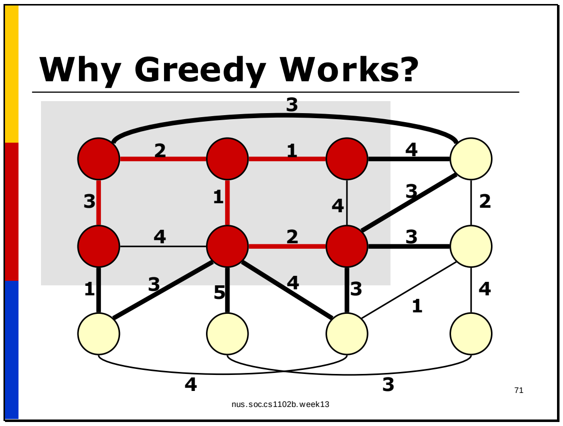
Prim’s Greedy Algorithm

```
color all vertices yellow
color the root red
while there are yellow vertices
  pick an edge (u,v) such that
     u is red, v is yellow & cost(u,v) is min
  color v red
```

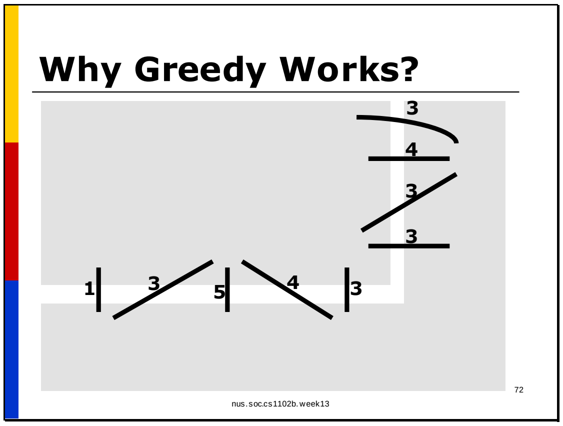
Note: we can pick any node to be the root.



Why Greedy Works?

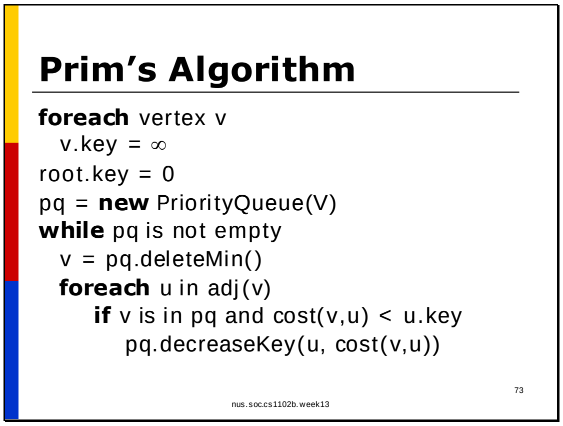


Why Greedy Works?



Why Greedy Works?

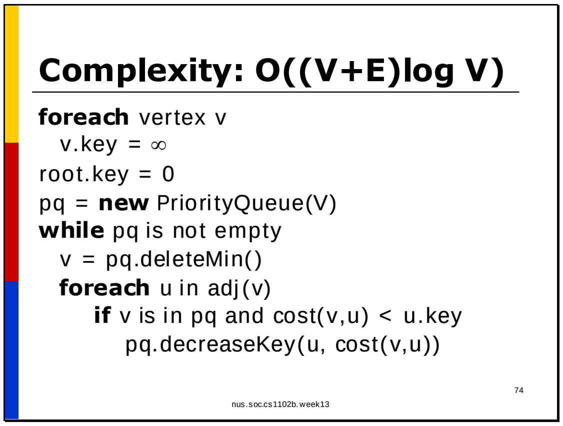
Greedy works in this case, because any spanning tree must include one of the edges that connects the yellow and the red vertices. The edge with minimum cost must be part of the minimum spanning tree.



Prim's Algorithm

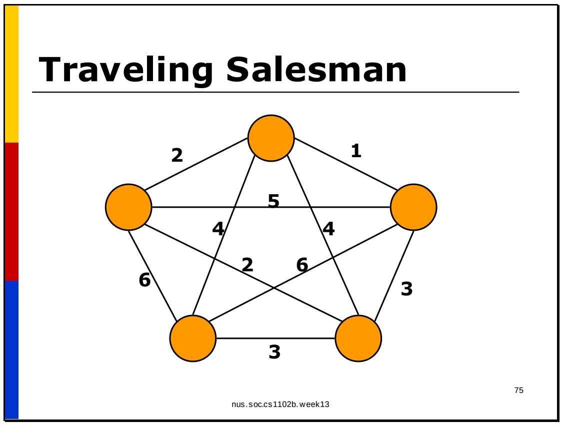
```
foreach vertex v
  v.key = ∞
root.key = 0
pq = new PriorityQueue(V)
while pq is not empty
  v = pq.deleteMin()
  foreach u in adj(v)
     if v is in pq and cost(v,u) < u.key
        pq.decreaseKey(u, cost(v,u))
```

We can implement Prim's algorithm using a priority queue as well, achieving the running time of O((V+E)log V).



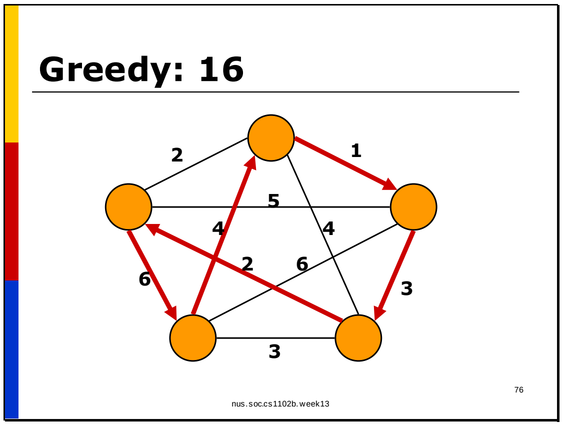
Complexity: O((V+E)log V)

```
foreach vertex v
  v.key = ∞
root.key = 0
pq = new PriorityQueue(V)
while pq is not empty
  v = pq.deleteMin()
  foreach u in adj(v)
     if v is in pq and cost(v,u) < u.key
        pq.decreaseKey(u, cost(v,u))
```



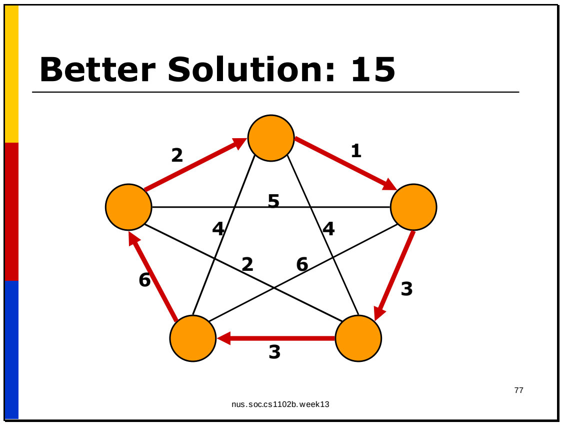
Traveling Salesman

Here is a problem that cannot be solved with greedy algorithm. The traveling salesman problem (TSP) can be stated as follows: given a graph, finds a simple cycle of |V| vertices with minimum cost. (i.e., find a tour that visits every vertex exactly once and return to the source with minimum cost.)

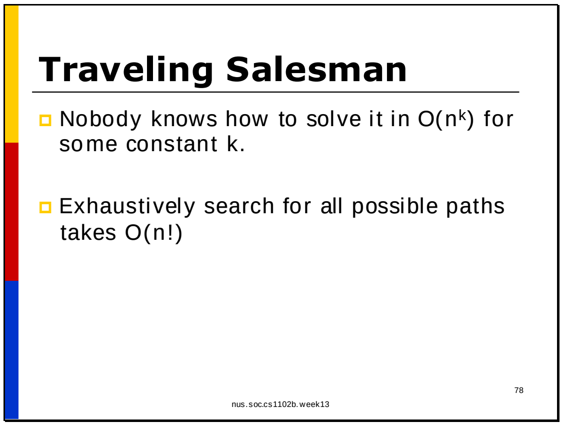


Greedy: 16

The greedy method will pick an outgoing edge to an unvisited vertex with minimum cost every time. This can land us in trouble, because we might be force to pick a very expensive edge later.



Better Solution: 15



Traveling Salesman

- Nobody knows how to solve it in $O(n^k)$ for some constant k.
- Exhaustively search for all possible paths takes O(n!)

The traveling salesman problem belongs to the class of problems known as NP. All the other problems (sorting, shortest path) that can be solved in $O(n^k)$ belongs to the class P. (NP stands for nondeterministic-polynomial and P stands for polynomial). No one knows how to solve problems in NP in $O(n^k)$ time, that is, no one knows if NP = P.