CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: THE GREEDY METHOD – PART I

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe another powerful algorithmic design technique, namely, the Greedy Method
- Explain what optimization problems and optimization techniques are
- Create and explore different greedy policies
- Develop greedy algorithms for several important optimization problems
- Prove non-optimality of some greedy solutions
- Select the right data structures for some greedy algorithms

OUTLINE

- Introduction to the greedy method
- Applying the greedy method to sorting
- Applying the greedy method to several basic problems
    - Optimal merge patterns
    - The knapsack problem
- A greedy algorithm for the Minimum Spanning Tree (MST) problem
- A greedy algorithm for the Single-Source Shortest Paths problem

THE GREEDY METHOD

-- BACKGROUND (1) --

- The greedy method is primarily an ***optimization*** technique
- An optimization problem is either a minimization problem or a maximization problem
- In a minimization problem, there are many solutions, each having a cost associated with it
- Solving a minimization problem means finding the solution that has minimum cost; such a solution is called a *minimum solution*
- In a maximization problem, there are many solutions, each having a profit associated with it, and the goal is to find a *maximum solution*, i.e., the solution with maximum profit

THE GREEDY METHOD

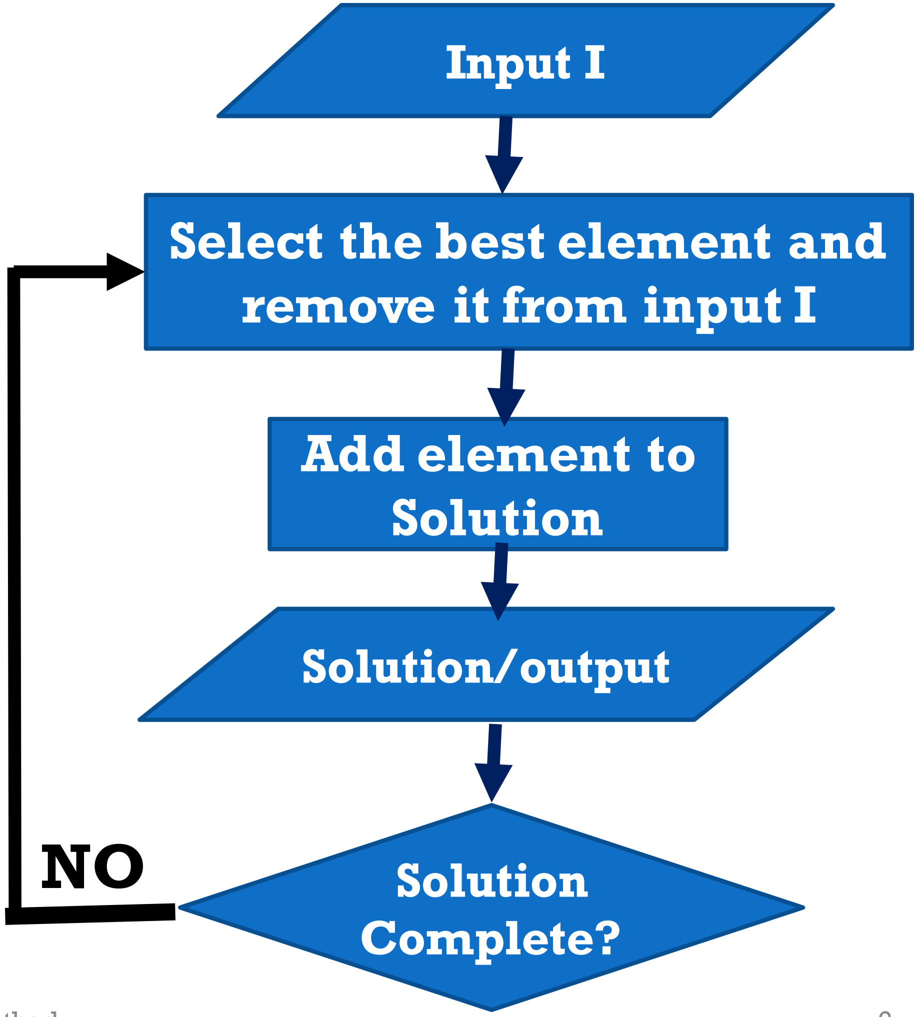
-- GENERAL STRATEGY --

- For greedy to apply, the solution must consist of a set/sequence of elements
  - The greedy method finds the solution one element after another: the i^{th} element in the i^{th} step.
- General strategy of the greedy method:
  - At every step,
    - select the **best** element **from** the **remaining input**,
    - delete it from the input, and put it in the output.
- What is "best"?
  - The answer is given by the algorithm designer, and
  - varies from problem to problem, and algorithm to algorithm

The statement "select the best at every step", along with the definition of "best", are referred to as the ***greedy policy.***

THE GREEDY METHOD
-- TEMPLATE --

```
Template Greedy (input I)
 begin
   Optional: Process I for faster exec
   while (solution is not complete) do
           Select the best element x in the
                        remaining input I;
           Remove x from the input I;
           Put x next in the output;
   endwhile
end
```



FIRST APPLICATION
-- GREEDY SORT --

- Greedy sorting
  - The selection policy: select the minimum of the remaining input
  - That is, best=minimum
  - So the method becomes:
    - While there is input, find the minimum of the remaining input, remove it from the input, and put it next in the output.
- Notes:
  - Intuitively, sorting is not an optimization problem, but still a simple illustration of applying the greedy method
  - Question to think about: can you formulate sorting as optimization?

GREEDY SORT

-- SELECTION SORT --

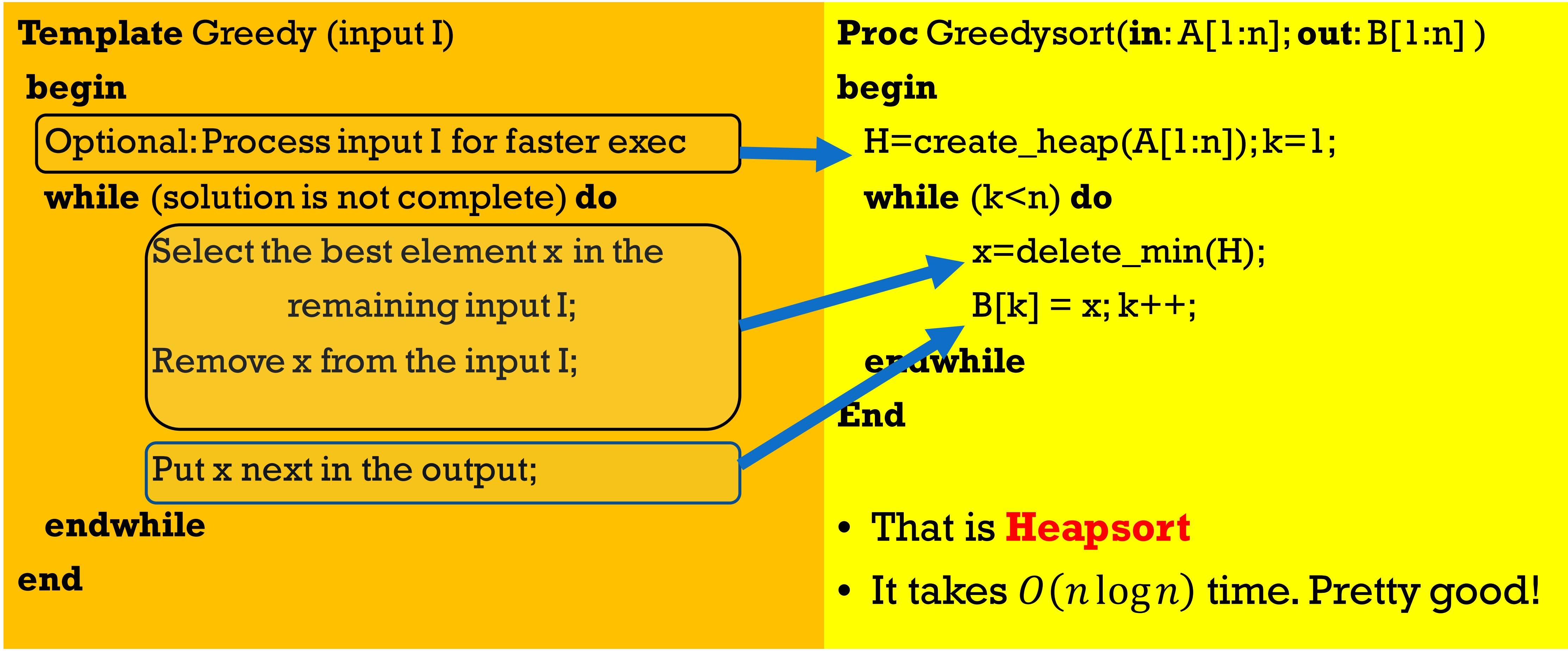
- If you implement the greedy policy of finding the minimum by always scanning the remaining input, the resulting algorithm is a well-known sorting algorithm, called ***Selection Sort***.
- It takes $O(n^2)$ time, so it is not the best sorting algorithm
- **Question**: Can you give a faster implementation of the greedy policy of finding (and deleting) the minimum of the remaining input?

GREEDY SORT
-- BETTER IMPLEMENTATION--

- Since in greedy sorting you need to repeatedly find and delete the min, it makes sense to build and use an appropriate data structure
- Which standard data structure do that? Think delete-min()!!
- Answer: min-heaps
    - Which leads to … (see next slide)

GREEDY SORT

-- HEAPSORT --



```
Template Greedy (input I)
 begin
   Optional: Process input I for faster exec
   while (solution is not complete) do
         Select the best element x in the
               remaining input I;
         Remove x from the input I;
         Put x next in the output;
   endwhile
end
```

```
Proc Greedysort(in: A[1:n]; out: B[1:n] )
begin
   H=create_heap(A[1:n]);k=1;
   while (k<n) do
         x=delete_min(H);
         B[k] = x; k++;
   endwhile
End
```

- That is **Heapsort**
- It takes $O(n \log n)$ time. Pretty good!

LESSONS LEARNED SO FAR

- The same greedy policy on the same problem can be implemented in different ways
- Some implementations can be much faster (e.g., min-heap leads to faster greedy sorting)
- Pre-processing the input can be very helpful
  - for faster implementation, and
  - sometimes for making greedy possible (to be seen later)
- More lessons to come (about the greedy method)

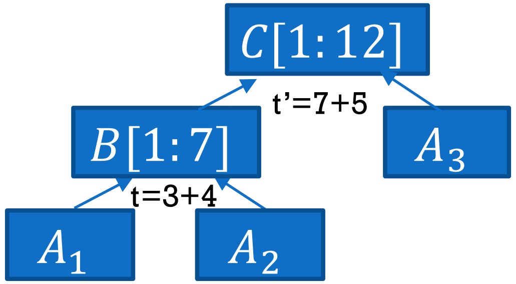
SECOND APPLICATION

-- OPTIMAL MERGE PATTERNS --

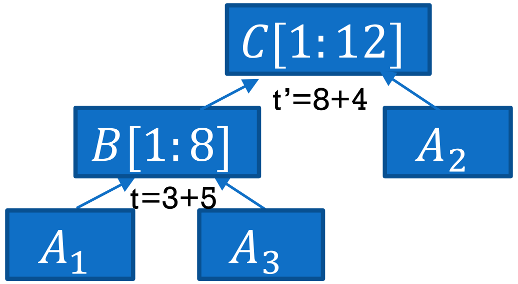
- **Input**: n sorted arrays $A_1[1: L_1], A_2[1: L_2], \ldots, A_n[1: L_n]$
- **Output**: The whole input combined into a single sorted array
- **Task**: Find a greedy algorithm that merges $A_1, \ldots, A_n$ pairwise into a single sorted array, taking a minimum # of steps

OPTIMAL MERGE PATTERNS
-- AN EXAMPLE--

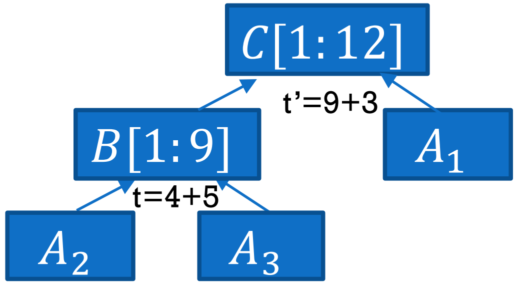
- Example: Take three sorted arrays $A_1[1:3], A_2[1:4], A_3[1:5]$
- Three ways (i.e., pairing sequences) to merge:



Time: (3+4)+(7+5)=19



Time: (3+5)+(8+4)=20



Time: (4+5)+(9+3)=21

- Although all the different pairings lead to the same final output C[1:12], they take different amounts of time
- Interested in an algorithm that finds the fastest way

OPTIMAL MERGE PATTERNS

-- A GREEDY ALGORITHM --

- **Greedy policy**: at every step, must choose the "best" pair (of arrays) to merge
- **Best**: pair of the two shortest arrays
- **Greedy policy**: Select the two shortest arrays to merge next
- **Optimality question**: Is this greedy method guaranteed to give us an optimal solution, i.e., the sequence of pairings that take the least amount of time?
- **Answer**: Yes, the greedy solution for this problem is always optimal
- **Proof**: It will not be provided, but you can work on it as an exercise

OPTIMAL MERGE PATTERNS

-- A GREEDY ALGORITHM: IMPLEMENTATION --

- The greedy algorithm is a loop where in every iteration:
  - we need to **find** the two **smallest-length** arrays,
  - remove them from the input, and
  - replace them, i.e., **insert** back to the input, with a **new** array of new length (sum of the previous two lengths)
- These operations are repeated over and over, so?
- So, better design/use a data structure of array lengths so we can find & delete the smallest-length very fast, and insert very fast

CYK

- What data structure best meet those needs?
  a. Stack?
  b. Queue?
  c. Binary search tree?
  d. Min-heap?
- Time complexity of the greedy optimal merge pattern?
  a. $O(n^2)$
  b. $O(n \log n)$
  c. $O(n)$

THIRD APPLICATION

-- THE KNAPSACK PROBLEM --

- **Input**:
    - Items: 1, 2, 3, … , n
    - Weights: W_1 W_2 W_3 … , W_n
    - Prices: P_1 P_2 P_3 … , P_n
    - Capacity: C

P_i is the price of the whole item i, not the price per pound

- **Output**: How much of item i to take such that the total of the taken weights is $\leq C$, and the total of the prices of the taken items is maximized.

  More formally:
    - $\forall i,$ let x_i be the fraction (between 0 and 1) of item i to take. **Ex:** if $x_i = \frac{1}{3}$, that means we're taking $\frac{1}{3}$ of item i, and so we're taking weight $\frac{W_i}{3}$ $(= x_i W_i)$ and price $\frac{P_i}{3}$ $(= x_i\ P_i)$
    - Output: Find $x_1, x_2, \dots, x_n$ to maximize $\Sigma_{i=1}^{n} x_i P_i$ such that $\Sigma_{i=1}^{n} x_i W_i \leq C$
- **Task**: Write a greedy algorithm for solving this problem

THE KNAPSACK PROBLEM
-- A GREEDY ALGORITHM: FIRST ATTEMPT --

- Solution is a sequence $x_1, x_2, \ldots, x_n$
- At step i, compute the value x_i
- **Greedy policy 1**: Select the item with the smallest weight from among the remaining items. If it still fits on the "sack" (of capacity C), take all of that item; otherwise, just take the largest fraction of it that fills the sack.
- **Rationale**: Since we're limited by total weight (C) that we can carry, if we always choose smallest-weight items, we end up with a lot of items, hoping that would maximize our profit.
- **Exercise**: Show that this greedy policy doesn't guarantee an optimal solution

HOW TO PROVE A GREEDY SOLUTION NOT OPTIMAL

- Method for proving non-optimality (by a counter-example)
  1. Construct an actual input (of size as small as possible)
  2. Find the greedy solution from that input
  3. Manually, find a better solution
- If you succeed in finding a better solution than the greedy solution, then obviously the greedy solution is non-optimal
- Note: the manual solution you find need not be optimal, i.e., <u>best</u>; it only needs to be <u>better</u> than the greedy solution.

LESSONS LEARNED SO FAR

- The same greedy policy on the same problem can be implemented in different ways
- Some implementations can be much faster (e.g., min-heap leads to faster greedy sorting)
- Pre-processing the input can be very helpful
- The greedy method does not always guarantee optimality (as in some greedy policies for the knapsack problem)
- To prove non-optimality, use counter-examples
- More lessons to come (about the greedy method)

THE KNAPSACK PROBLEM
-- A GREEDY ALGORITHM: SECOND ATTEMPT --

- **Greedy policy 2**: Select the item with the largest price.

  Again, if it still fits in the sack (of capacity C), take all of that item; otherwise, just take the largest fraction of it that fills the sack.

- **Rationale**: by taking as many most expensive items as fit on the sack, we hopefully end up with maximum profit

- **Exercise**: Show that this greedy policy doesn't guarantee an optimal solution

THE KNAPSACK PROBLEM

-- A GREEDY ALGORITHM: THIRD ATTEMPT --

- **Greedy policy 3**: Select the item with the highest price per unit weight, i.e., with the highest $\frac{P_i}{W_i}$, out of the remaining items.

  Again, if it still fits on the sack (of capacity C), take all of that item; otherwise, just take the largest fraction of it that fills the sack .

- **Claim**: This policy guarantees that the greedy solution of the knapsack problem is always optimal

- **Proof**: It will not be given in this course.

THE KNAPSACK PROBLEM

-- A GREEDY ALGORITHM: AN EXAMPLE --

- Example:

i:	1	2	3	4	5
P_i:	5	9	4	8	1
W_i:	1	3	2	2	2

C = 4

$\frac{P_i}{W_i}$:	5	3	2	4	1/2

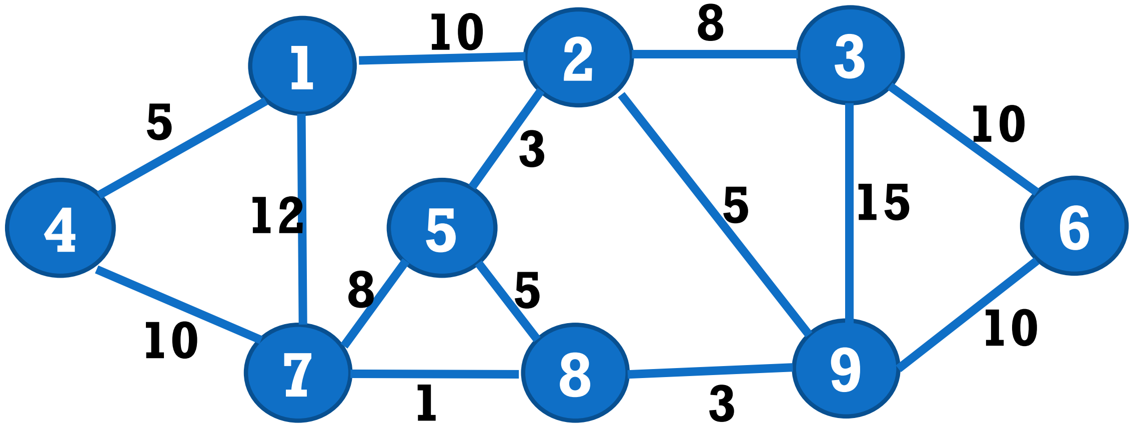
- Solution:
  - 1st item to select: item 1, so $x_1 = 1, x_1 W_1 = 1$ Weight so far=1
  - 2nd item to select: item 4, so $x_4 = 1, x_4 W_4 = 2$ Weight so far=3
  - 3rd item to select: item 2, so $x_2 = \frac{1}{3}, x_2 W_2 = \frac{3}{3} = 1$ Weight so far=4=C
  - Profit (i.e., total price taken): $x_1 P_1 + x_4 P_4 + x_2 P_2 = 1 \times 5 + 1 \times 8 + \frac{1}{3} \times 9 = 16$
  - Note that $x_3 = 0$ and $x_5 = 0$.

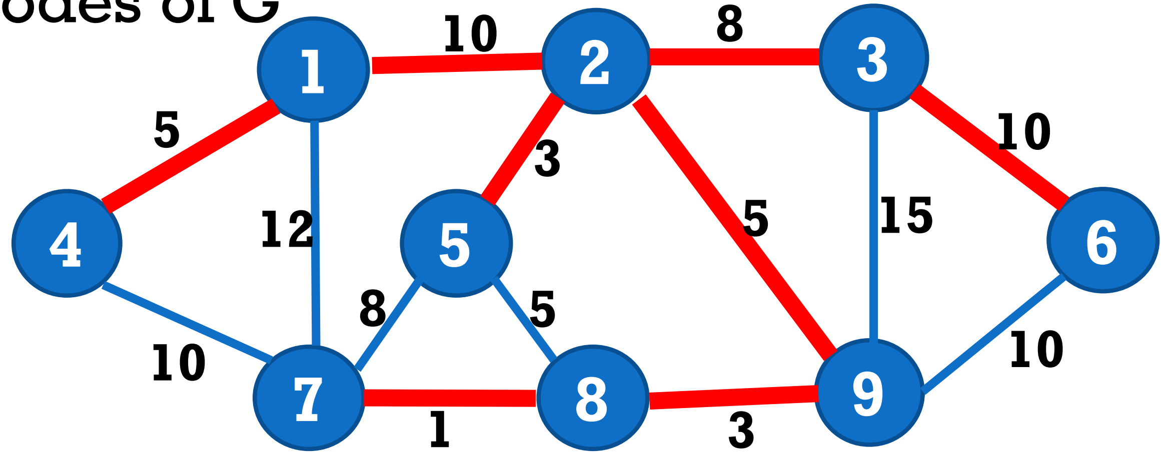
LESSONS LEARNED SO FAR

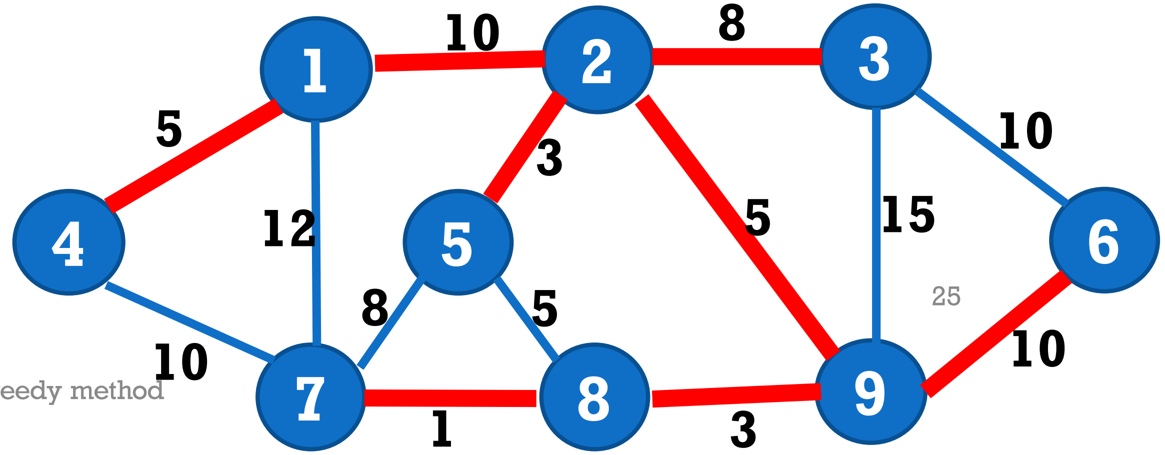
- The same greedy policy on the same problem can be implemented in different ways
- Some implementations can be much faster
- Pre-processing the input can be very helpful
- The greedy method does not always guarantee optimality
- To prove non-optimality, use counter-examples
- For the same problem, one can formulate different greedy policies, some non-optimal and some optimal
- More lessons to come (about the greedy method)

THE MINIMUM SPANNING TREE PROBLEM
-- PRELIMINARY DEFINITIONS (1/2) --

- **Definition**: A ***spanning tree*** T of a graph G is a tree that has all the nodes of G such that every edge in T is an edge in G
- "Spanning" means "*including all*" the nodes of G



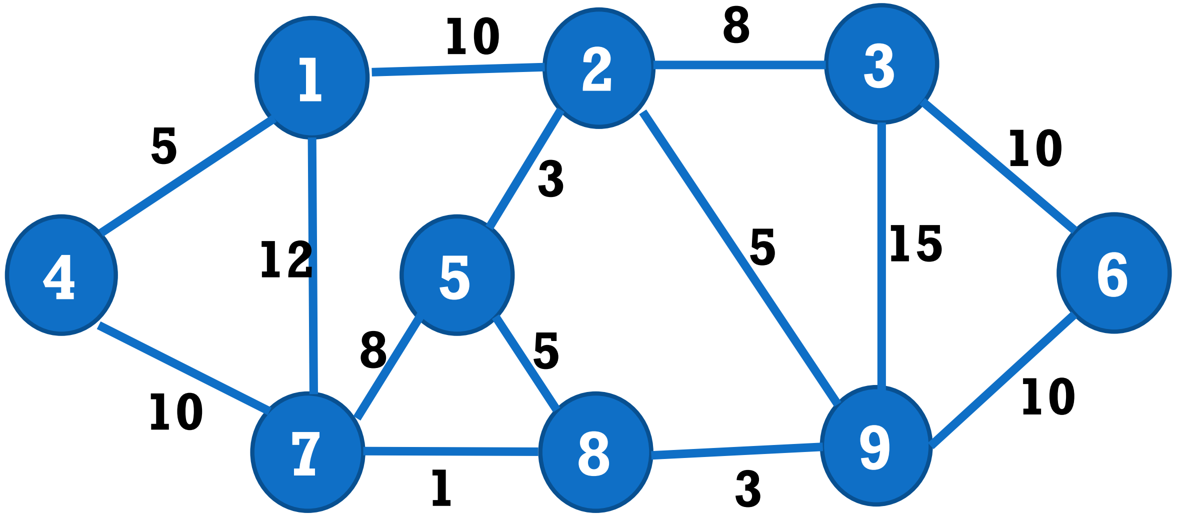


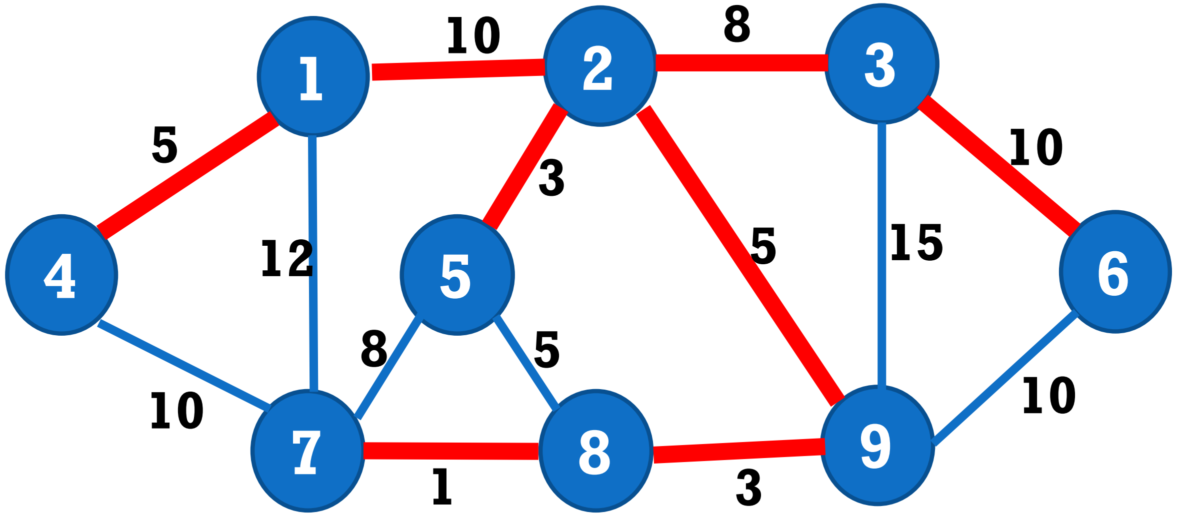


- G can have many spanning trees

THE MINIMUM SPANNING TREE PROBLEM
-- PRELIMINARY DEFINITIONS (2/2) --

- **Definition**: If G is weighted, i.e., the edges have weights, then the **weight of T** is the **sum of the weights of its edges**
- **Definition:** A ***minimum spanning tree*** (MST) of a weighted graph G is a spanning tree that has minimum weight among all spanning trees of G.





THE MINIMUM SPANNING TREE PROBLEM

-- STATEMENT OF THE PROBLEM--

- **Input**: A weighted graph G, typically represented by a weight matrix W[1:n,1:n], where for non-edges (i,j): $W[i,j]=\infty$
- **Output**: A minimum spanning tree in G
- **Task:** Develop a greedy algorithm that finds a MST in any input weighted graph

GREEDY ALGORITHM FOR THE MST PROBLEM
-- KRUSKAL'S ALGORITHM --

- **Solution as a set of elements:** the elements are the edges of the tree
- The Greedy method will find the tree one edge at a time
- **Greedy policy:** At every step, **select (and remove) the min-weight edge** out of the remaining edges in the graph
- Can we always add a selected edge to the growing tree T?
  - No, not always: if the selected edge would create a cycle in T, it must not be added (recall that a tree has no cycles)
- **Adjustment to the greedy method:** if the min-weight edge creates a cycle in T, throw it out; else, add it to T

PSEUDOCODE OF KRUSKAL'S GREEDY MST ALGORITHM

```
Procedure ComputeMST(in: $W[1:n,1:n]$; out: T)    // non-edges $(i,j)$:  $W[i,j] = \infty$
begin
    Put in T all the n nodes and no edges;
    while T has less than n-1 edges do
        Select a min-weight edge e out of the remaining edges;
        Delete e from the graph;
        if (e does not create a cycle in T) then
            Add e to T;
        endif
    endwhile
end ComputeMST
```

ILLUSTRATION OF THE GREEDY MST ALGORITHM

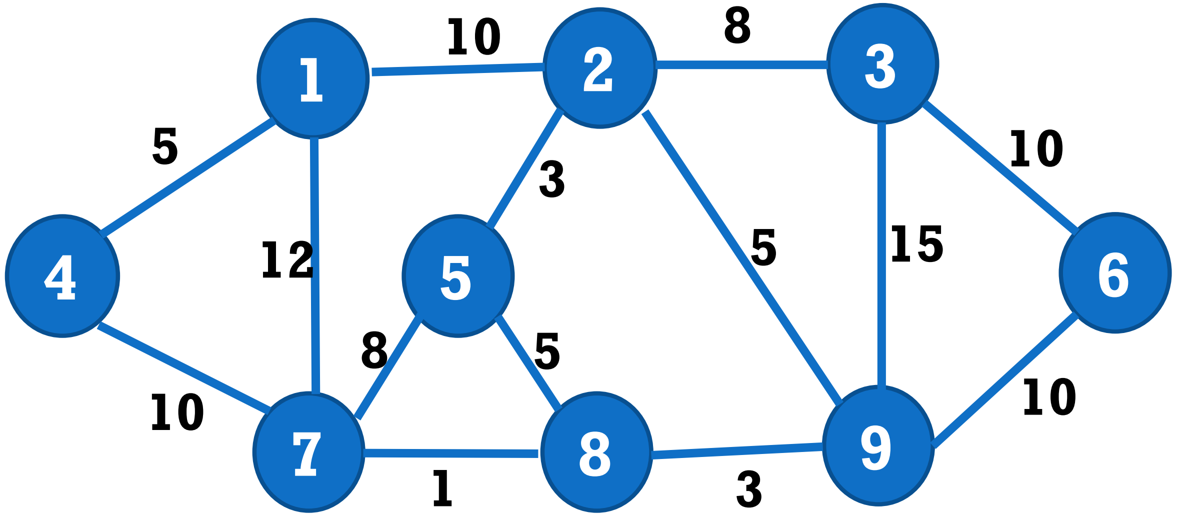
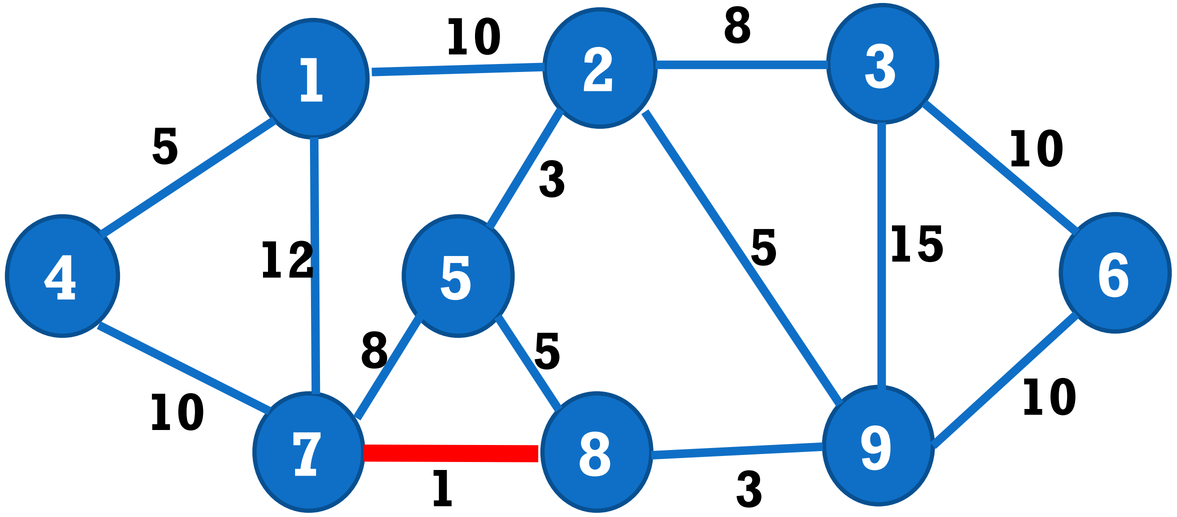
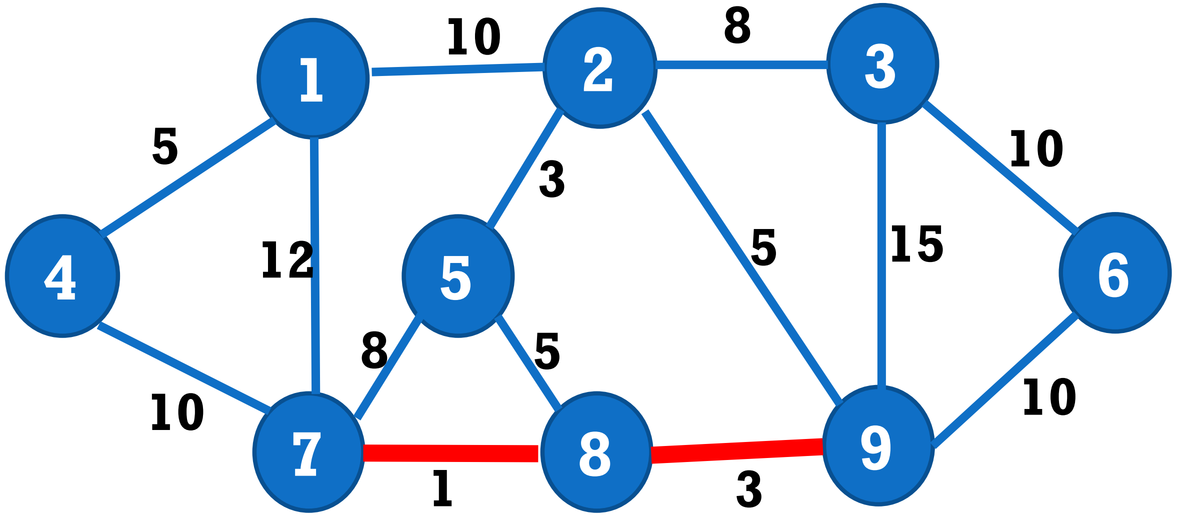


ILLUSTRATION OF THE GREEDY MST ALGORITHM



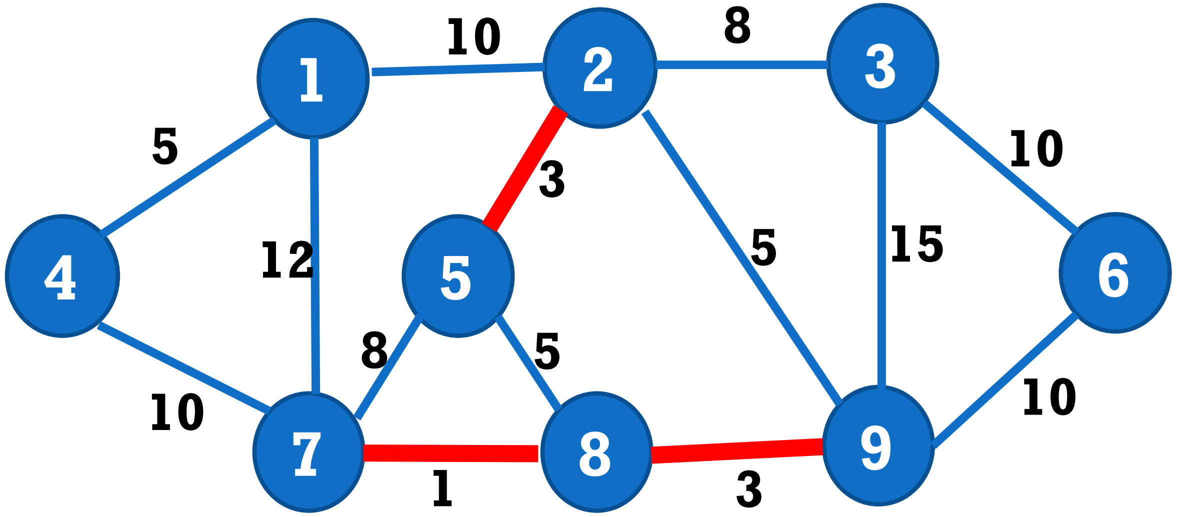
- Min edge: (7,8). No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



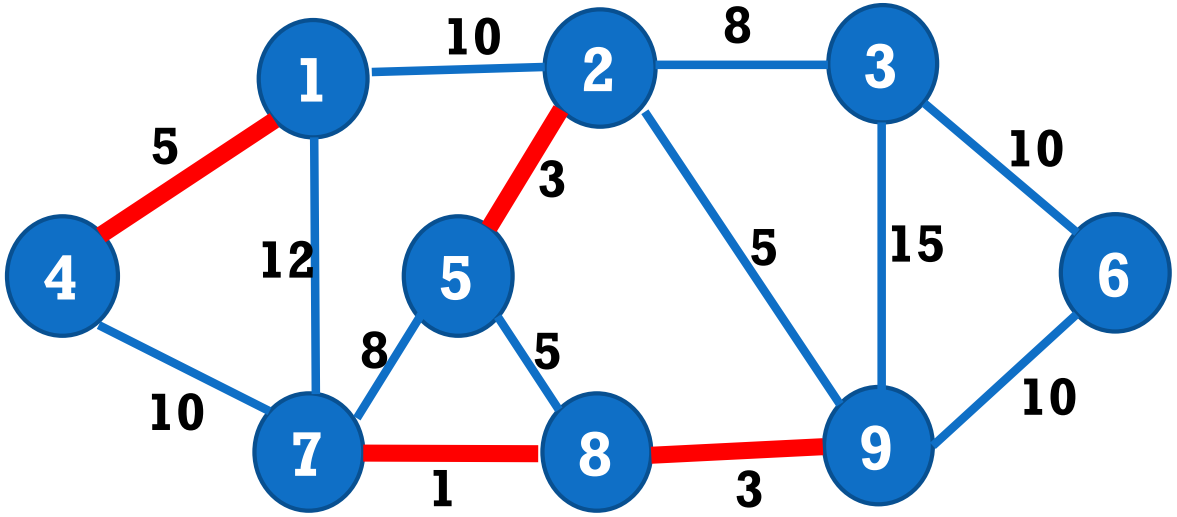
- Min edge: (8,9), (2,5). Pick (8,9). No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



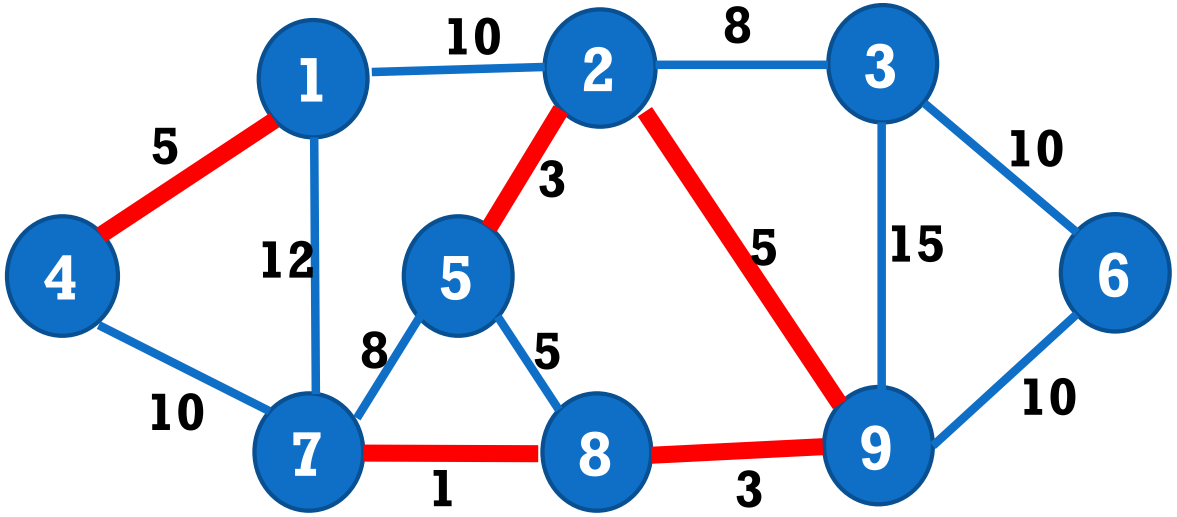
- Min edge: (2,5). No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



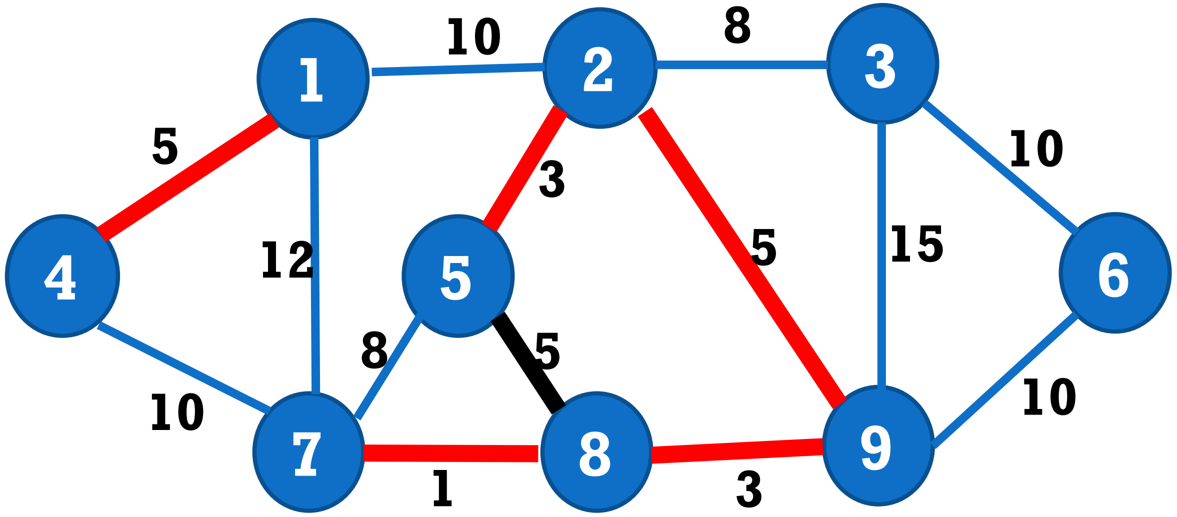
- Min edge: (1,4), (2,5), (5,8). Pick (1,4): No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



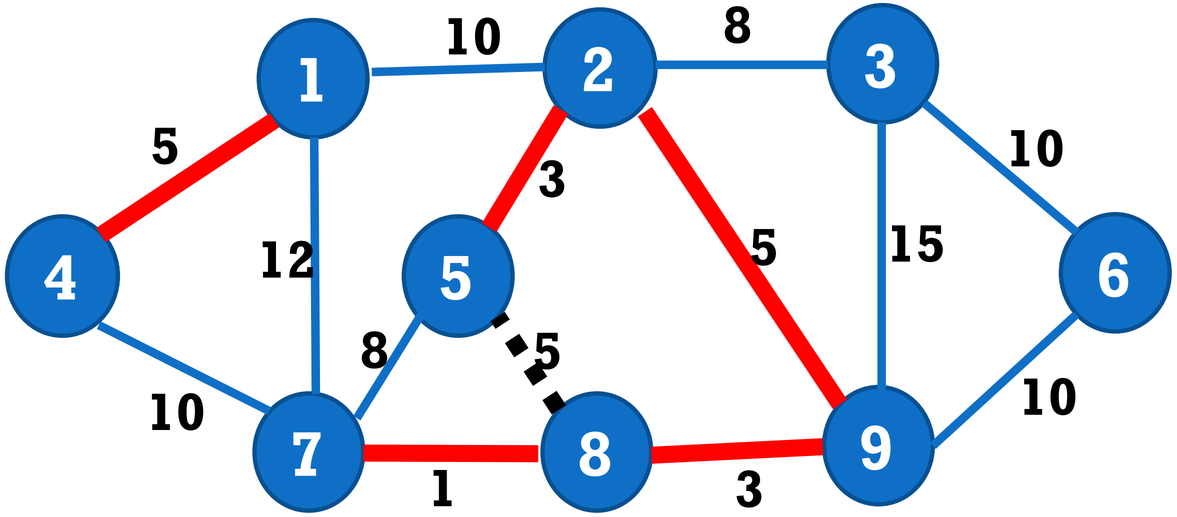
- Min edge: (2,5), (5,8). Pick (2,5). No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



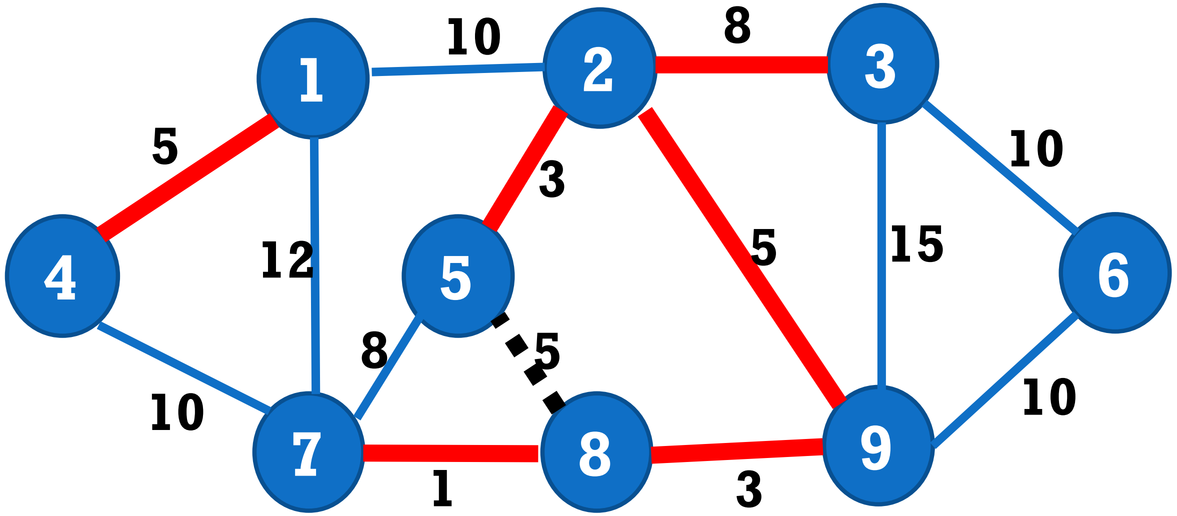
- Min edge: (5,8). Creates cycle

ILLUSTRATION OF THE GREEDY MST ALGORITHM



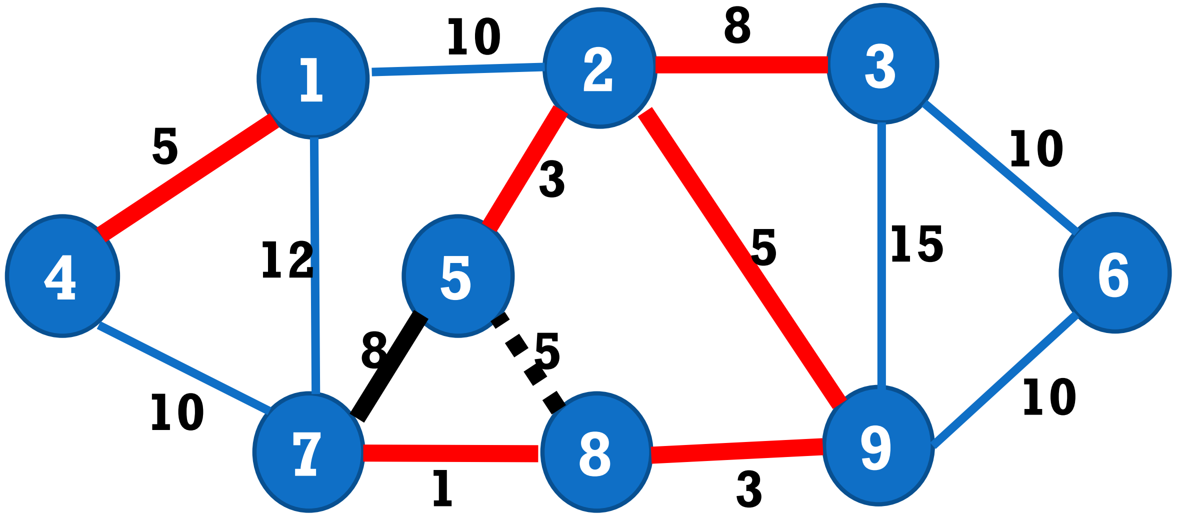
- Min edge: (5,8). Creates cycle => throw it out

ILLUSTRATION OF THE GREEDY MST ALGORITHM



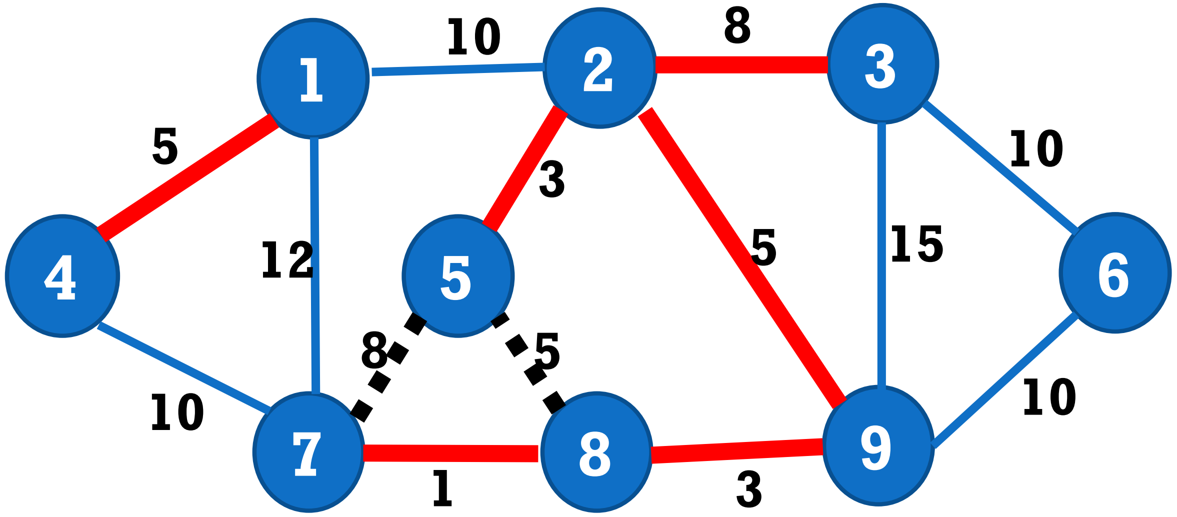
- Min edge: (2,3) and (5,7). Pick (2,3): No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



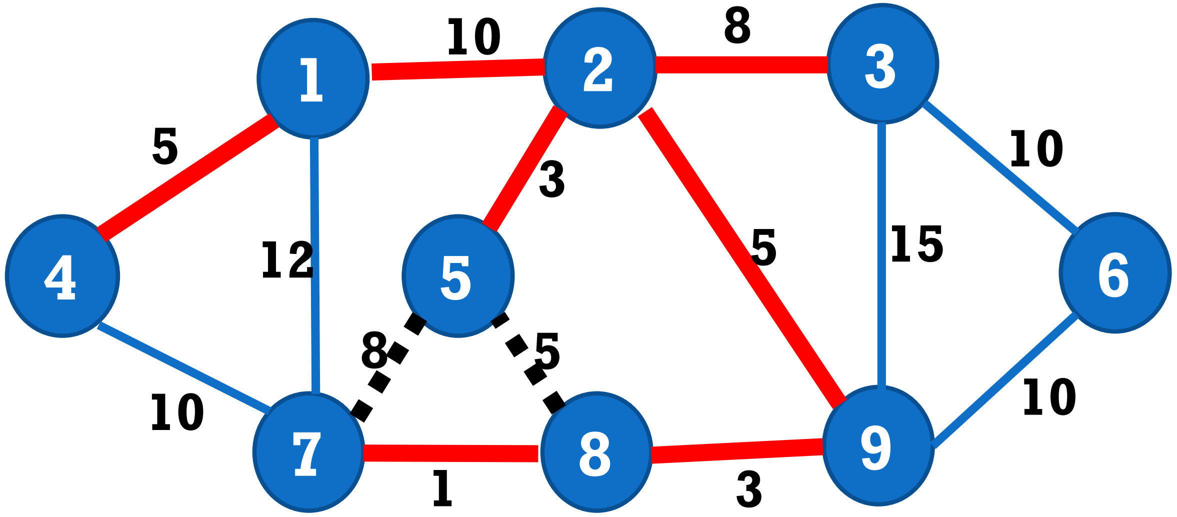
- Min edge: (5,7). Creates cycle

ILLUSTRATION OF THE GREEDY MST ALGORITHM



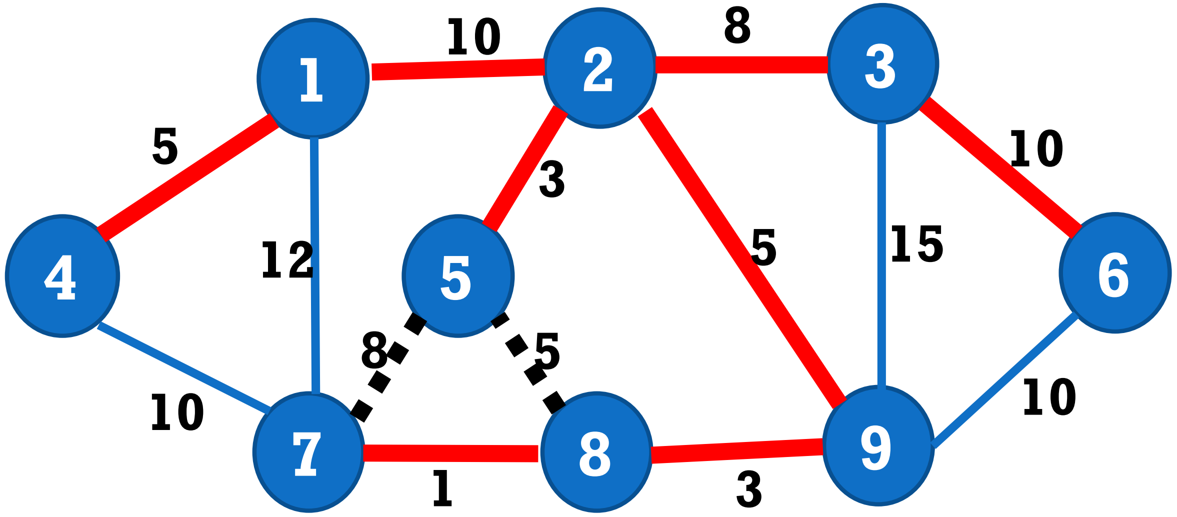
- Min edge: (5,7). Creates cycle => throw it out

ILLUSTRATION OF THE GREEDY MST ALGORITHM



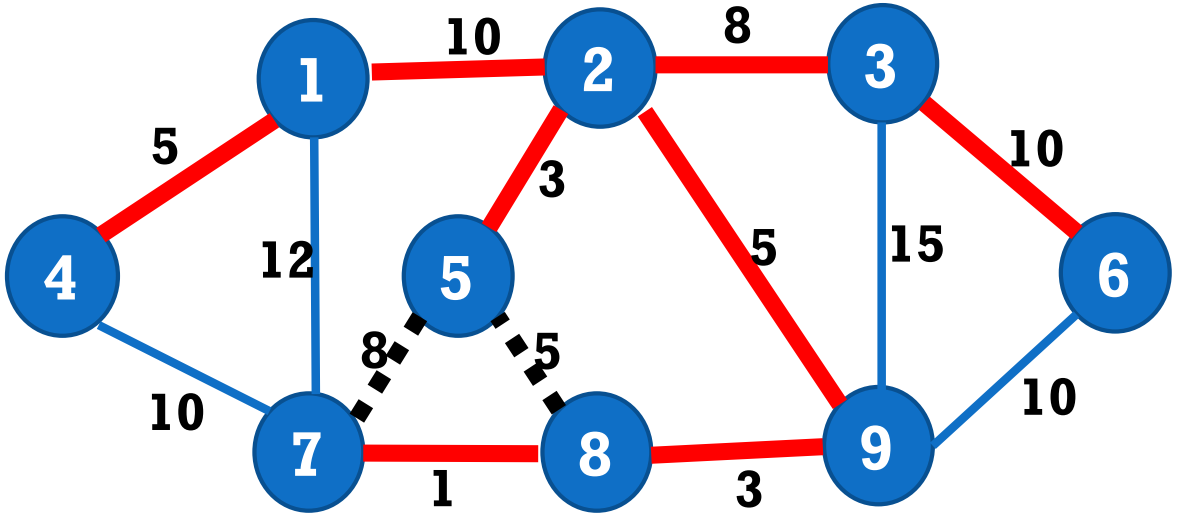
- Min edge: (1,2), (3,10), (4,7), 6,9). Pick (1,2): No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



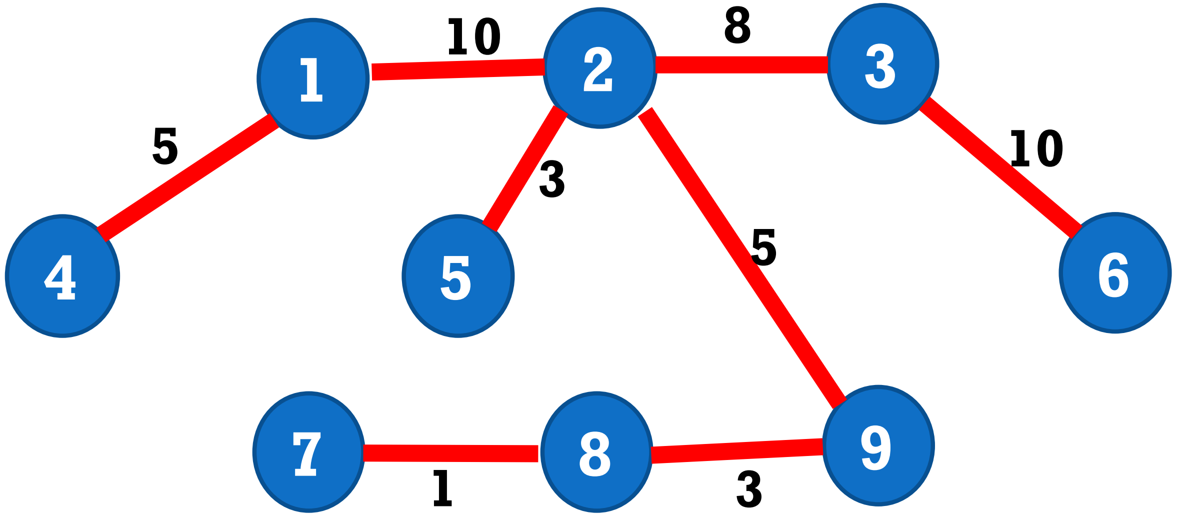
- Min edge: (3,6), (4,7), (6,9). Pick (3,6): No cycle => OK to add

ILLUSTRATION OF THE GREEDY MST ALGORITHM



- Tree completed (got 8 edges)

ILLUSTRATION OF THE GREEDY MST ALGORITHM



- This is the spanning tree produced by the greedy algorithm

THE GREEDY MST ALGORITHM
-- IMPLEMENTATION ISSUES --

```
Procedure ComputeMST(in: $W[1:n,1:n]$; out: T)    // non-edges $(i,j)$: $W[i,j]=\infty$
begin
    Put in T all the n nodes and no edges;
    while T has less than n-1 edges do
        Select a min-weight edge e out of the remaining edges e;
        Delete e from the graph;

        if ( e does not create a cycle in T ) then
            Add e to T;
        endif
    endwhile
end ComputeMST
```

THE GREEDY MST ALGORITHM
-- IMPLEMENTATION ISSUES --

```
Procedure ComputeMST(in: $W[1:n,1:n]$; out: T)    // non-edges $(i,j)$: $W[i,j] = \infty$
begin
    Put in T all the n nodes and no edges;
    while T has less than n-1 edges do
        Select a min-weight edge e out of the remaining edges e;
        Delete e from the graph;

        if ( e does not create a cycle in T ) then
            Add e to T;
        endif
    endwhile
end ComputeMST
```

How?

How?

THE GREEDY MST ALGORITHM

-- IMPLEMENTATION: FINDING-DELETING MIN-EDGE --

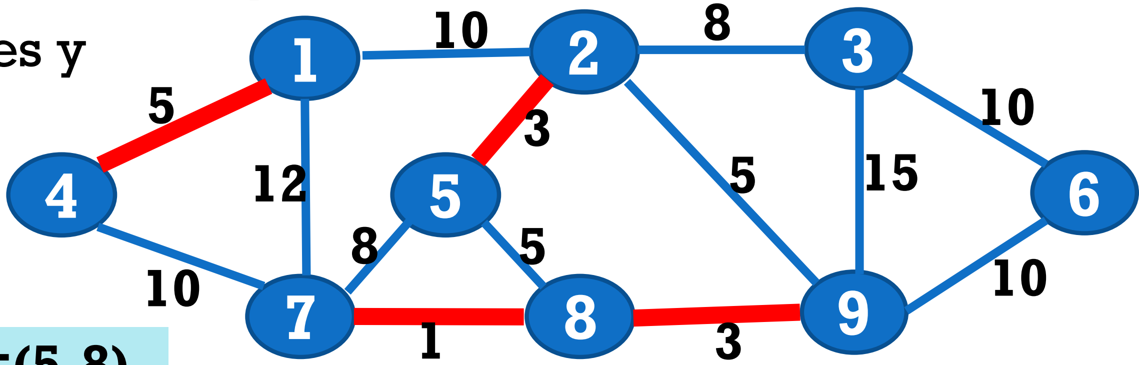
- Since we want to repeatedly find the min-weight edge and delete it from the set of edges, it is good to build a data structure to do that
  - What suitable data structure?
    - Min-heap (of edges, where the key=weight)
    - Build a heap at the start of the MST algorithm
- Alternative solution?
  - Sort the edges (by weight) from at the start of MST algorithm
  - Consider the edges in that sorted order

THE GREEDY MST ALGORITHM

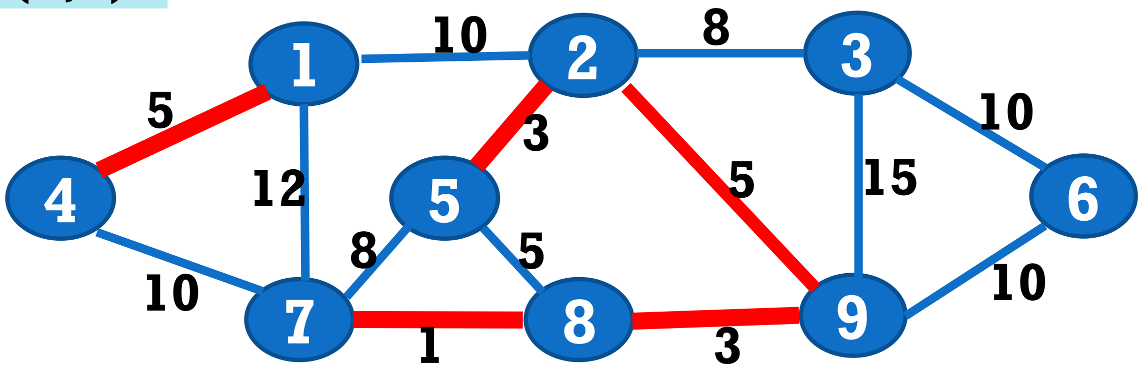
-- IMPLEMENTATION: CHECKING IF EDGE CREATES CYCLE --

- During the algorithm, T is a "forest" of small trees
- When an edge e=(x,y) is being tested if it creates a cycle in T:
  - Node x belongs to one small tree, and so does y
  - If x and y belong to **different** small trees (regardless of the shape of the trees):
    - Adding edge (x,y) will not create a cycle
    - So add it.
  - If x and y belong to the **same small tree** :
    - Adding edge (x,y) creates a cycle
- So, if we can **find** which small tree has x, and which has y, we can check for cycles
- Of course, when we add an edge, the two small trees combine into a new small tree (and the two old small trees no longer exit separately)

Ex: e=(2,9)



Ex: e=(5,8)



THE GREEDY MST ALGORITHM

-- CHECKING IF EDGE CREATES CYCLE: HOW?--

- Do we know of a data structure
  - that can find which tree (or set of elements) contains a given element/node x (or y), and
  - that can combine two old sets to a new set after which the two old sets no longer exit separately?
- The Union-Find data structure does exactly those two operations!!

THE GREEDY MST ALGORITHM

-- IMPLEMENTATION --

```
Procedure ComputeMST(in: $W[1:n,1:n]$; out:T)    // non-edges $(i,j)$: $W[i,j]=\infty$
begin
    integer PARENT[1:n]=[-1,-1,...,-1;                    // for Union-Find
    Build a minheap H[1:|E|] for all the |E| edge
    Put in T the n nodes and no edges;
    while (T has less than n-1 edges) do
        e=delete-min(H);      // assume e=(x,y)
        r1 := F(x);  r2 := F(y);
        if (r1 != r2) then
            Add e to T;
            U(r1,r2);
        endif
  endwhile
end ComputeMST
```

THE GREEDY MST ALGORITHM

-- TIME COMPLEXITY ANALYSIS--

```
Procedure ComputeMST(in: $W[1:n,1:n]$; out: T)   // non-edges $(i,j)$: $W[i,j] = \infty$
begin
    integer PARENT[1:n]=[-1,-1,...,-1];          // for Union-Find
    Build a minheap H[1:|E|] for all the |E| edge
    Put in T the n nodes and no edges;
    while (T has less than n-1 edges) do
        e=delete-min(H);     // assume e=(x,y)
        r1 := F(x);  r2 := F(y);
        if (r1 != r2) then
            Add e to T;
            U(r1,r2);
        endif
  endwhile
end ComputeMST
```

Iterates |E| times, not n-1 times. Why?

- O(|E|) to build the heap
- Up to |E| calls to delete-min: $O(|E|\log|E|)$ time
- Up to |E| calls to U and F: $O(|E|\log n)$ time
- Therefore, the total time: $O(|E|\ log|E|)$

PROOF OF OPTIMALITY OF THE GREEDY MST

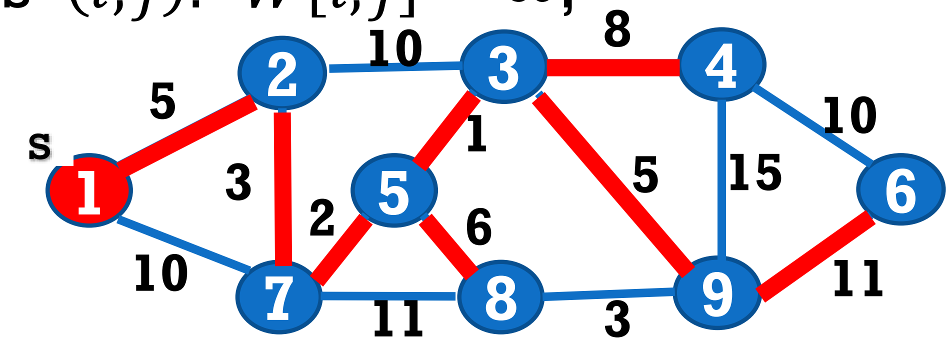
- Next lecture:
  - We will prove that the spanning tree T produced by the greedy algorithm is indeed a minimum spanning tree

LESSONS LEARNED SO FAR

- The same greedy policy on the same problem can be implemented in different ways
- Some implementations can be much faster (e.g., min-heap 4 greedy sorting)
- Pre-processing the input can be very helpful (e.g., sorting P/W)
- The greedy method does not always guarantee optimality
- To prove non-optimality, use counter-examples
- For the same problem, one can formulate different greedy policies, some non-optimal and some optimal
- Some greedy selections may have to be discarded sometimes (like in MST)
- More lessons to come (about the greedy method)

THE SINGLE-SOURCE SHORTEST PATHS PROBLEM
-- PROBLEM STATEMENT --

- **Input**:
  - A weighted connected graph G=(V,E), represented by its weight matrix **W[1:n,1:n]**, where for non-edges (i,j): $W[i,j]=\infty$, and $\forall i, W[i,i]=0$
  - A source node **s** of G.



- **Output**: Shortest paths from source node s to every other node in the graph, one path per node
- **Simpler output:** distance[1:n] where distance[i] is the distance from source node s to node i, i.e., the length of the shortest path from s to i.
- **Task:** Develop a greedy algorithm for this problem

SINGLE-SOURCE SHORTEST PATHS (SSSP)
-- GREEDY METHOD PRELIMINARIES--

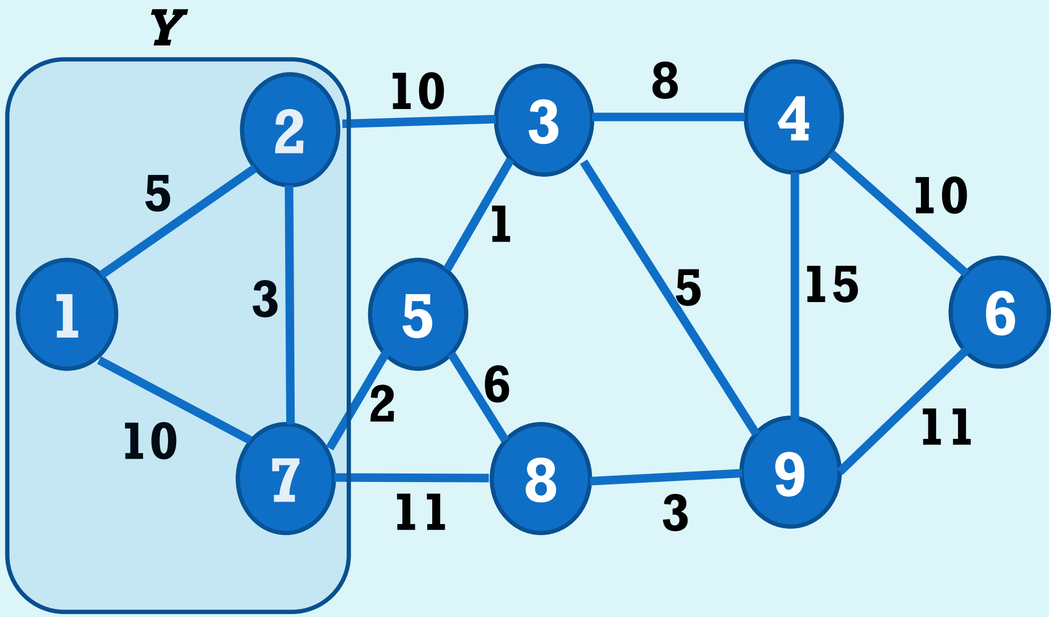
- **Issue**: It is not clear how the solution can be viewed as a set/sequence of elements? What are the elements?
- Recall that sometimes we need to pre-process the input: to make the solution more efficient, and/or to make the greedy solution *formulatable*
- New concepts and definitions will be introduced so a greedy method can be formulated

GREEDY SSSP IDEA

-- DISTANCE APPROXIMATIONS: SPECIAL PATHS --

- Let *Y* be a set := {s} initially
- **Definition**: A path from s to a node x outside Y is called ***special path*** if each intermediary node on the path belongs to Y.
- Let DIST[1:n] be:
  - DIST[i] = the length of the shortest special path from s to i
- **Greedy selection policy**: choose from outside Y the node of minimum DIST value, and add it to Y
- **Claim** (proved later) :

  $\forall i \in Y, DIST[i] = distance[i]$, that is, when a node i joins Y, its DIST is equal to its distance from s.

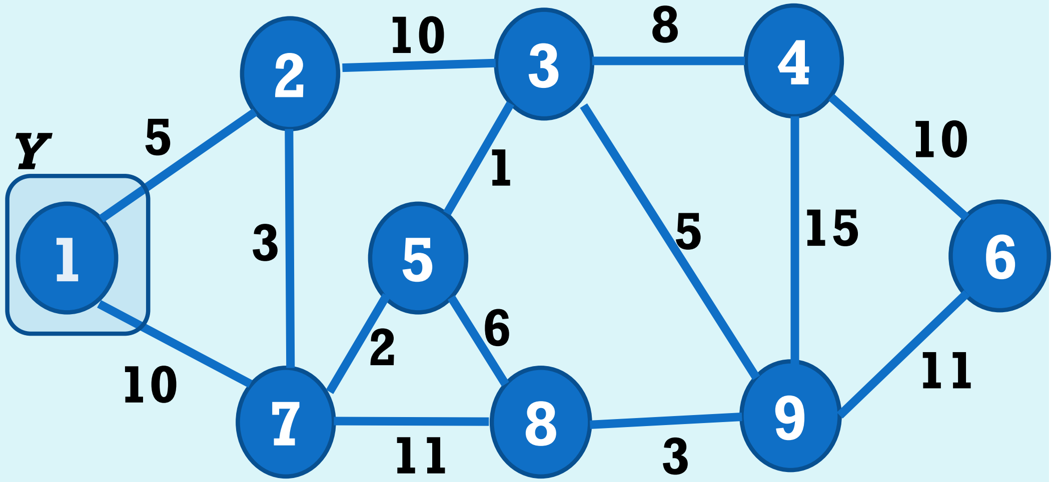


- Special paths:
  - 1, 2, 3 because 1 is source, and 1 and 2 are inside Y
  - 1, 2, 7, 5 b/c 1 is source and 1, 2, and 7 are inside Y
  - 1,5 (missing edge is an edge of weight ∞)
- Not Special paths: 1, 2, 3, 4 (b/c 3 is not in Y); 1,7,5,8 (Why?)

SPECIAL PATHS

-- INITIALIZATION --

- Initially:
  - Y={s}
  - $\forall i \in V$, the only special path from s to i is the edge $(s.i)$, either a real edge of a finite weight, or imaginary of weight ∞
  - Therefore, DIST[i]=W[s,i]
  - In this example:

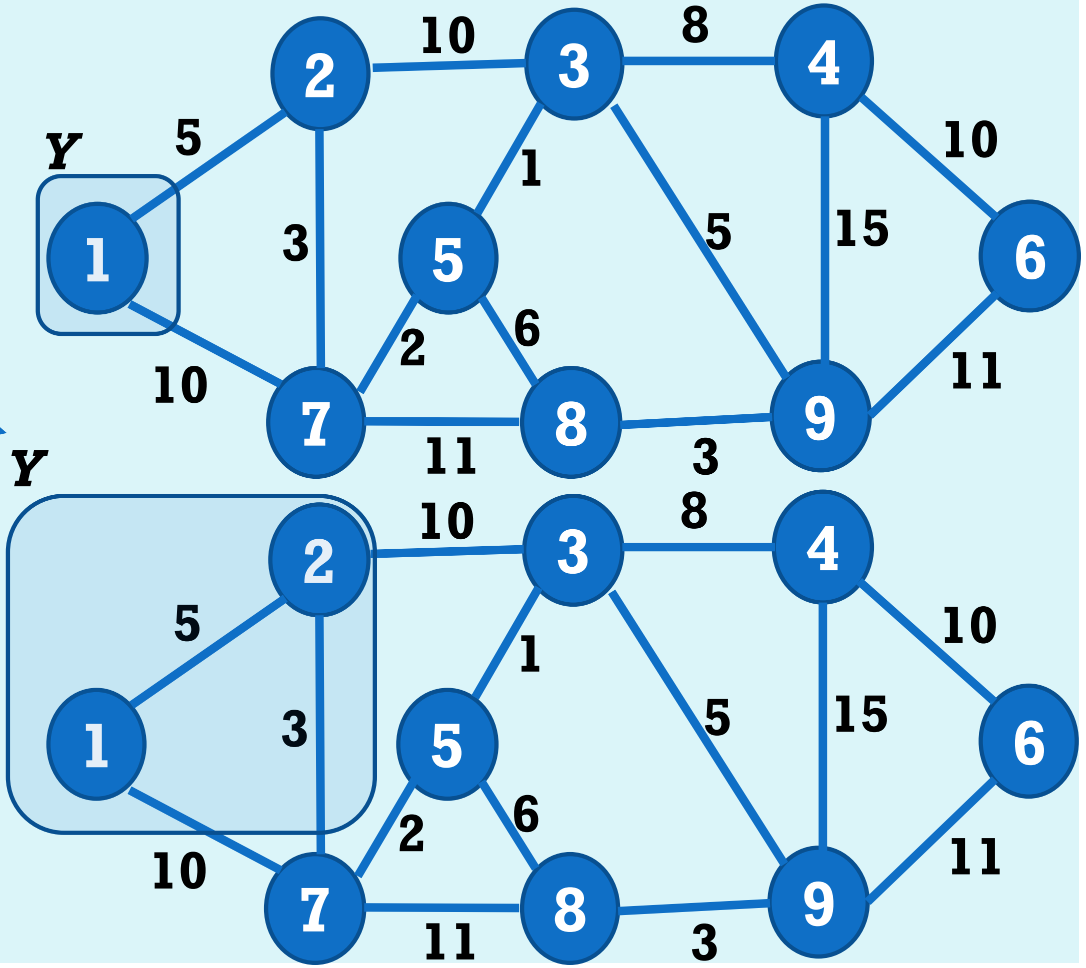


i:	1	2	3	4	5	6	7	8	9
DIST[i]	0	5	∞	∞	∞	∞	10	∞	∞

SPECIAL PATHS: UPDATES

i:	1	2	3	4	5	6	7	8	9
DIST[i]	0	5	∞	∞	∞	∞	10	∞	∞

- **Greedy selection:**
  - Choose from outside Y the node of minimum DIST value, and add it to Y. Call it u
- In this example, u=2 (DIST[2]=5)
- Add 2 to Y: Y={1,2}
- How does DIST change?
  - For any node v outside Y, v gained new special paths, the shortest of which is: MinSpecialPath[s → u]+(u,v) of length: $DIST[u] + W[u,v])$
  - This new special path may be shorter or longer than the precious MinSpecialPath[s → v]
  - $\therefore DIST[v] = \min(DIST[v],\ DIST[u] + W[u,v])$



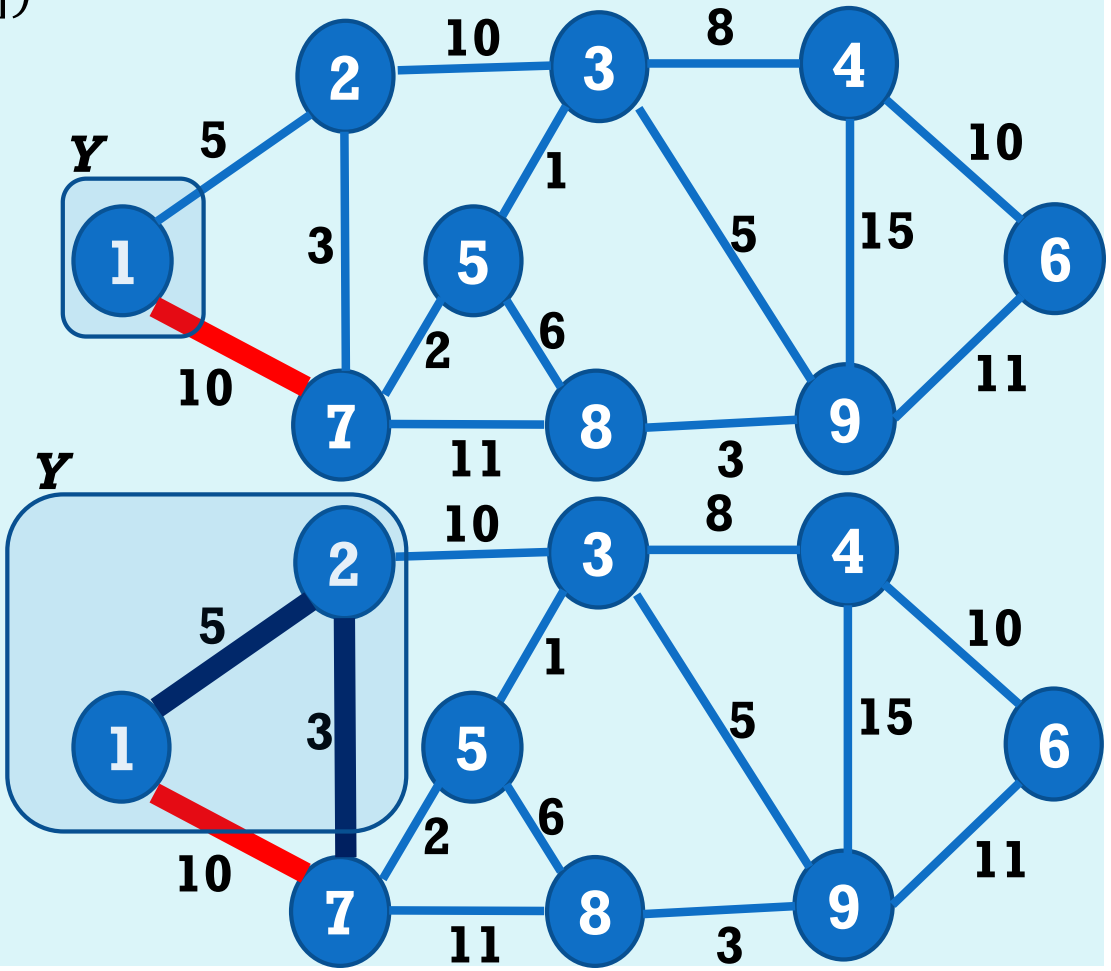
SPECIAL PATHS
-- UPDATES EXAMPLE --

- $DIST[v] = \min(DIST[v],\ DIST[u] + W[u,v])$

i:	1	2	3	4	5	6	7	8	9
DIST[i]	0	5	∞	∞	∞	∞	10	∞	∞

- **Before update: DIST[7]=10**
- After update: DIST[7]=
  min(10, DIST[2]+W[2,7])=
  min(10, 5+3)=8.
- DIST[3]=min(∞, DIST[2]+W[2,3])=15

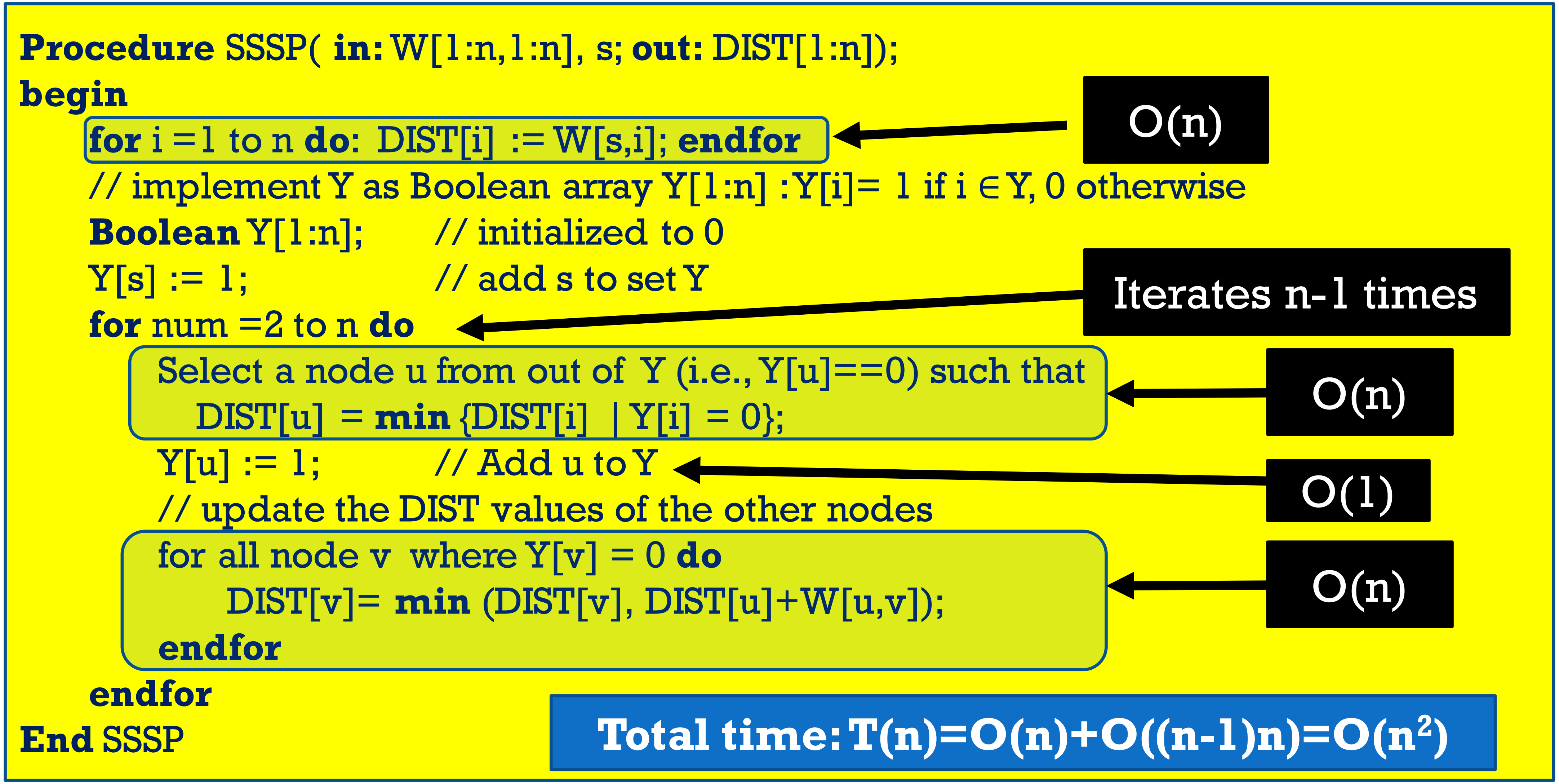
i:	1	2	3	4	5	6	7	8	9
DIST[i]	0	5	15	∞	∞	∞	8	∞	∞



GREEDY SSSP ALGORITHM

```
Procedure SSSP( in: W[1:n,1:n], s; out: DIST[1:n]);
begin
    for i =1 to n do:  DIST[i]  := W[s,i]; endfor
    // implement Y as Boolean array Y[1:n] :Y[i]= 1 if i ∈Y, 0 otherwise
    Boolean Y[1:n];          // initialized to 0
    Y[s] := 1;               // add s to set Y
    for num =2 to n do
        Select a node u from out of  Y (i.e., Y[u]==0) such that
          DIST[u]  = min {DIST[i]  | Y[i]  = 0};
        Y[u] := 1;           // Add u to Y
        // update the DIST values of the other nodes
        for all node v  where Y[v] = 0 do
            DIST[v]= min (DIST[v], DIST[u]+W[u,v]);
        endfor
    endfor
End SSSP
```

GREEDY SSSP ALGORITHM COMPLEXITY



```
Procedure SSSP( in: W[1:n,1:n], s; out: DIST[1:n]);
begin
    for i =1 to n do:  DIST[i]  := W[s,i]; endfor
    // implement Y as Boolean array Y[1:n] : Y[i]= 1 if i ∈ Y, 0 otherwise
    Boolean Y[1:n];        // initialized to 0
    Y[s] := 1;             // add s to set Y
    for num =2 to n do
        Select a node u from out of  Y (i.e., Y[u]==0) such that
            DIST[u]  = min {DIST[i]  | Y[i] = 0};
        Y[u] := 1;         // Add u to Y
        // update the DIST values of the other nodes
        for all node v  where Y[v] = 0 do
            DIST[v]= min (DIST[v], DIST[u]+W[u,v]);
        endfor
    endfor
End SSSP
```

O(n)

Iterates n-1 times

O(n)

O(1)

O(n)

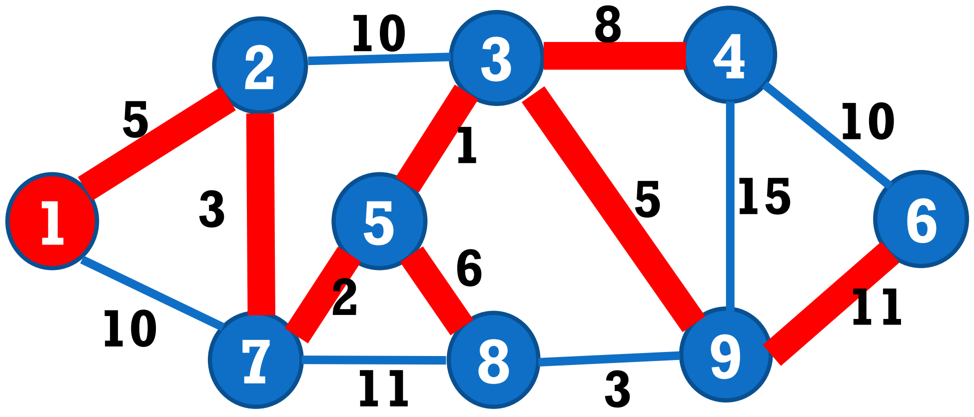
Total time: $T(n)=O(n)+O((n-1)n)=O(n^2)$

GREEDY SSSP

-- COMPLETE EXAMPLE --

i	1	2	3	4	5	6	7	8	9	Y={1}
$DIST[i]$	0	**5**	∞	∞	∞	∞	10	∞	∞	u=2 -> Y={1,2}
$DIST[i]$	0	5	15	∞	∞	∞	**8**	∞	∞	u=7 -> Y={1,2,7}
$DIST[i]$	0	5	15	∞	**10**	∞	8	19	∞	u=5 -> Y={1,2,7,5}
$DIST[i]$	0	5	**11**	∞	10	∞	8	16	∞	u=3 -> Y={1,2,7,5,3}
$DIST[i]$	0	5	11	19	10	∞	8	**16**	16	u=8 -> Y={1,2,7,5,3,8}
$DIST[i]$	0	5	11	19	10	∞	8	16	**16**	u=9 ->Y={1,2,7,5,3,8,9}
$DIST[i]$	0	5	11	**19**	10	27	8	16	16	u=4 -> Y=1,2,7,5,3,8,9,4}
$DIST[i]$	0	5	11	19	10	**27**	8	16	16	U=6 -> Y=1,2,7,5,3,8,9,4,6}

Shortest paths from 1 to the other nodes, highlighted in red edges



LESSONS LEARNED SO FAR

- The same greedy policy on the same problem can be implemented in different ways
- Some implementations can be much faster (e.g., min-heap for greedy sorting)
- Pre-processing the input can be very helpful (e.g., sorting P/W)
- The greedy method does not always guarantee optimality
- To prove non-optimality, use counter-examples
- For the same problem, one can formulate different greedy policies, some non-optimal and some optimal
- Sometimes greedy selections may have to be discarded sometimes (like in MST)
- Sometimes, problems may have to be reformulated to make the greedy formulatable (as in SSSP)
- More lessons to come (about the greedy method)

OPTIMALITY OF THE GREEDY SSSP

- Next lecture
  - We will show that the final DIST values of all the nodes are indeed the distances (i.e., lengths of shortest paths) from s to the other nodes

ADDITIONAL WORK (1)

- **An exercise for the students**:

  How will you modify the greedy SSSP algorithm so it returns the actual shortest paths, not just the distances

- **Helpful observations**:
  - The greedy-selected shortest paths from s to all the nodes form a tree rooted at s
  - Have the edges of that tree point backward (towards the root s)
  - Your modified greedy SSSP can include that tree, and updates to Y and DIST can translate to updates to that tree
  - Once the tree is fully derived, the shortest paths can be generated by tracing back from each node to the root s (and then reversing those paths)

ADDITIONAL WORK (2)

-- THE COIN CHANGE PROBLEM --

- **Input:**
  - A currency system made up of an unlimited number of coins of the following denominations, i.e., values, $\{C_1, C_2, \ldots, C_m\}$. For example, denominations {1, 5, 10, 25} representing a penny, nickel, dime, and quarter.
  - An amount N (like N cents)
- **Output**: A minimum number of coins whose total value is N
- **Task**: Formulate a greedy algorithm for this problem
- **Questions**:
  - Does your greedy algorithm guarantee optimality (i.e., guarantee that the number of coins making up the change N is minimum)? For an any arbitrary currency system? For the American coinage system ({1, 5, 10, 25})?
  - If for some currency systems the greedy method doesn't guarantee optimality, give a counter-example of a currency system and an N for which the greedy solution is not best