CSE 332 Winter 2024 Lecture 15: Sorting

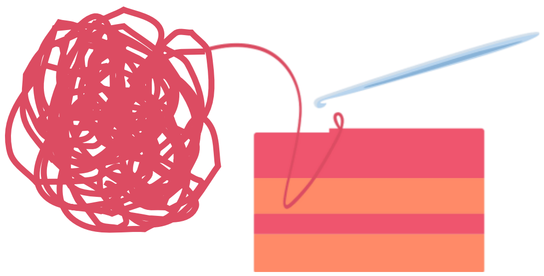
Nathan Brunelle

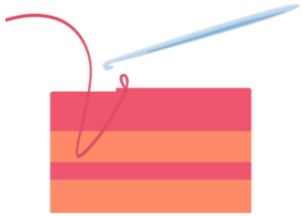
http://www.cs.uw.edu/332

Divide And Conquer Sorting

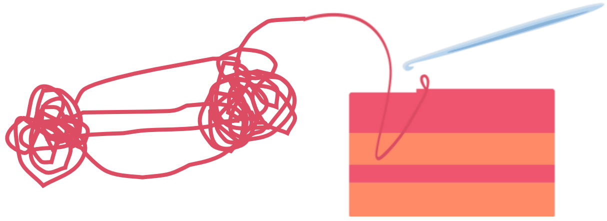
- Divide and Conquer:
  - Recursive algorithm design technique
  - Solve a large problem by breaking it up into smaller versions of the same problem

Divide and Conquer

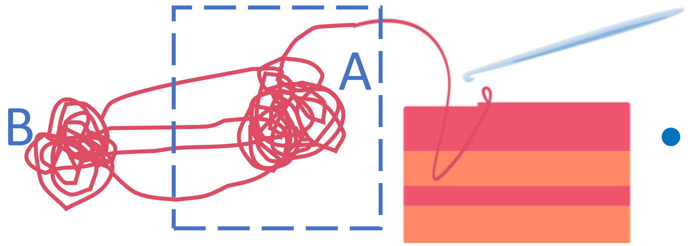




- **Base Case**:
  - If the problem is “small” then solve directly and return



- **Divide**:
  - Break the problem into subproblem(s), each smaller instances

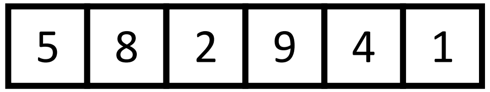


- **Conquer**:
  - Solve subproblem(s) recursively
- **Combine**:
  - Use solutions to subproblems to solve original problem

Divide and Conquer Template Pseudocode

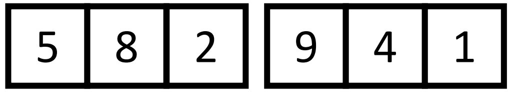
```
def my_DandC(problem){
   // Base Case
   if (problem.size() <= small_value){
      return solve(problem);  // directly solve (e.g., brute force)
   }
   // Divide
   List subproblems = divide(problem);

   // Conquer
   solutions = new List();
   for (sub : subproblems){
      subsolution = my_DandC(sub);
      solutions.add(subsolution);
   }
   // Combine
   return combine(solutions);
}
```

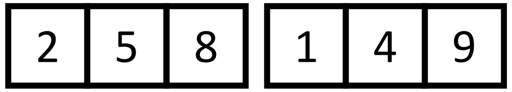


Merge Sort

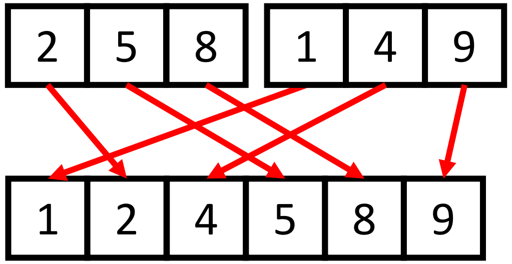
- **Base Case**:
  - If the list is of length 1 or 0, it's already sorted, so just return it



- **Divide**:
  - Split the list into two "sublists" of (roughly) equal length



- **Conquer**:
  - Sort both lists recursively



- **Combine**:
  - **Merge** sorted sublists into one sorted list

Merge Sort In Action!

Sort between indices low and $high$

5	8	2	9	4	1	3	7
low							$high$

Base Case: if low == $high$ then that range is already sorted!

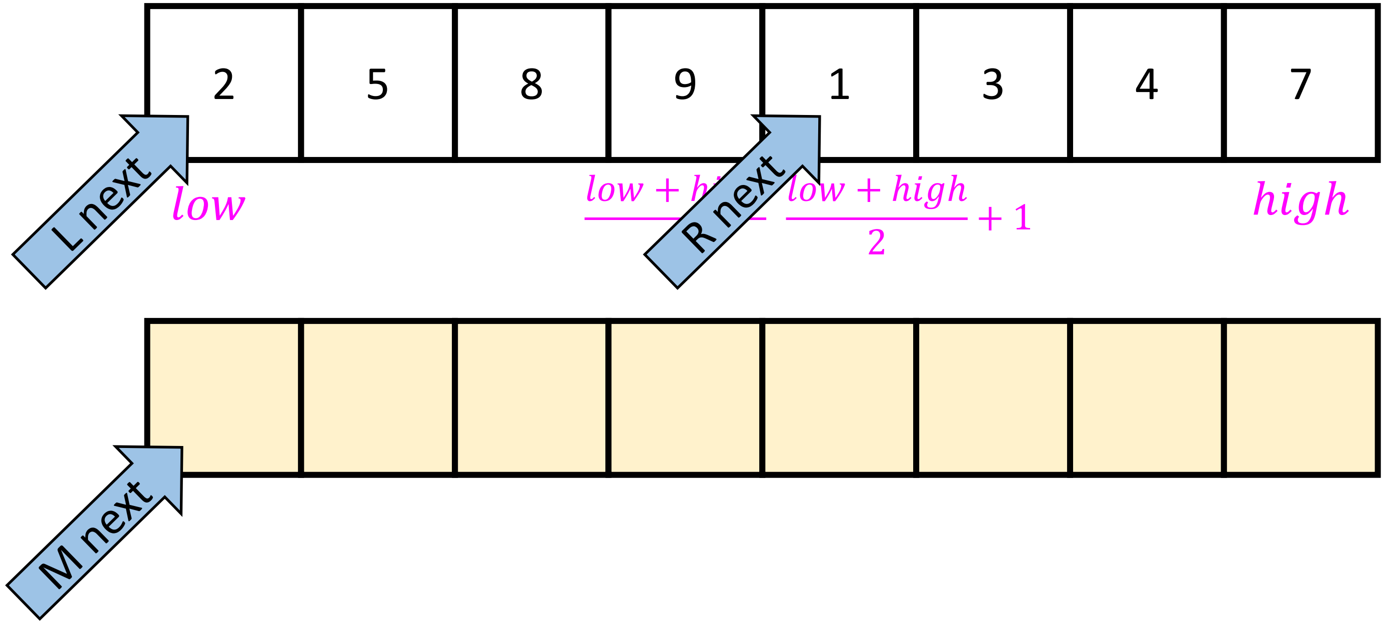
Divide and Conquer: Otherwise call mergesort on ranges $\left(low, \frac{low+high}{2}\right)$ and $\left(\frac{low+high}{2}+1, high\right)$

5	8	2	9	4	1	3	7
low			$\frac{low+high}{2}$	$\frac{low+high}{2}+1$			$high$

After Recursion:

2	5	8	9	1	3	4	7
low							$high$

Merge (the combine part)



Create a new array to merge into, and 3 pointers/indices:

- L_next: the smallest “unmerged” thing on the left
- R_next: the smallest “unmerged” thing on the right
- M_next: where the next smallest thing goes in the merged array

One-by-one: put the smallest of L_next and R_next into M_next, then advance both M_next and whichever of L/R was used.

Merge Sort Pseudocode

```
void mergesort(myArray){
        ms_helper(myArray, 0, myArray.length());
}
void mshelper(myArray, low, high){
        if (low == high){return;}  // Base Case
        mid = (low+high)/2;
        ms_helper(low, mid);
        ms_helper(mid+1, high);
        merge(myArray, low, mid, high);
}
```

Merge Pseudocode

```
void merge(myArray, low, mid, high){
        merged = new int[high-low+1]; // or whatever type is in myArray
        l_next = low;
        r_next = high;
        m_next = 0;
        while (l_next <= mid && r_next <= high){
                if (myArray[l_next] <= myArray[r_next]){
                        merged[m_next++] = myArray[l_next++];
                }
                else{
                        merged[m_next++] = myArray[r_next++];
                }
        }
        while (l_next <= mid){ merged[m_next++] = myArray[l_next++]; }
        while (r_next <= high){  merged[m_next++] = myArray[r_next++]; }
        for(i=0; i<=merged.length; i++){ myArray[i+low] = merged[i];}
}
```

Analyzing Merge Sort

1. Identify time required to Divide and Combine
2. Identify all subproblems and their sizes
3. Use recurrence relation to express recursive running time
4. Solve and express running time asymptotically

- **Divide**: 0 comparisons
- **Conquer**: recursively sort two lists of size $\frac{n}{2}$
- **Combine**: n comparisons
- **Recurrence:**

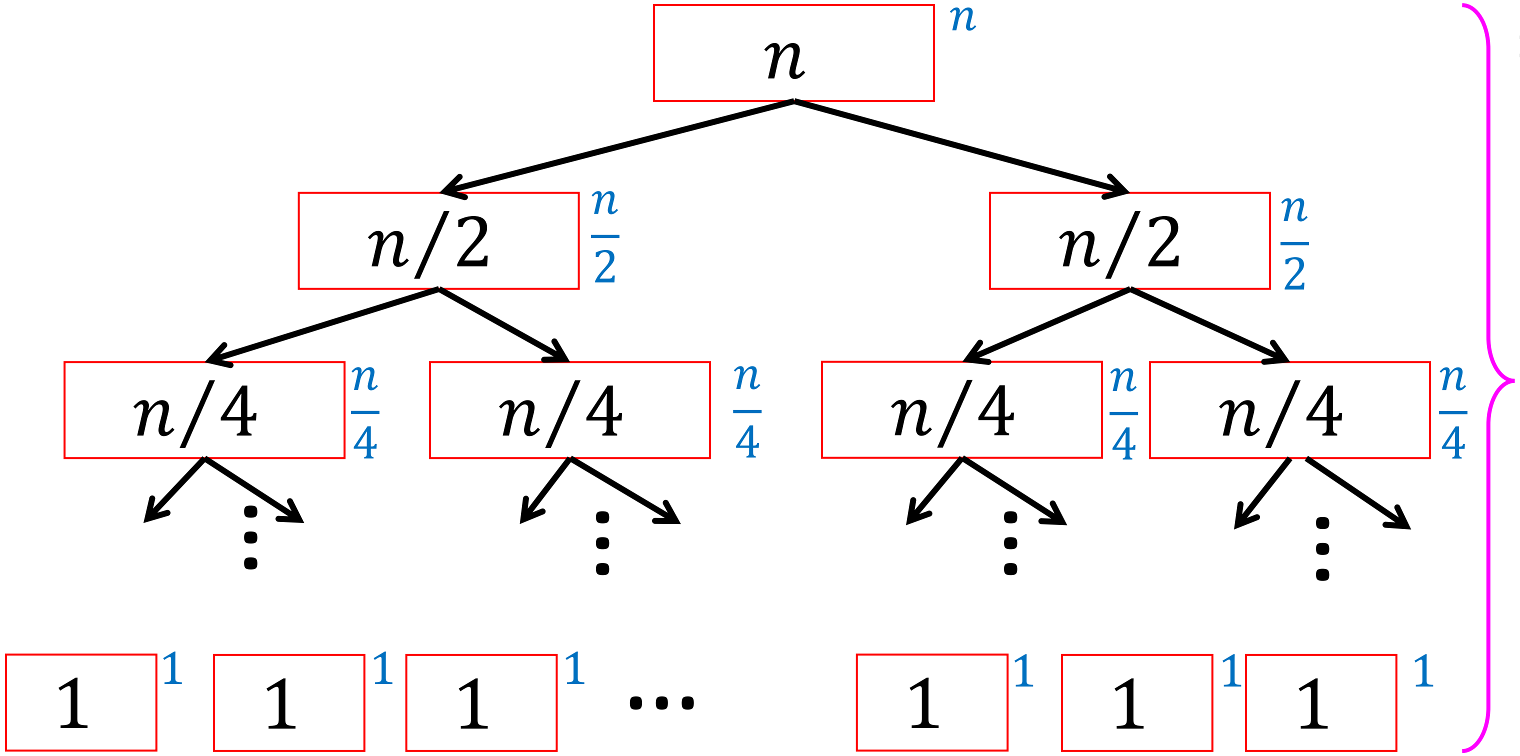
$$T(n) = 0 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

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

Red box represents a problem instance

Blue value represents time spent at that level of recursion

$$T(n) = 2T(\frac{n}{2}) + n$$



$\Rightarrow n$ comparisons / level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_2 n} n = n \log_2 n$$

Properties of Merge Sort

- Worst Case Running time:
  - $\Theta(n \log n)$
- In-Place?
  - No!
- Adaptive?
  - No!
- Stable?
  - Yes!
  - As long as in a tie you always pick l_next

Quicksort

- Like Mergesort:
  - Divide and conquer
  - $O(n \log n)$ run time (kind of…)
- Unlike Mergesort:
  - Divide step is the “hard” part
  - *Typically* faster than Mergesort

Quicksort

Idea: pick a pivot element, recursively sort two sublists around that element

- Divide: select pivot element p, Partition(p)
- Conquer: recursively sort left and right sublists
- Combine: Nothing!

Partition (Divide step)

Given: a list, a pivot p

Start: unordered list

8	5	7	3	12	10	1	2	4	9	6	11

Goal: All elements $< p$ on left, all $> p$ on right

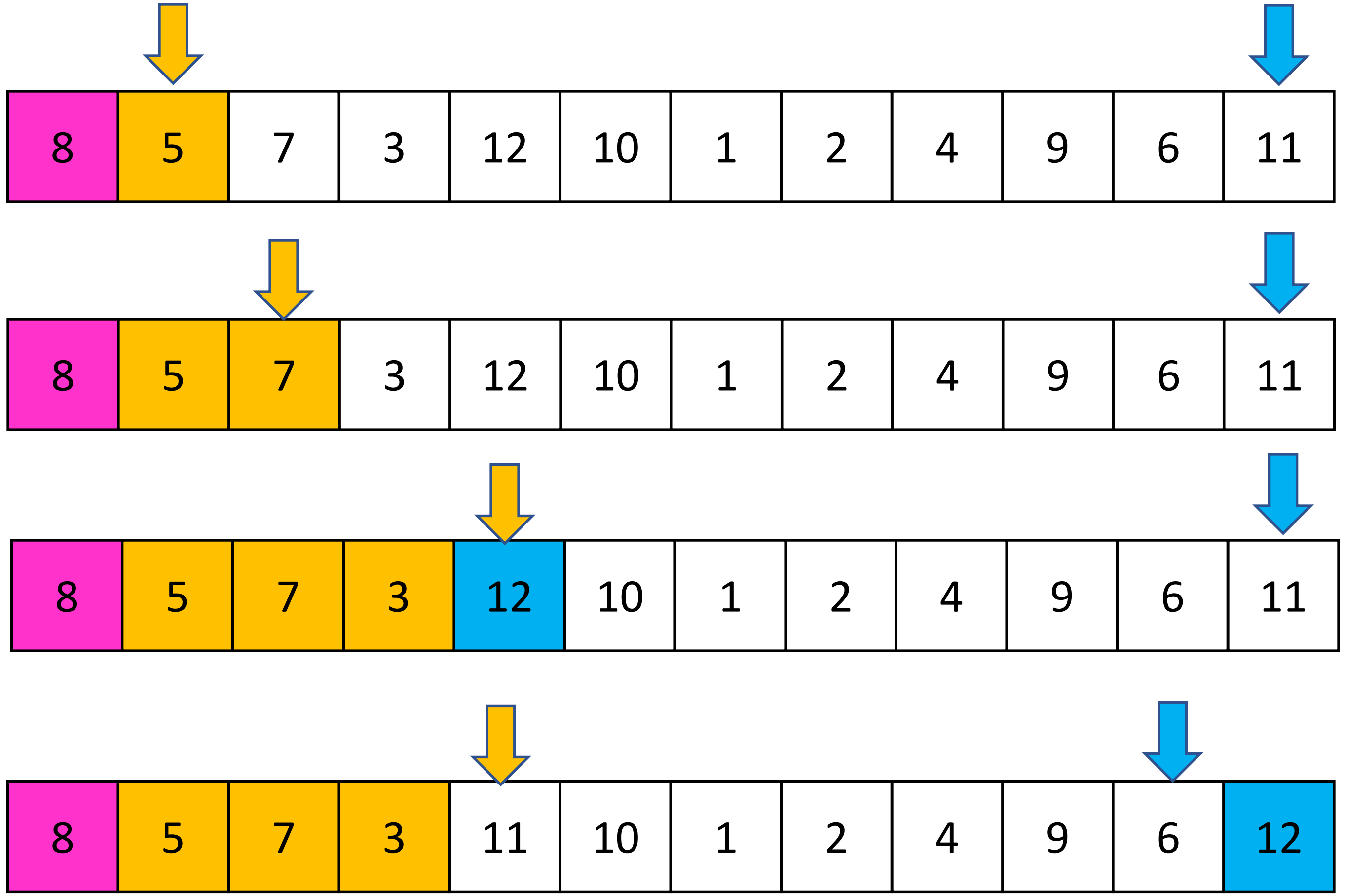
5	7	3	1	2	4	6	8	12	10	9	11

Partition, Procedure

If Begin value $< p$, move Begin right

Else swap Begin value with End value, move End Left

Done when Begin = End

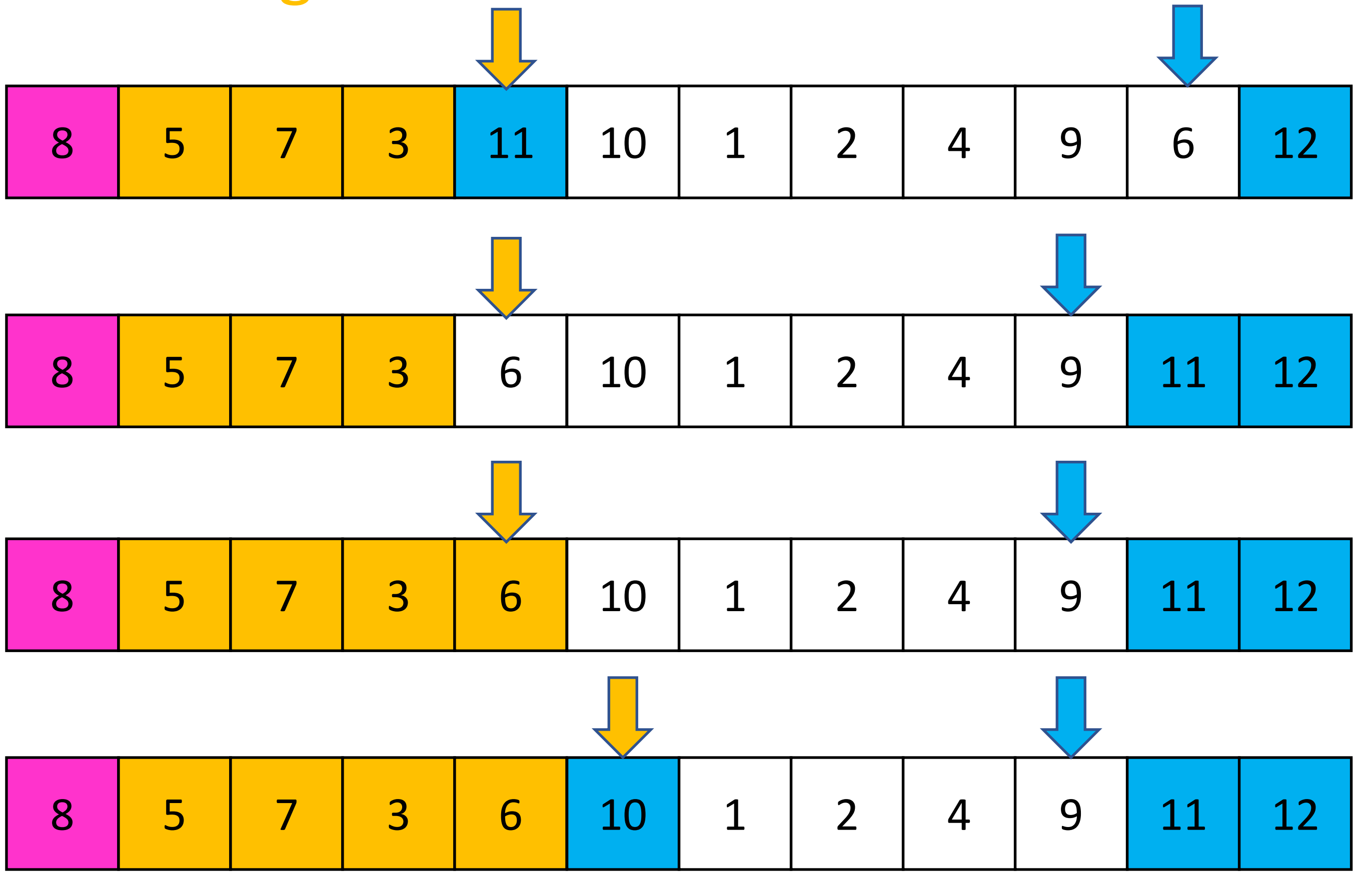


Partition, Procedure

If Begin value $< \ p$, move Begin right

Else swap Begin value with End value, move End Left

Done when Begin = End



Partition, Procedure

If Begin value $<\ p$, move Begin right

Else swap Begin value with End value, move End Left

Done when Begin = End

8	5	7	3	6	4	1	2	10	9	11	12

Case 1: meet at element $< p$

Swap p with pointer position (2 in this case)

2	5	7	3	6	4	1	8	10	9	11	12

Partition, Procedure

If Begin value $<\ p$, move Begin right

Else swap Begin value with End value, move End Left

Done when Begin = End

8	5	7	3	6	4	1	2	10	9	11	12

Case 2: meet at element $> p$

Swap p with value to the left (2 in this case)

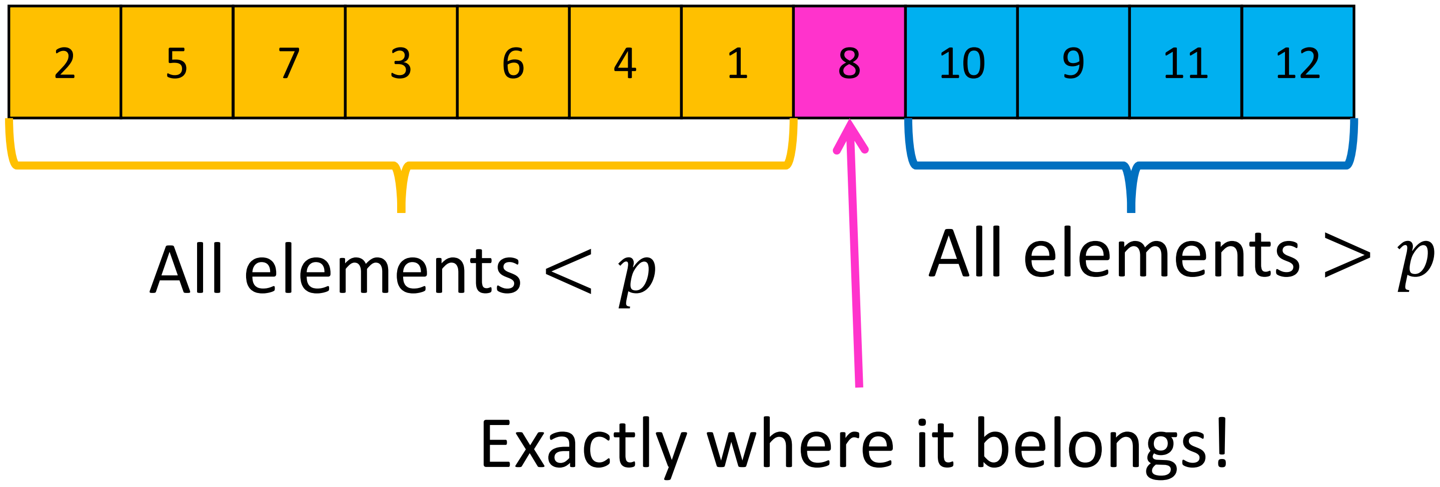
2	5	7	3	6	4	1	8	10	9	11	12

Partition Summary

1. Put p at beginning of list
2. Put a pointer (Begin) just after p, and a pointer (End) at the end of the list
3. While Begin < End:
    1. If Begin value $< p$, move Begin right
    2. Else swap Begin value with End value, move End Left
4. If pointers meet at element $< p$: Swap p with pointer position
5. Else If pointers meet at element $> p$: Swap p with value to the left

Run time? $O(n)$

Conquer



Recursively sort Left and Right sublists

Quicksort Run Time (Best)

If the pivot is always the median:

2	5	1	3	6	4	7	8	10	9	11	12

2	1	3	5	6	4	7	8	9	10	11	12

Then we divide in half each time

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

$$T(n) = O(n \log n)$$

Quicksort Run Time (Worst)

If the pivot is always at the extreme:

1	5	2	3	6	4	7	8	10	9	11	12

1	2	3	5	6	4	7	8	10	9	11	12

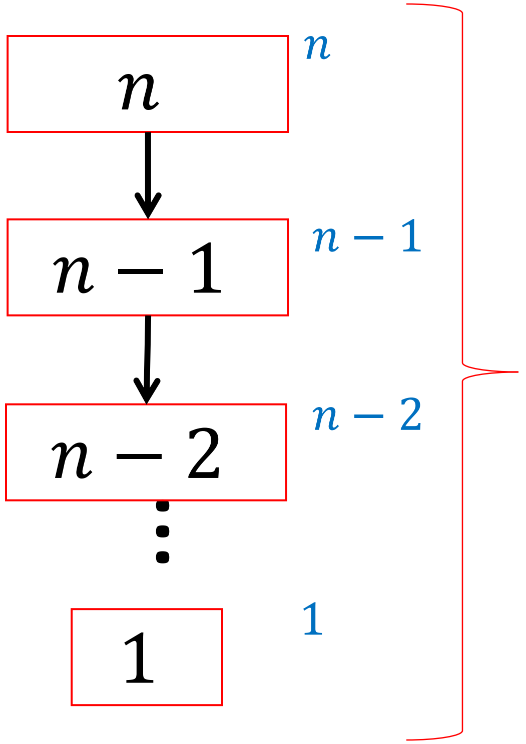
Then we shorten by 1 each time

$$T(n) = T(n-1) + n$$

$$T(n) = O(n^2)$$

Quicksort Run Time (Worst)

$$T(n) = T(n-1) + n$$



$$T(n) = 1 + 2 + 3 + \cdots + n$$

$$T(n) = \frac{n(n+1)}{2}$$

$$T(n) = O(n^2)$$

Quicksort on a (nearly) Sorted List

First element always yields unbalanced pivot

1	2	3	4	5	6	7	8	9	10	11	12

1	2	3	4	5	6	7	8	9	10	11	12

So we shorten by 1 each time

$$T(n) = T(n-1) + n$$

$$T(n) = O(n^2)$$

Good Pivot

- What makes a good Pivot?
  - Roughly even split between left and right
  - Ideally: median
- There are ways to find the median in linear time, but it’s complicated and slow and you’re better off using mergesort
- In Practice:
  - Pick a random value as a pivot
  - Pick the middle of 3 random values as the pivot

Properties of Quick Sort

- Worst Case Running time:
  - $\Theta(n^2)$
  - But $\Theta(n \log n)$ average! And typically faster than mergesort!
- In-Place?
  - ….Debatable
- Adaptive?
  - No!
- Stable?
  - No!

Improving Running time

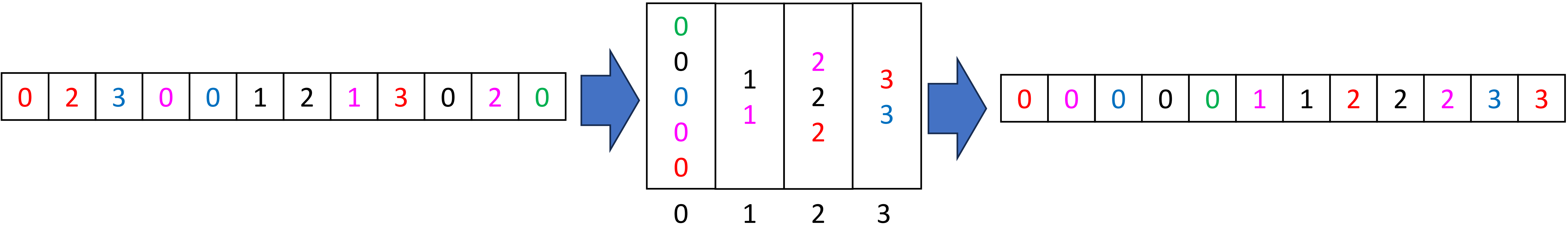
- Recall our definition of the sorting problem:
    - Input:
        - An array A of items
        - A comparison function for these items
            - Given two items x and y, we can determine whether $x < y$, $x > y$, or $x = y$
    - Output:
        - A permutation of A such that if $i \leq j$ then $A[i] \leq A[j]$
- Under this definition, it is impossible to write an algorithm faster than $n \log n$ asymptotically.
- Observation:
    - Sometimes there might be ways to determine the position of values without comparisons!

“Linear Time” Sorting Algorithms

- Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
  - Examples:
    - The list contains only positive integers less than k
    - The number of distinct values in the list is much smaller than the length of the list
- The running time expression will always have a term other than the list’s length to account for this assumption
  - Examples:
    - Running time might be $\Theta(k \cdot n)$ where k is the range/count of values

BucketSort

- Assumes the array contains integers between 0 and $k - 1$ (or some other small range)
- Idea:
  - Use each value as an index into an array of size k
  - Add the item into the “bucket” at that index (e.g. linked list)
  - Get sorted array by “appending” all the buckets



BucketSort Running Time

- Create array of k buckets
  - Either $\Theta(k)$ or $\Theta(1)$ depending on some things…
- Insert all n things into buckets
  - $\Theta(n)$
- Empty buckets into an array
  - $\Theta(n+k)$
- Overall:
  - $\Theta(n+k)$
- When is this better than mergesort?

Properties of BucketSort

- In-Place?
  - No
- Adaptive?
  - No
- Stable?
  - Yes!

RadixSort

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
103	801	401	323	255	823	999	101	113	901	555	512	245	800	018	121

Place each element into a "bucket" according to its 1's place

0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

Place each element into a “bucket” according to its 10’s place

0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

RadixSort

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

Place each element into a "bucket" according to its 100's place

0	1	2	3	4	5	6	7	8	9
018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999

RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999
0	1	2	3	4	5	6	7	8	9

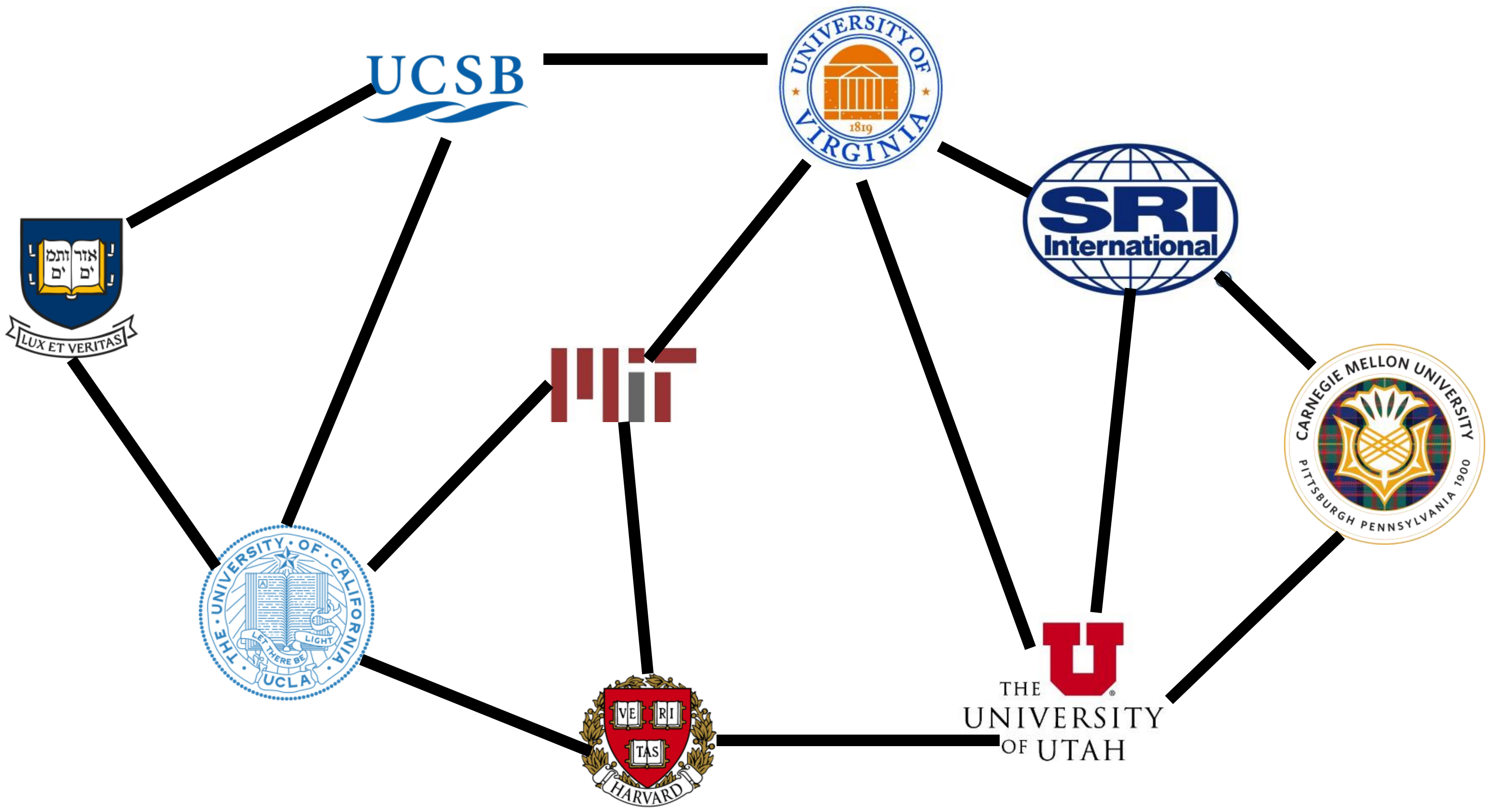
Convert back into an array

018	811	103	113	121	245	255	323	401	512	555	800	801	823	901	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

RadixSort Running Time

- Suppose largest value is m
- Choose a radix (base of representation) b
- BucketSort all n things using b buckets
  - $\Theta(n+k)$
- Repeat once per each digit
  - $\log_b m$ iterations
- Overall:
  - $\Theta(n \log_b m + b \log_b m)$
- In practice, you can select the value of b to optimize running time
- When is this better than mergesort?

ARPANET

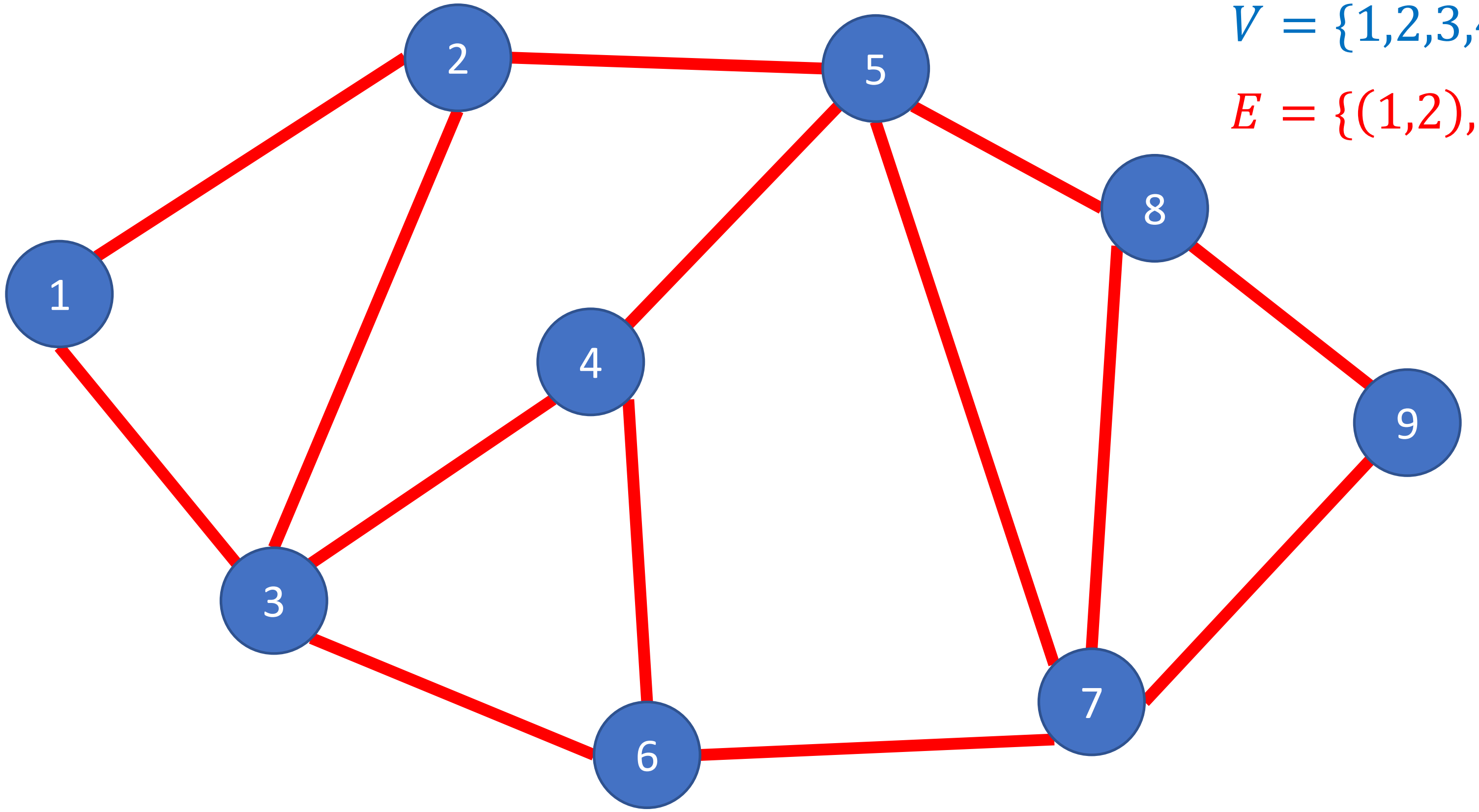

UCSB
UNIVERSITY OF VIRGINIA
1819
SRI
International
LUX ET VERITAS
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PENNSYLVANIA 1900
THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA
VE RI TAS
HARVARD
THE UNIVERSITY OF UTAH

Undirected Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges



$V = \{1,2,3,4,5,6,7,8,9\}$

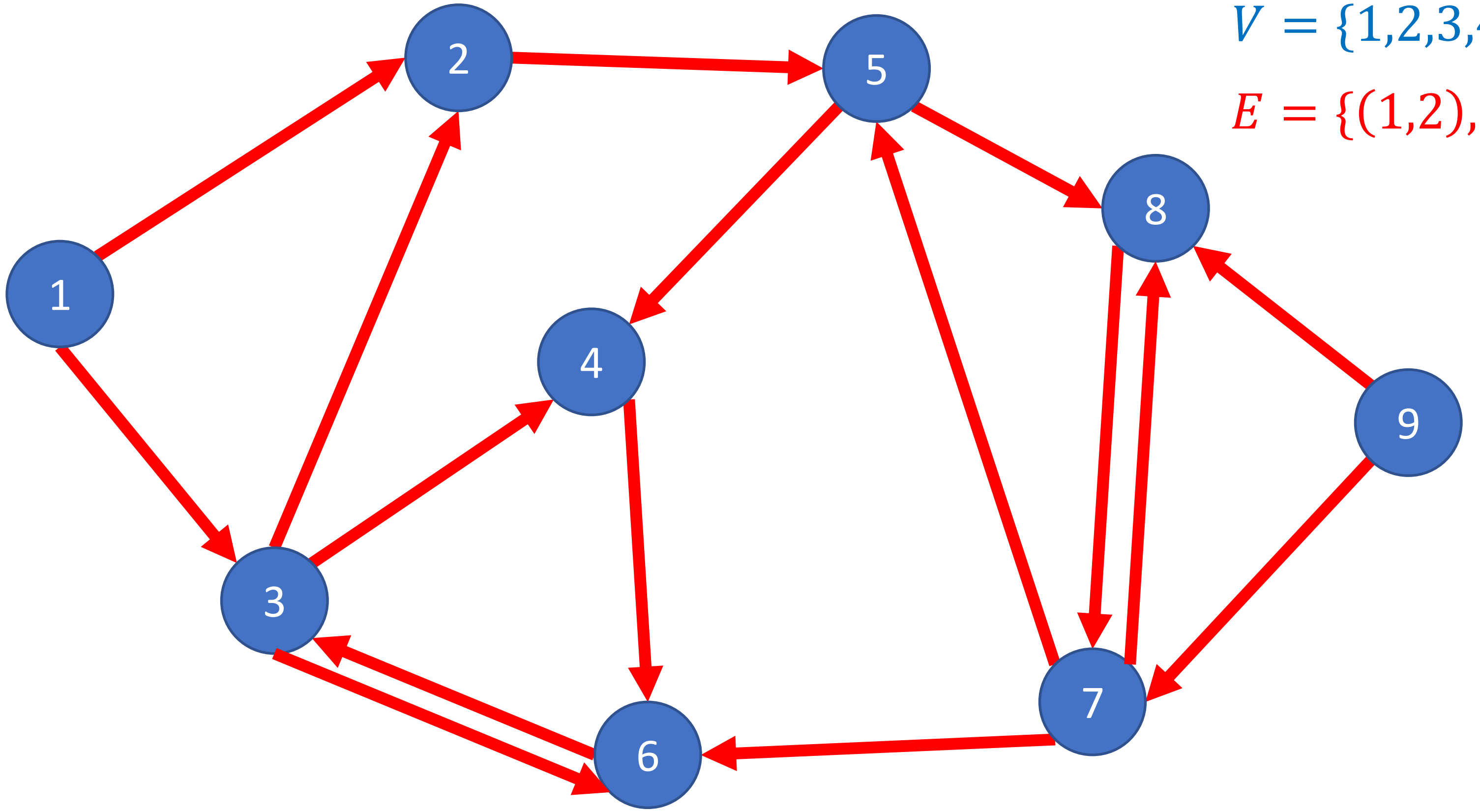
$E = \{(1,2), (2,3), (1,3), \ldots\}$

Directed Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

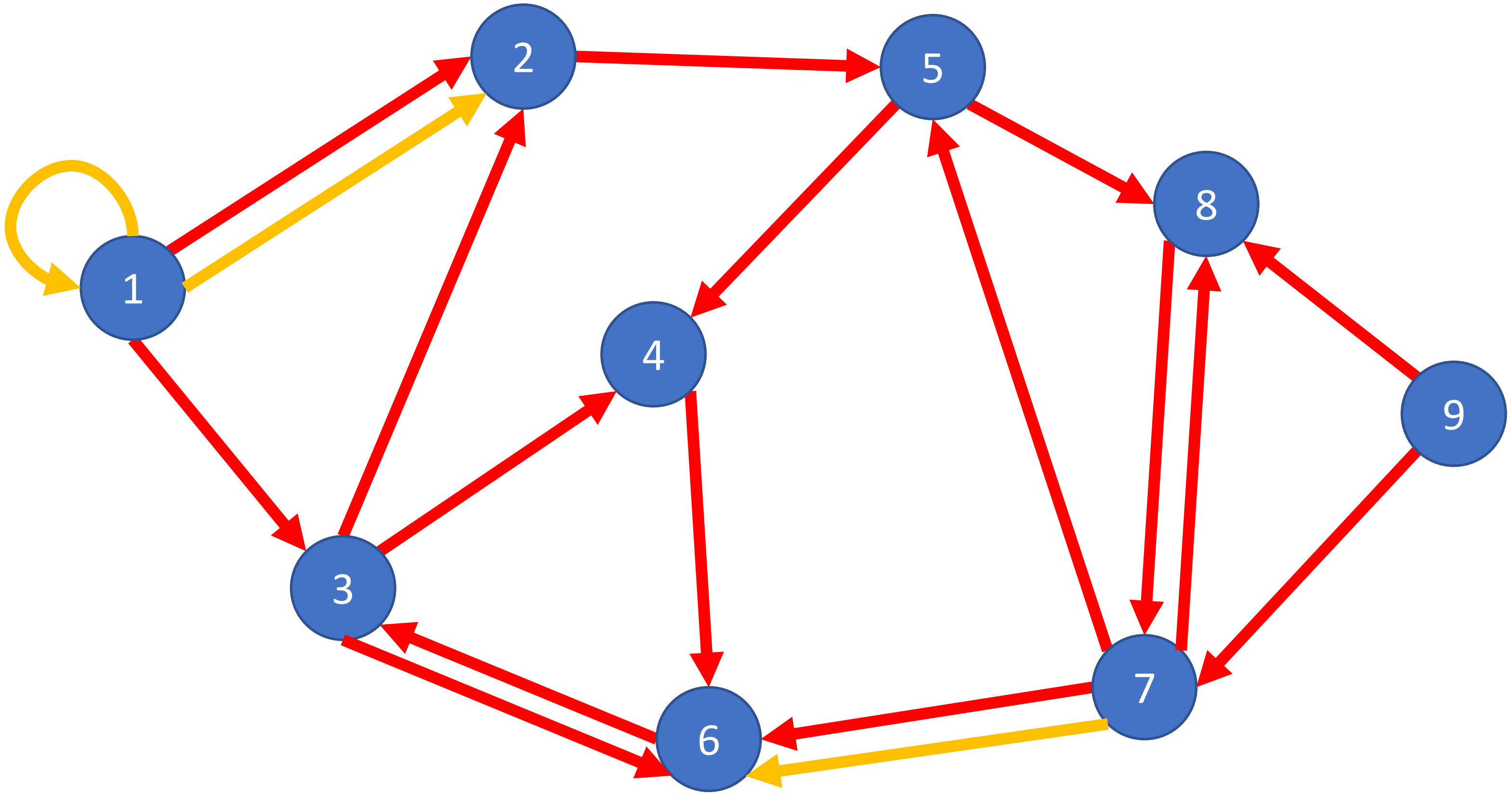


$V = \{1,2,3,4,5,6,7,8,9\}$

$E = \{(1,2), (2,3), (1,3), \dots\}$

Self-Edges and Duplicate Edges

Some graphs may have duplicate edges (e.g. here we have the edge (1,2) twice).
Some may also have self-edges (e.g. here there is an edge from 1 to 1).
Graph with Neither self-edges nor duplicate edges are called **simple graphs**



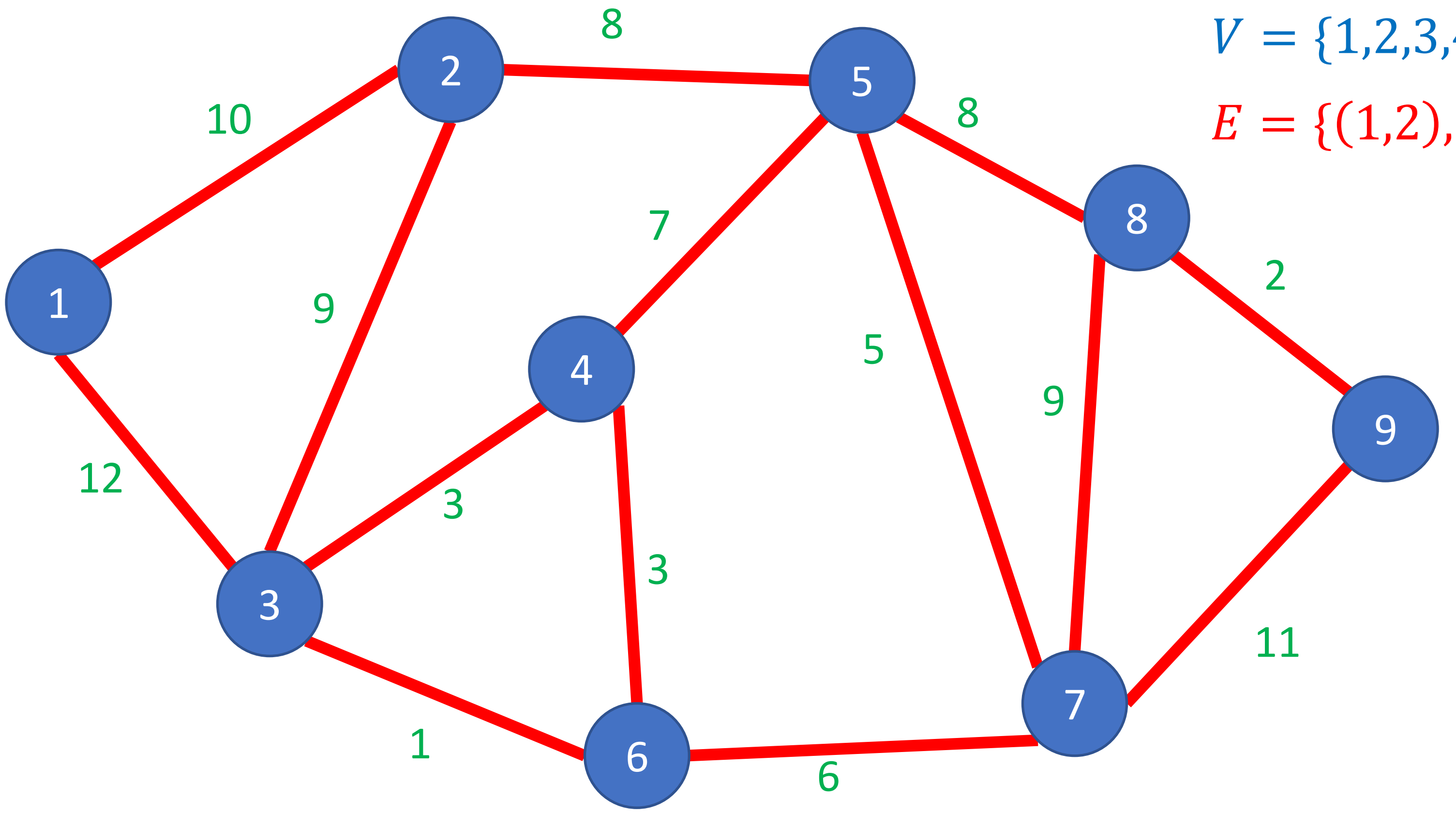
Weighted Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

$w(e) =$ weight of edge e



$V = \{1,2,3,4,5,6,7,8,9\}$

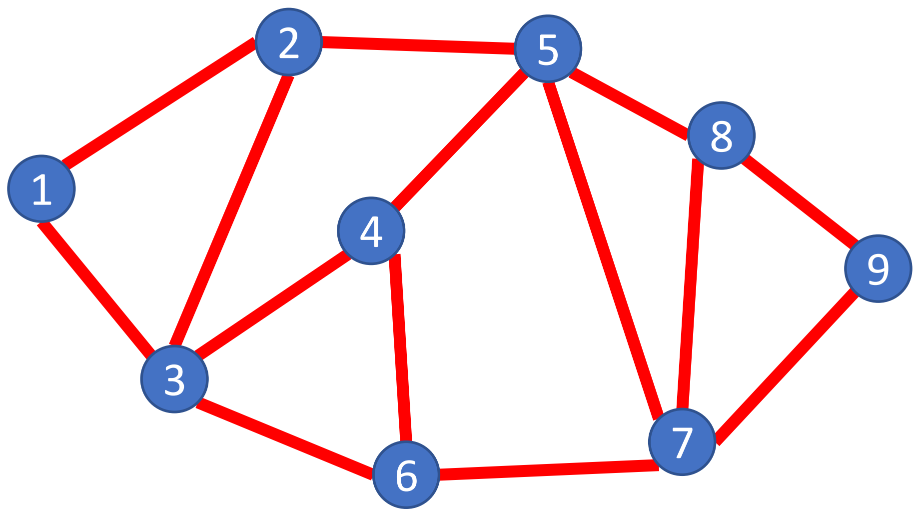
$E = \{(1,2), (2,3), (1,3), \dots\}$

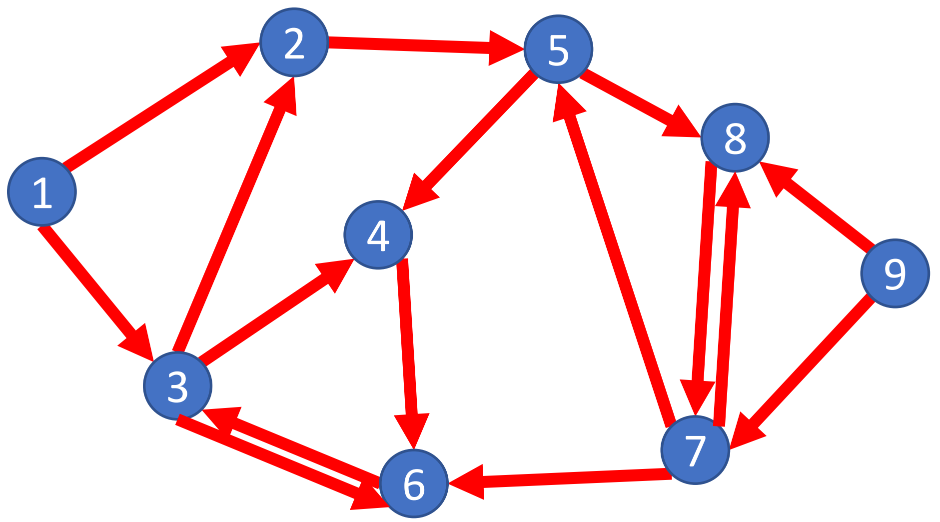
Graph Applications

- For each application below, consider:
  - What are the nodes, what are the edges?
  - Is the graph directed?
  - Is the graph simple?
  - Is the graph weighted?
- Facebook friends
- Twitter followers
- Java inheritance
- Airline Routes

Some Graph Terms

- Adjacent/Neighbors
  - Nodes are adjacent/neighbors if they share an edge
- Degree
  - Number of “neighbors” of a vertex
- Indegree
  - Number of incoming neighbors
- Outdegree
  - Number of outgoing neighbors

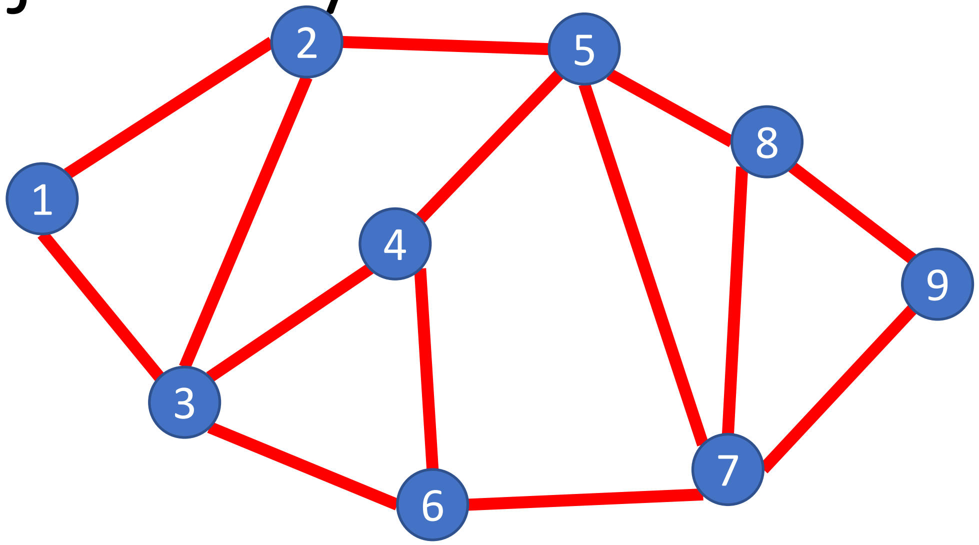


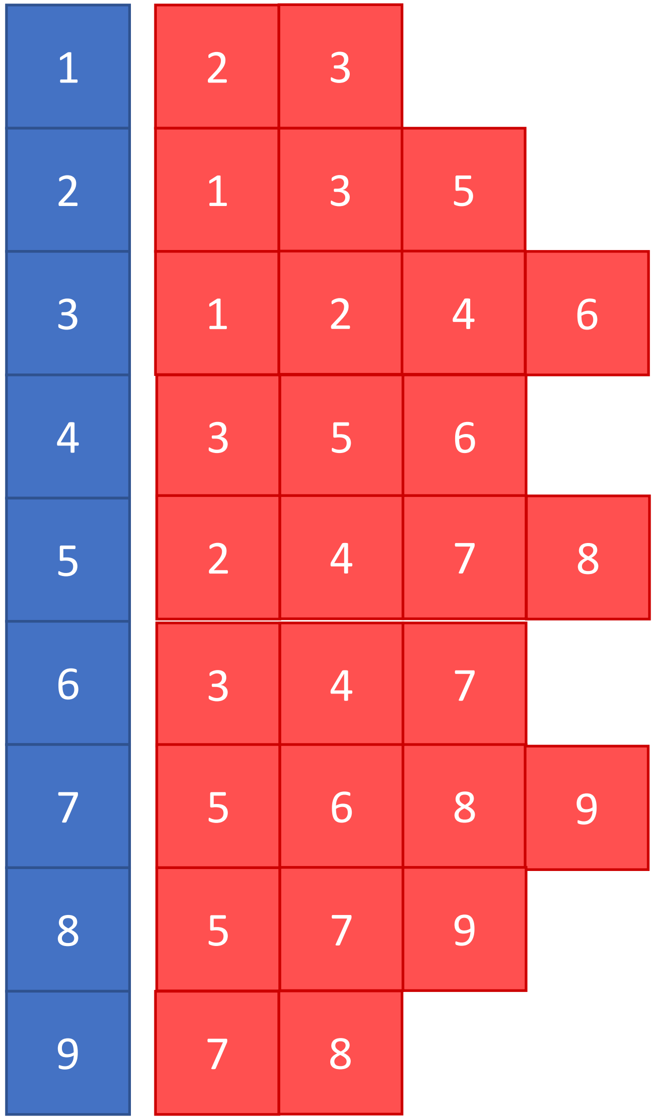


Graph Operations

- To represent a Graph (i.e. build a data structure) we need:
  - Add Edge
  - Remove Edge
  - Check if Edge Exists
  - Get Neighbors (incoming)
  - Get Neighbors (outgoing)

Adjacency List





<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n+m)$

Add Edge: $\Theta(1)$

Remove Edge: $\Theta(1)$

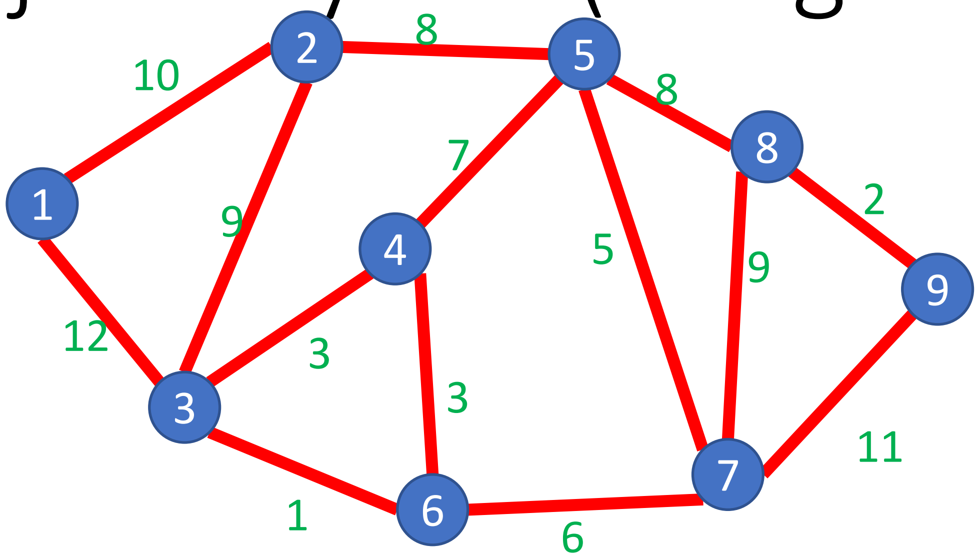
Check if Edge Exists: $\Theta(n)$

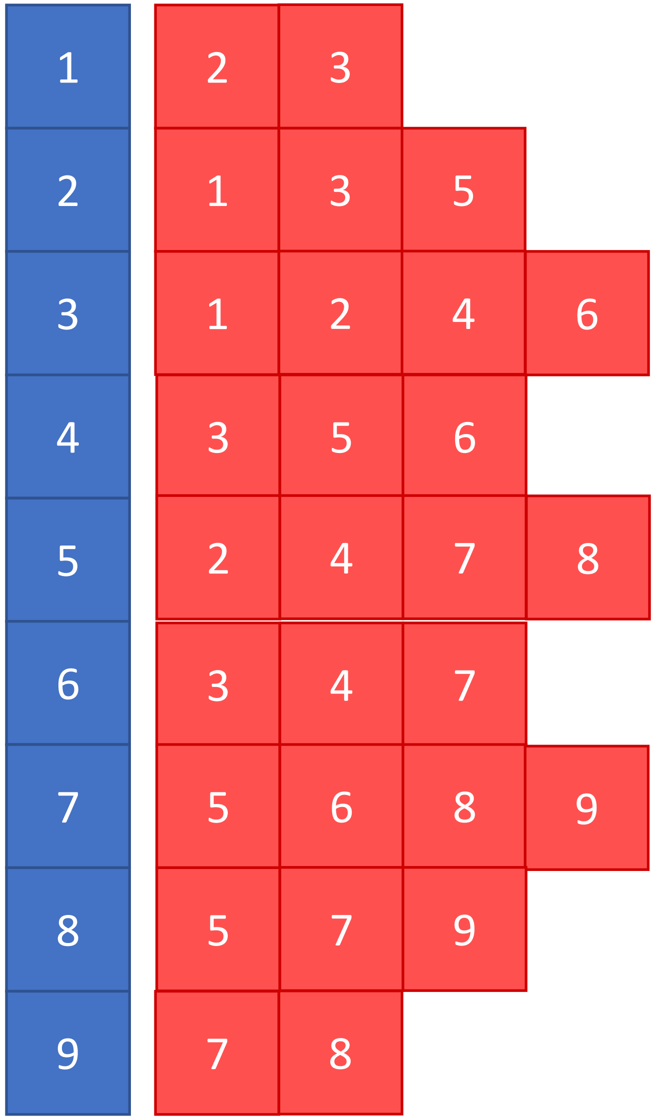
Get Neighbors (incoming): $\Theta(n+m)$

Get Neighbors (outgoing): $\Theta(\deg(v))$

$|V| = n$
$|E| = m$

Adjacency List (Weighted)





Time/Space Tradeoffs

Space to represent: $\Theta(n+m)$

Add Edge: $\Theta(1)$

Remove Edge: $\Theta(1)$

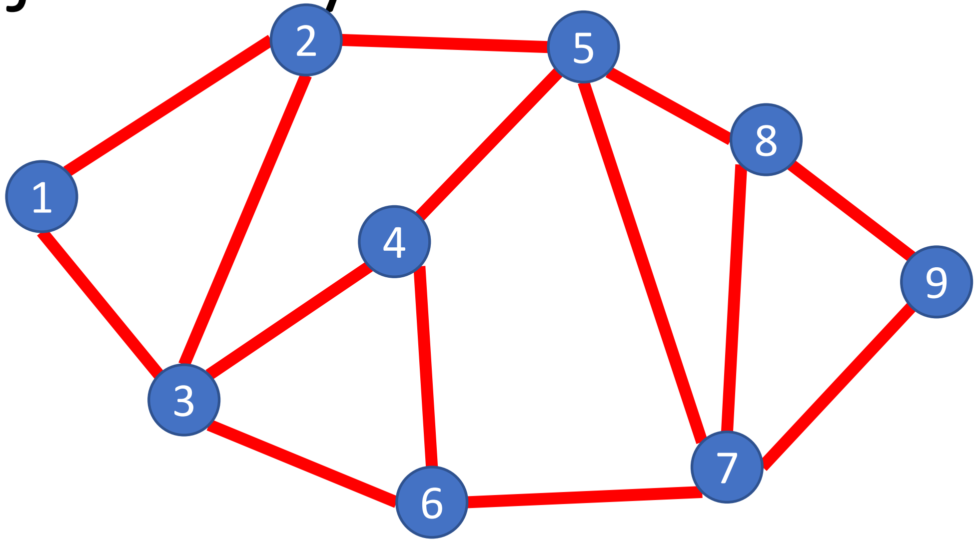
Check if Edge Exists: $\Theta(n)$

Get Neighbors (incoming): $\Theta(?)$

Get Neighbors (outgoing): $\Theta(?)$

$|V| = n$
$|E| = m$

Adjacency Matrix



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(?)$

Add Edge: $\Theta(?)$

Remove Edge: $\Theta(?)$

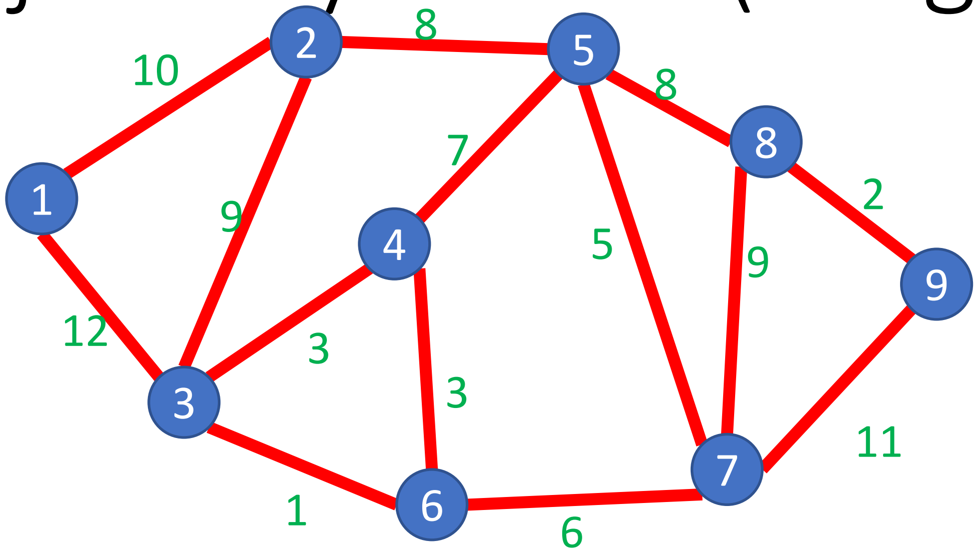
Check if Edge Exists: $\Theta(?)$

Get Neighbors (incoming): $\Theta(?)$

Get Neighbors (outgoing): $\Theta(?)$

$$|V| = n$$
$$|E| = m$$

Adjacency Matrix (weighted)



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n^2)$

Add Edge: $\Theta(1)$

Remove Edge: $\Theta(1)$

Check if Edge Exists: $\Theta(1)$

Get Neighbors (incoming): $\Theta(n)$

Get Neighbors (outgoing): $\Theta(n)$

$$|V| = n$$
$$|E| = m$$

Aside

- Almost always, adjacency lists are the better choice
- Most graphs are missing most of their edges, so the adjacency list is much more space efficient and the slower operations aren’t that bad