Data Structures
Lec-12 Sorting

Course Project

- Explore different methods (**at least three**) to find out all unique elements in a given list of numbers.
- **Due at 23:59 on Apr 25, 2024**
- Use different kinds of data structures (but your own implement.)
- Need to test on OJ for all methods
- Submit a zip file on Canvas, including
  - Codes of all methods
  - The screenshots of all passes on OJ
  - Final report
- 5% of the final marks

Review: What are suffixes?

Given a string (or text)

$T = t_1 \ t_2 \ t_3 \ \dots \ t_{n-1} \ t_n$

then it has n suffixes,

they are

$\mathrm{Suffix}(T,i) = S[i]$

$= t_i \ t_{i+1} \ t_{i+2} \ \dots \ t_n$

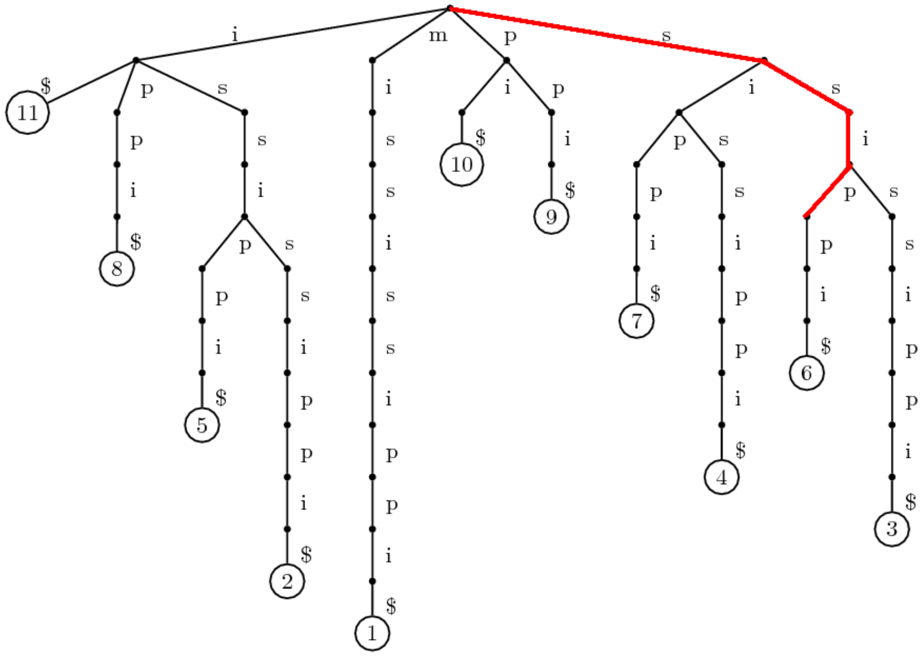
for

$1 \le i \le n$

T =	innovation
S[1]=	innovation
S[2]=	nnovation
S[3]=	novation
S[4]=	ovation
S[5]=	vation
S[6]=	ation
S[7]=	tion
S[8]=	ion
S[9]=	on
S[10]=	n

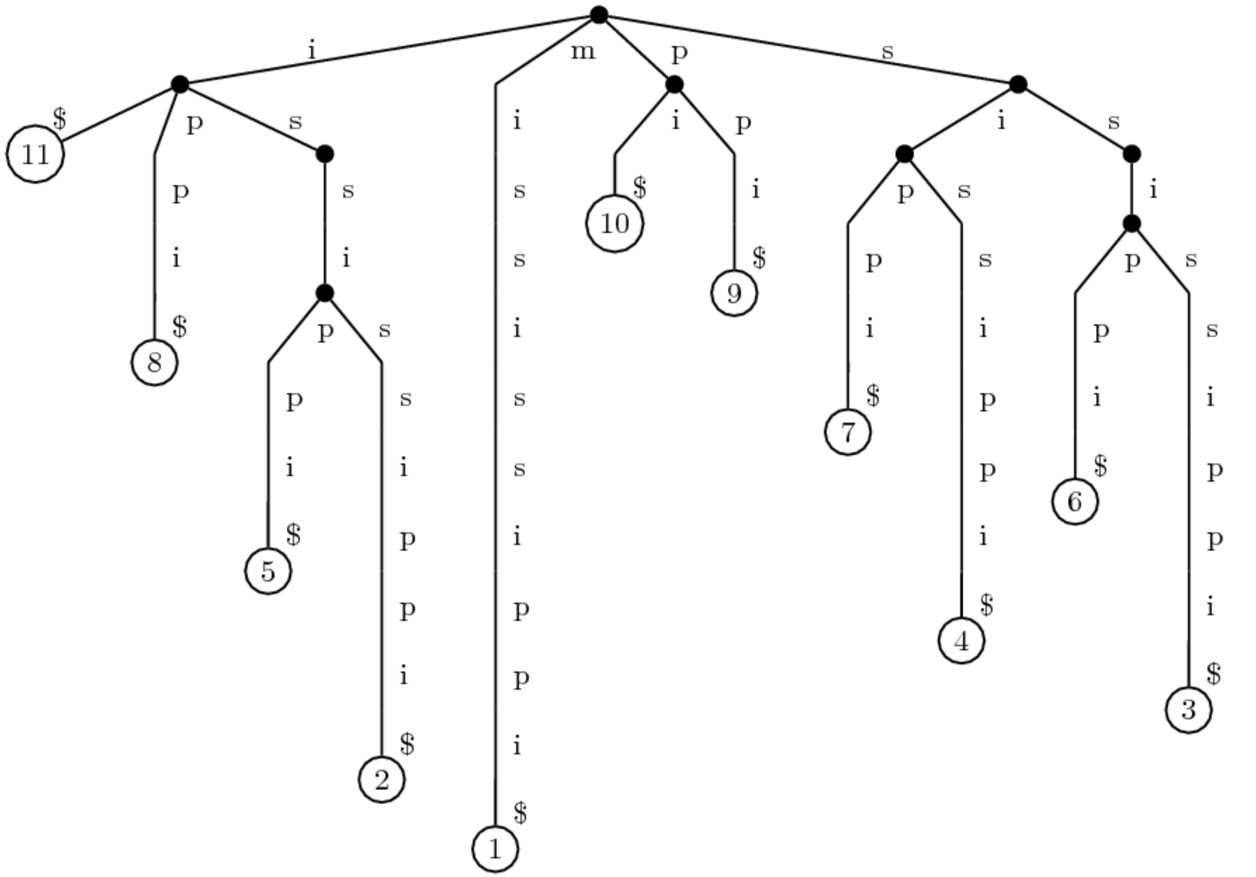
Review: Trie (“retrieval”) (cont.)

- Put all suffixes into a Trie!
  - Every top-down path starts from root corresponds to a substring
  - Those paths ending with a leaf correspond to suffixes
- Complexity
  - Preprocessing **$O(T^2)$**
  - Matching **$O(P)$**



Review: Suffix Tree

- Trie of all suffixes: too many nodes! **O(T^2)**
- Suffix Tree
  - Compact Trie
  - **O(T)** nodes



Review: Suffix Array

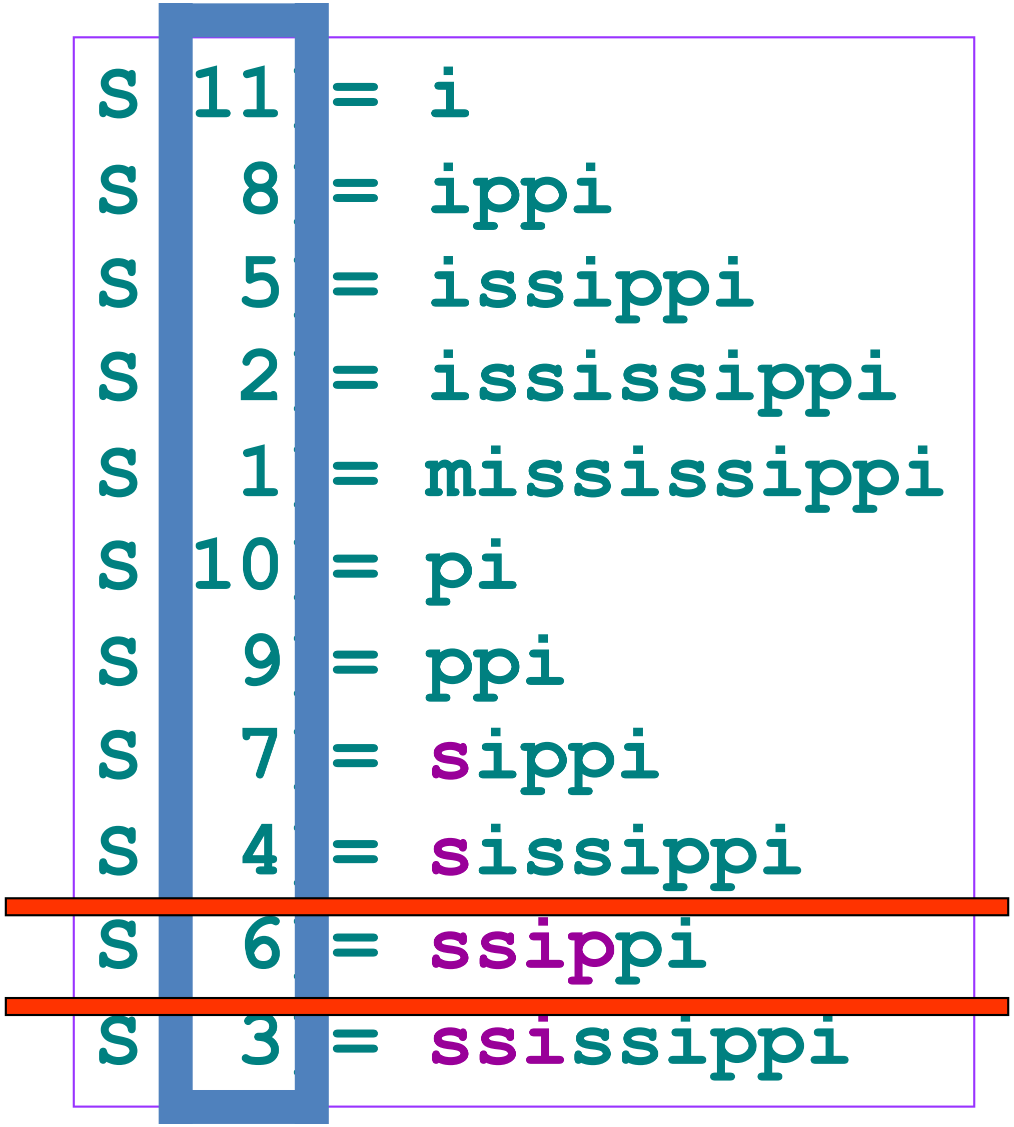
- Suffix Array (SA) : ***sorted indexes*** of all suffixes of a string in lexicographical order.
- Given the Suffix Array of **T**
  - Find all occurrences of **P** by a naïve binary search in **O(P * log T)** time
  - Can be done in **O(P)** time (more advanced topic)
- But how to get the Suffix Array?
  - In another words: How to sort suffixes?

Review: Match Char by Char

Suffix Array

T= mississippi

Search **ssip**



```
S 11 = i
S  8 = ippi
S  5 = issippi
S  2 = ississippi
S  1 = mississippi
S 10 = pi
S  9 = ppi
S  7 = sippi
S  4 = sissippi
S  6 = ssippi
S  3 = ssissippi
```

Review: Sorting Suffixes

- **QSort** or other comparison-based method
  - **$O(n^2 \log n)$**
  - Much faster in practice (real world problem)
- **Radix Sort**
  - **$O(n^2)$**
- **Doubling Algorithm**
  - Udi Manber and Gene Myers, *Suffix arrays: a new method for on-line string searches***,** SODA 1990, SIAM J. Comput. 1993
  - **$O(n \log n)$**
- **Skew Algorithm** (for integer alphabet)
  - Kärkkäinen, Sanders and Burkhardt, *Linear Work Suffix Array Construction*, Journal of the ACM, 2006
  - **$O(n)$**

Review: L-order

- Definition: $S[i] \leq_L S[j]$
  - Use the first L chars of each suffixes as key

- Examples:
  - ippi $<_2$ issippi
  - ssissippi $=_3$ ssippi
  - ssissippi $>_4$ ssippi

Review: Doubling Algorithm

- Sort by 1-order ($\leq_1$)
- Sort by 2-order ($\leq_2$)

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississsippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Review: Doubling Algorithm

- Sort by 1-order ($\leq_1$)
- Sort by 2-order ($\leq_2$)

Then…

- Sort by 3-order ($\leq_3$) ?
  - No! This is what ***Radix Sort*** do for general strings.
  - But we are sorting suffixes!
- Sort by 4-order ($\leq_4$) directly

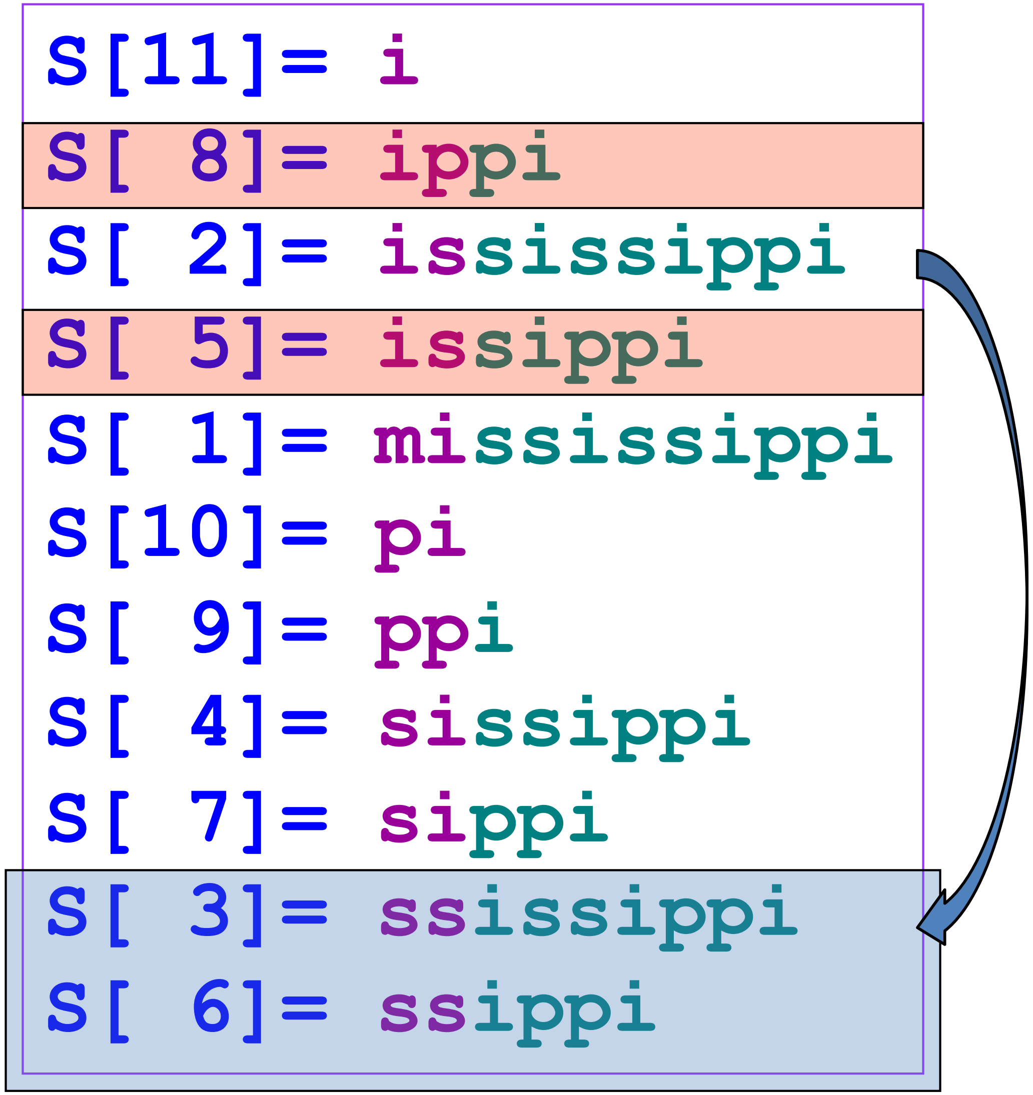
```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Review: Extend 2-order to 4-order

- To compare
  - S[3]=ssissippi
  - S[6]=ssippi
- ss is sippi

  ss ip pi
- issippi$>_2$ippi

  from S[5]$>_2$S[8]



S[11]= i

S[8]= ippi

S[2]= ississippi

S[5]= issippi

S[1]= mississippi

S[10]= pi

S[9]= ppi

S[4]= sissippi

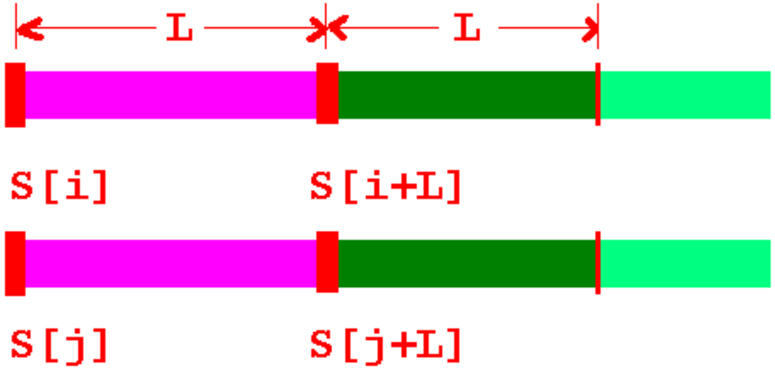
S[7]= sippi

S[3]= ssissippi

S[6]= ssippi

Review: Extend L-order to 2L-order

- If we have the L-order, then 2L-order could be obtain by
  - $S[i] <_L S[j]$ → $S[i] <_{2L} S[j]$
  - $S[i] >_L S[j]$ → $S[i] >_{2L} S[j]$
  - $S[i] =_L S[j]$
    - $S[i+L] <_L S[j+L]$ → $S[i] <_{2L} S[j]$
    - $S[i+L] >_L S[j+L]$ → $S[i] >_{2L} S[j]$



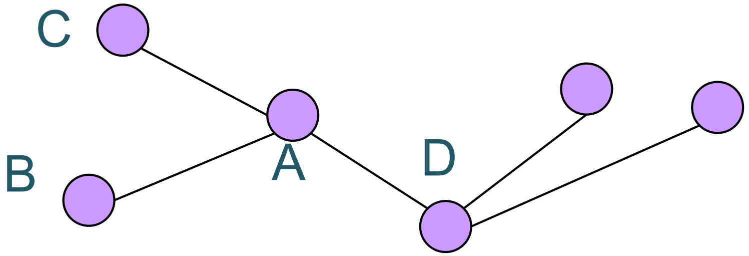
Review: Terms and definitions

- A graph G consists of:
  - A non-empty set of vertices: V
  - A set of edges: E
  - E & V are related in a way that the vertices on both ends of an edge are members of V
  - Usually written as $G = (V, E)$

- Usually, Vertices are used to represent a position or state meanwhile Edges are used to represent a transaction or relationship

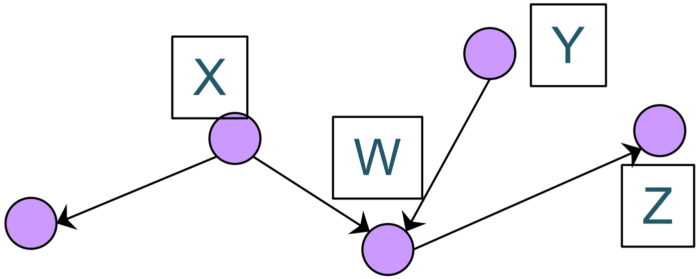
Review: Terms and definitions: Degree

- Degree of a vertex is the number of edges connecting to it



Node A is having a degree of **3** (connects B, C, and D)

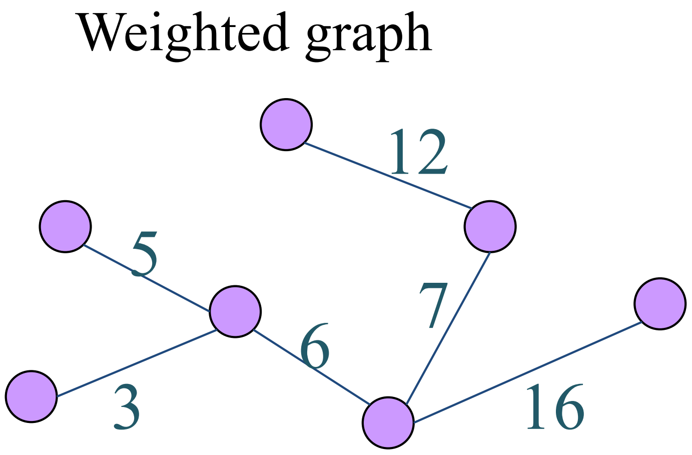
- For directed graph, degree is further classified as in-degree *(to this vertex)* & out-degree *(from this vertex)*

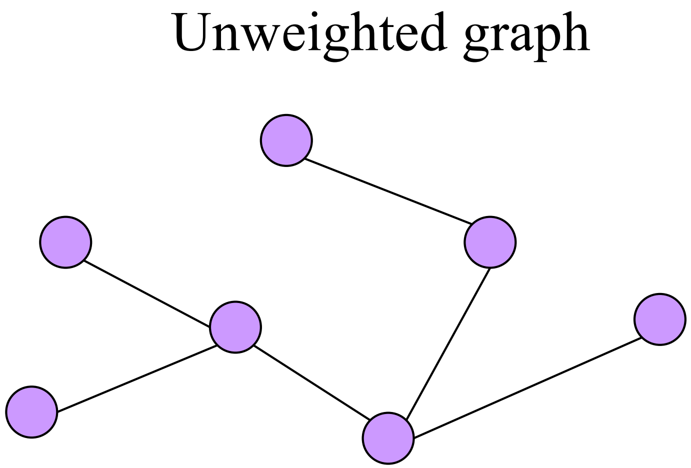


Node W is having an
in-degree of **2** (X, Y)
and an out-degree of **1** (Z)

Review: Terms and definitions: Weights

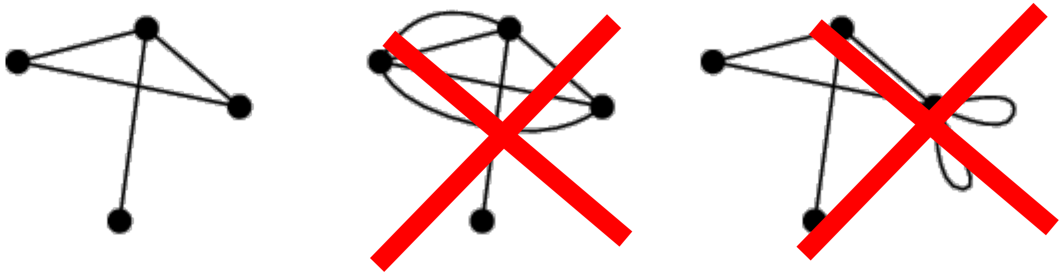
- Graph can be unweighted or weighted, in which a value is associated with each edge.
- Graph can be undirected or directed. In directed graph, the weights of edges going in opposite directions can be different.
- For example:
  - *Whether a bus can go from Shatin to CityU:* Unweighted (=1...)
  - *The bus fee it takes from Shatin to CityU:* Weighted



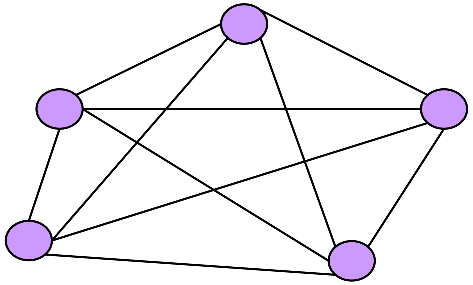


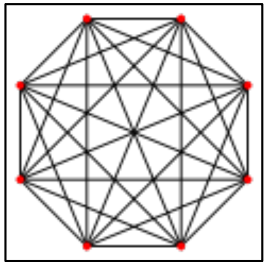
Review: Terms and definitions: Simple/Complete

- Simple graph:
  - an un-weighted, undirected graph containing no graph loops or multiple edges



- Complete graph:
  - A simple graph in which every pair of vertices are connected directly.
  - If number of vertices (||*V*||) = *n*, number of edges = n(n-1)/2





Review: Representations of graphs

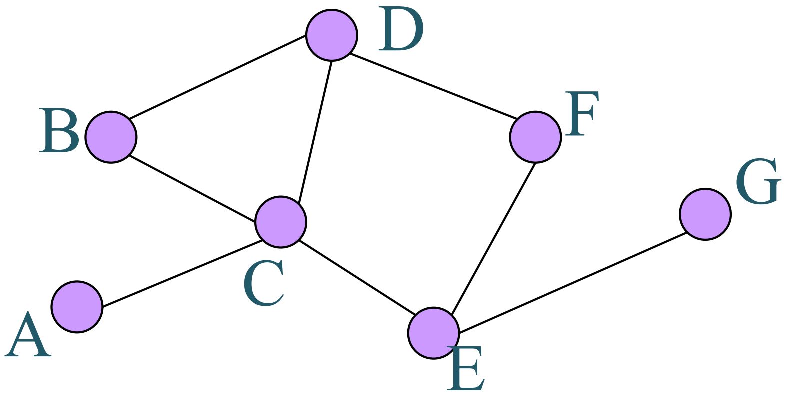
- When working with graph, we always perform one of the following operation:
  - Get the list of vertices connecting a given vertex.
  - Is vertices A & B connected?
  - What is the weight of edge from A to B?
  - What is the in/out degree of a vertex?

- 3 standard representations
  - Adjacency Matrix
  - Adjacency List
  - Compressed Sparse Row (CSR)

Review: Adjacency Matrix

- Use *N*N* 2D array to represent the weight (or T/F) of one vertex to another

An undirected graph



	A	B	C	D	E	F	G
A	0	0	1	0	0	0	0
B	0	0	1	1	0	0	0
C	1	1	0	1	1	0	0
D	0	1	1	0	0	1	0
E	0	0	1	0	0	1	1
F	0	0	0	1	1	0	0
G	0	0	0	0	1	0	0

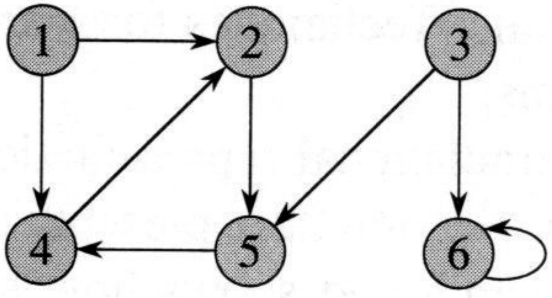
Review: Adjacency Matrix

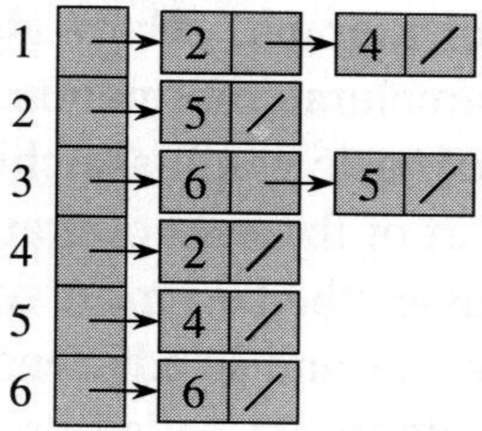
- Advantage: Fast query on edge weight & connection

- Disadvantage:
  - For undirected graph, only half of the array is used
  - Total memory used: N^2 (what if num of vertex = 10K?)
  - Waste memory if the graph is sparse
    - i.e. #Edge is much smaller than half of $(\#\text{Vertex})^2$, a large proportion of the array will be zero
  - Slow when querying the list of neighboring vertices if the graph is sparse

Review: Adjacency List

- Use link list (or equivalent) to store the list of neighboring vertex.
- Save memory if the graph is sparse.
- Query on edge weight / connection can be slow.
- Graph update is slow (especially if one have to maintain order of neighbors)
- Enumeration of all neighbors is fast



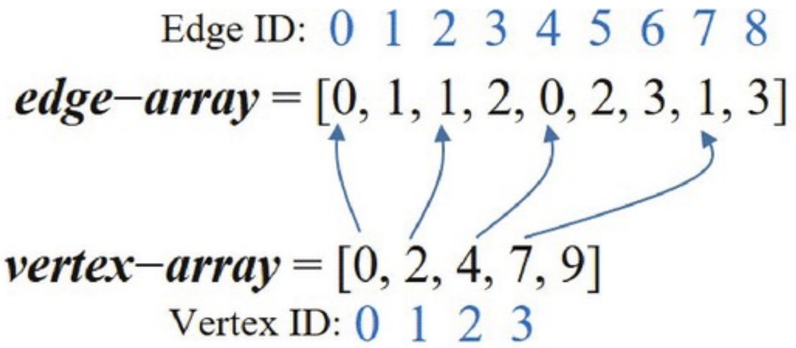


Review: Compressed Sparse Row (CSR)

- An efficient way to store sparse matrices or graphs
- The edge array is sorted by the source of each edge, but contains only the targets for the edges.
- The vertex array stores offsets into the edge array, providing the offset of the first edge outgoing from each vertex.

$$
\boldsymbol{A} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{array}
$$

Adjacency matrix



Edge ID: 0 1 2 3 4 5 6 7 8

edge−array = [0, 1, 1, 2, 0, 2, 3, 1, 3]

vertex−array = [0, 2, 4, 7, 9]

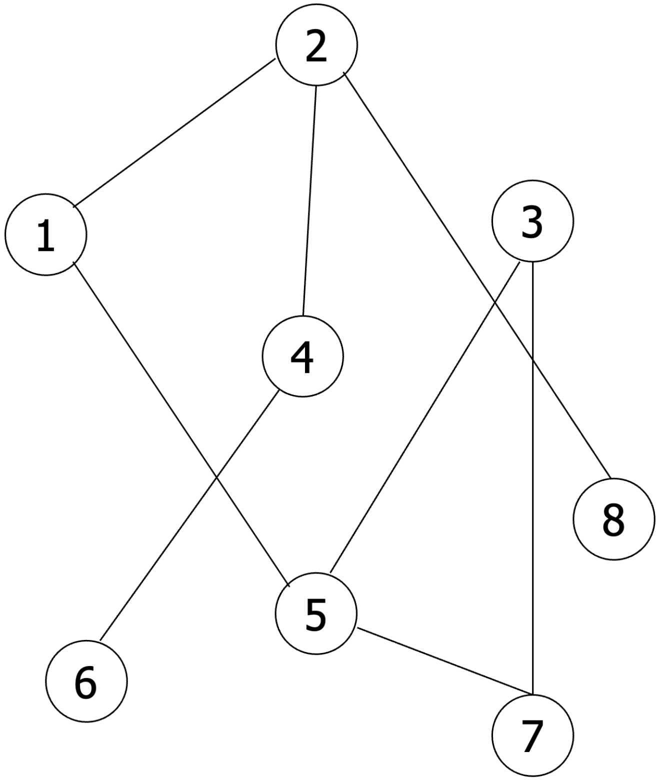
Vertex ID: 0 1 2 3

Review: Graph Searching

- To determine whether two vertices are connected (indirectly via some intermediate)
  - A is a relative of B, B is a relative of C, are A & C relative?
- To list out all members of a *connected-component*
  - List out all the direct/indirect family members of A…
- To find the shortest path (of un-weighted graph) from one vertex to another
  - Travel from Shatin to Central with minimum number of changes of transportation…
- TWO algorithms:
  - DFS (Depth First Search)
  - BFS (Breadth First Search)

Review: DFS on Graphs

- Go as deep as you can
- Example DFS order (starting from 1):
  - 1,2,4,6,8,5,3,7
  - 1,5,7,3,2,8,4,6
- Using Stack to store nodes
  - Put the starting node into the stack
  - Repeat checking the top
- If top is unvisited
  - Print this element
- If top has unvisited neighbors
  - Push one of the neighbors on stack
- If top has no unvisited neighbors
  - Pop one element

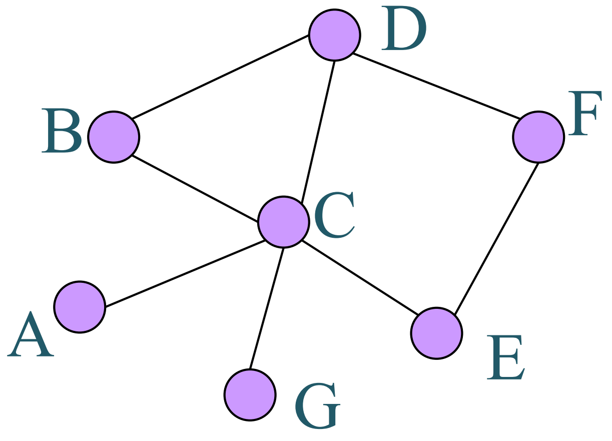


Review: Depth first search (DFS)

- Starts with vertex *v*:

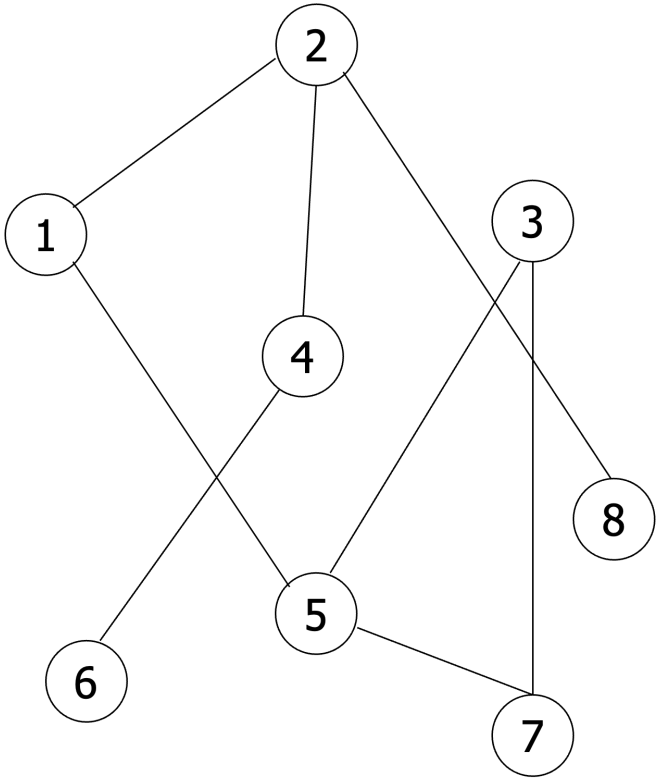
```
DFS (v) {
    visited[v] = true;
    for each vertex w adjacent to v {
      if (! visited[w])
          DFS (w); //Recursion
    }
}
```

DFS(A) = A, C, B, D, F, E, G

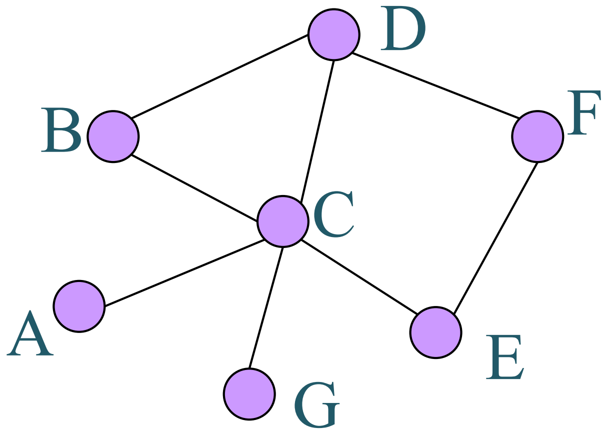


Review: BFS on Graphs

- Go as broad as you can
- Example BFS order (starting from 1):
  - 1,2,5,4,8,3,7,6
  - 1,5,2,7,3,8,4,6
- Using Queue to store nodes
  - Put the starting node into queue
  - Repeat the following “Remove”
- Remove:
  - Remove a node from the queue
  - Print this element
  - Insert all his unvisited (haven’t been in the queue) neighbors into the queue



Review: BFS (v)



```
BFS(v) {
   visited[v] = true;
   Enqueue(v);
   While queue not empty {
   x = Dequeue();
         for each vertex w adjacent to x {
         if (! visited[w]) {
               Enqueue (w);
               visited[w] = true;
        }
      }
   }
}
```

BFS(A) = A, C, B, D, E, G, F

Review: Application 1: Spanning Trees

- Given (connected) graph G(V,E),

  a spanning tree T(V',E'):

  - Is a subgraph of G; that is, V' $\subseteq$ V, E' $\subseteq$ E.
  - Spans the graph (V' = V)
  - Forms a tree (no cycle);
  - So, E' has |V| -1 edges

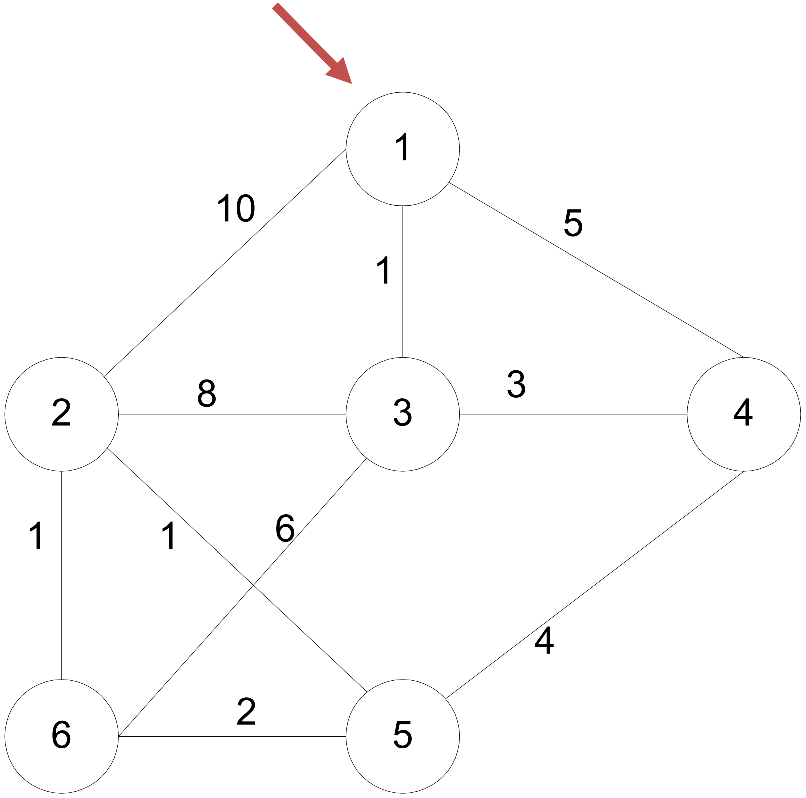
Review: Minimum Spanning Trees (MST)

- Edges are weighted: find minimum cost spanning tree
- Applications
  - Find cheapest way to wire your house
  - Find minimum cost to send a message on the Internet

Review: Prim’s algorithm

Starting from empty T, choose a vertex at random and initialize

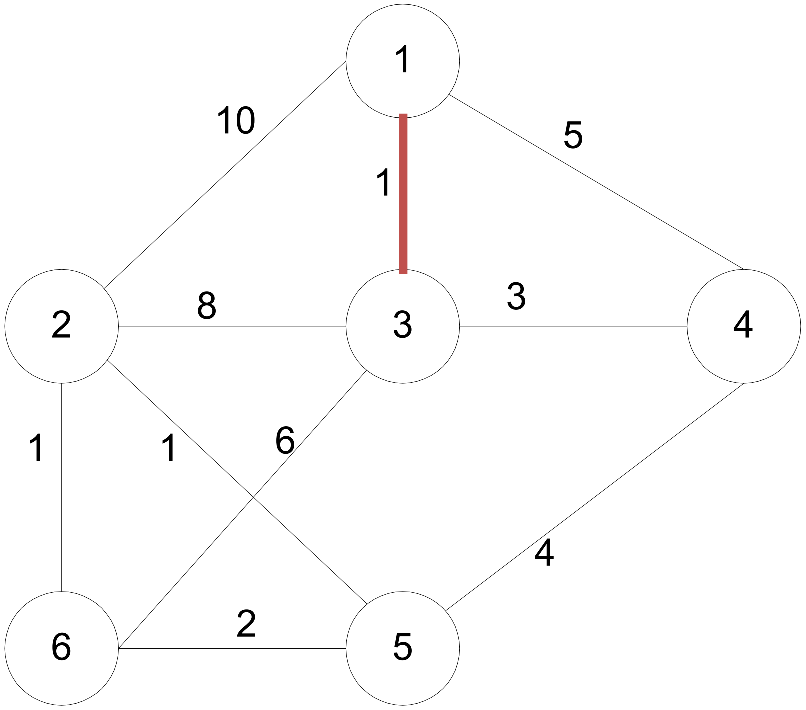
V’ = {1), E’ ={}



Review: Prim’s algorithm

Choose the vertex u not in V’ such that edge weight from u to a vertex in V’ is minimal (greedy!)

V’={1,3} E’= {(1,3) }



Review: Prim’s algorithm

Repeat until all vertices have been chosen

Choose the vertex u not in V’ such that edge weight from v to a vertex in V’ is minimal (greedy!)

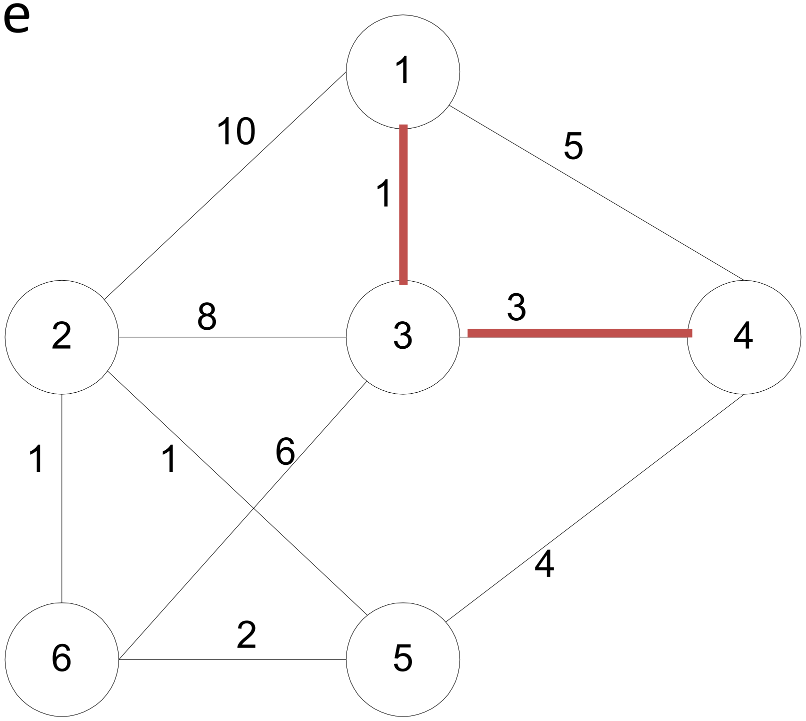
V’= {1,3,4} E’= {(1,3),(3,4)}

V’={1,3,4,5} E’={(1,3),(3,4),(4,5)}

….

V’={1,3,4,5,2,6}

E’={(1,3),(3,4),(4,5),(5,2),(2,6)}



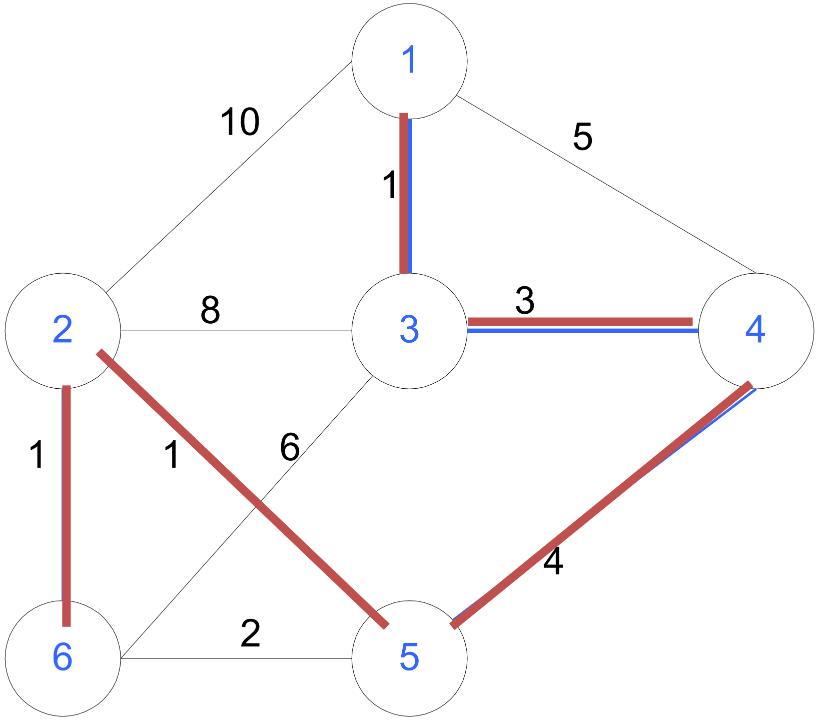
Review: Prim’s algorithm

Repeat until all vertices have been chosen

V={1,3,4,5,2,6}

E’={(1,3),(3,4),(4,5),(5,2),(2,6)}

Final Sum: 1 + 3 + 4 + 1 + 1 = 10



Review: Prim’s algorithm Implementation

- Assume adjacency list representation
  - Initialize connection cost of each node to “inf” and “unmark” them
  - Choose one node, say v and set cost[v] = 0 and prev[v] =0
  - While they are unmarked nodes
    - Select the unmarked node **u** with minimum cost; mark it
    - For each unmarked node **w** adjacent to **u**

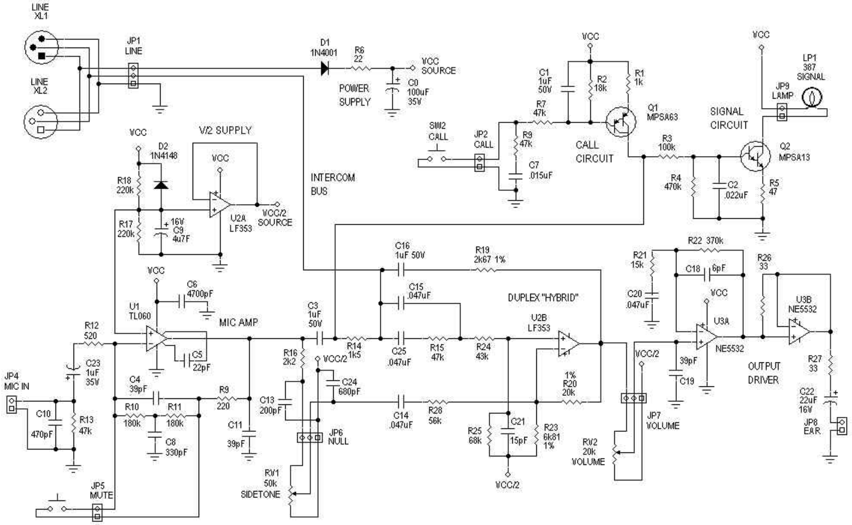
      if cost(u,w) < cost(w) then cost(w) := cost (u,w)

      prev[w] = u

- If the “Select the unmarked node u with minimum cost” is done with binary heap, then O((n+m)logn)

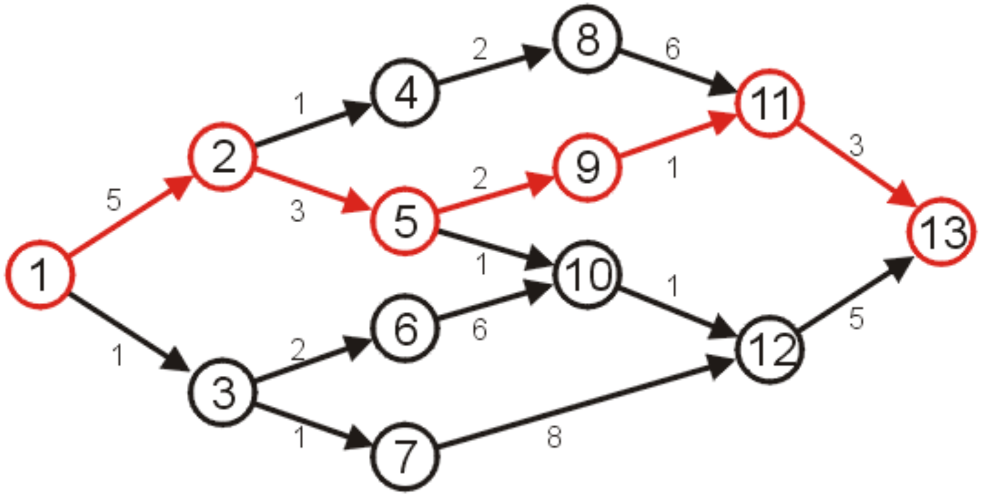
Review: Application 2: Shortest Path

- Given a weighted directed graph, one common problem is finding the shortest path between two given vertices
- Recall that in a weighted graph, the *length* of a path is the sum of the weights of each of the edges in that path
- Application: in circuit design, the time it takes for a change in input to affect an output depends on the shortest path



Review: Shortest Path

- After some consideration, we may determine that the shortest path is as follows, with length 14



- Other paths exists, but they are longer

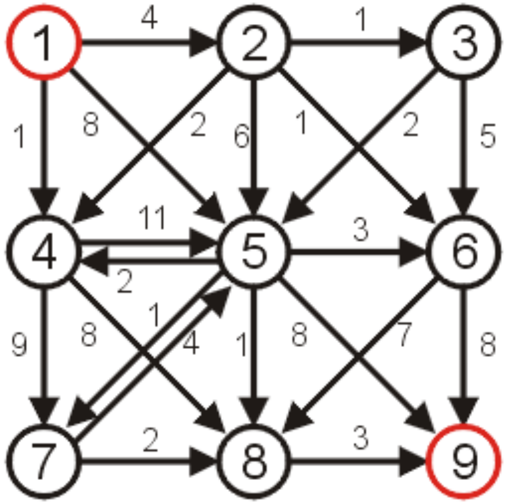
Review: Key Observation

- If the shortest path of *s*->*t* contains the node *v*, then:
  - It will only contain *v* once
  - The path *s* -> *v* must be the shortest path to *v* from *s*.
  - The path *v* -> *t* must be the shortest path to *t* from *v*.

Review: Dijkstra’s algorithm

- Works when all of the weights are positive.
- Provides the shortest paths from a source to **all** other vertices in the graph.

- Consider the following graph with positive weights and cycles.



Review: Dijkstra’s algorithm

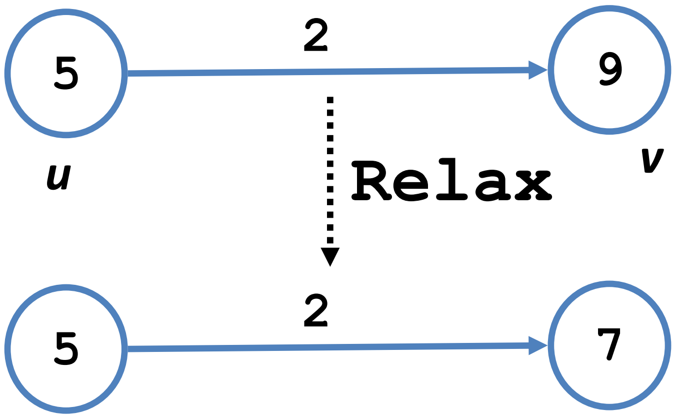
```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    $d[v] \leftarrow \infty$
    $p[v] \leftarrow$ undefined
$S \leftarrow \varnothing$
$Q \leftarrow V$          ▹ $Q$ is a priority queue maintaining $V - S$

while $Q \neq \varnothing$
    $u \leftarrow$ Extract-Min($Q$)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
        if $d[v] > d[u] + w(u, v)$
            $d[v] \leftarrow d[u] + w(u, v)$
            $p[v] \leftarrow u$
```

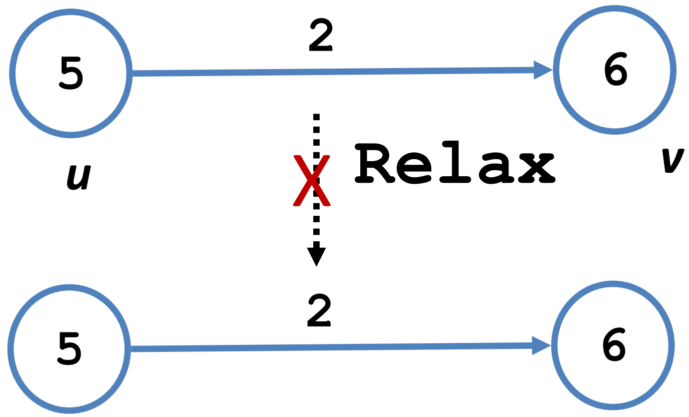
relaxation

Review: Relaxation

- Maintaining this shortest discovered distance d[*v*] is called **relaxation**:

```
Relax(u,v,w) {
     if (d[v] > d[u]+w)
          d[v]=d[u]+w;
}
```

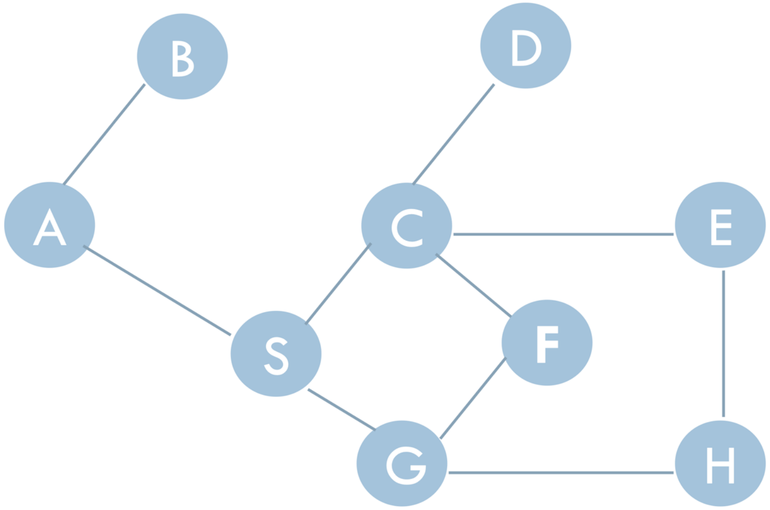




Exercise 1

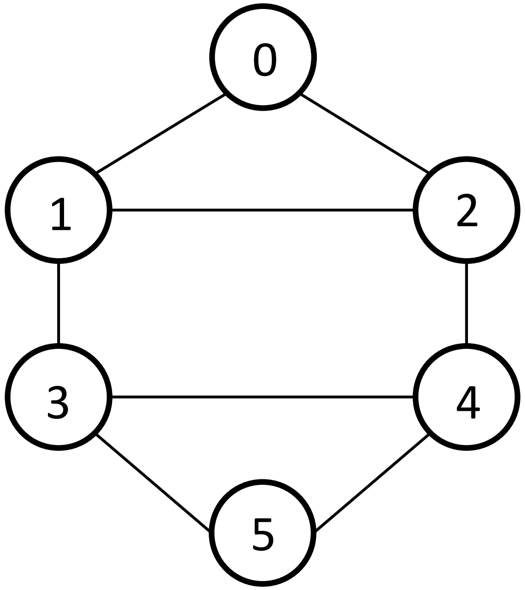
Consider the following graph. In what order will the nodes be visited using a BFS? In what order will the nodes be visited using a DFS? Assume we start from node A.

If there is ever a decision between multiple neighbor nodes in the BFS or DFS algorithms, following the alphabetical order.



Exercise 2

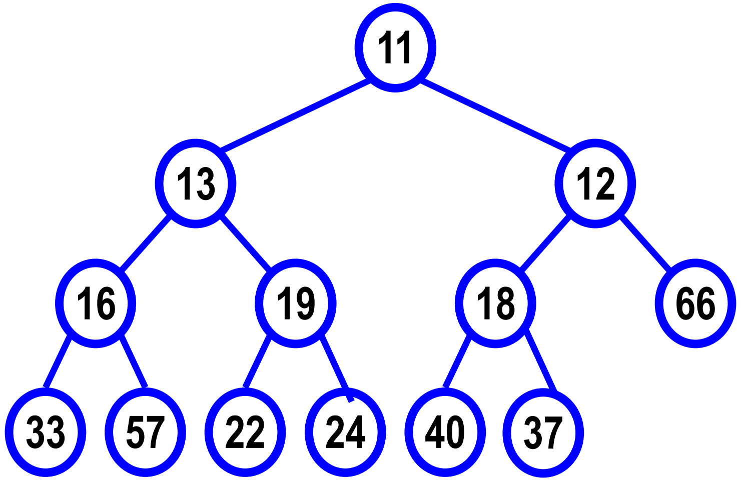
Provide the CSR representation of the following graph.



Outline

- Heapsort
- Merge Sort
- Quicksort
- Bucket Sort
- Stable Sort
- Radix Sort

Heapsort

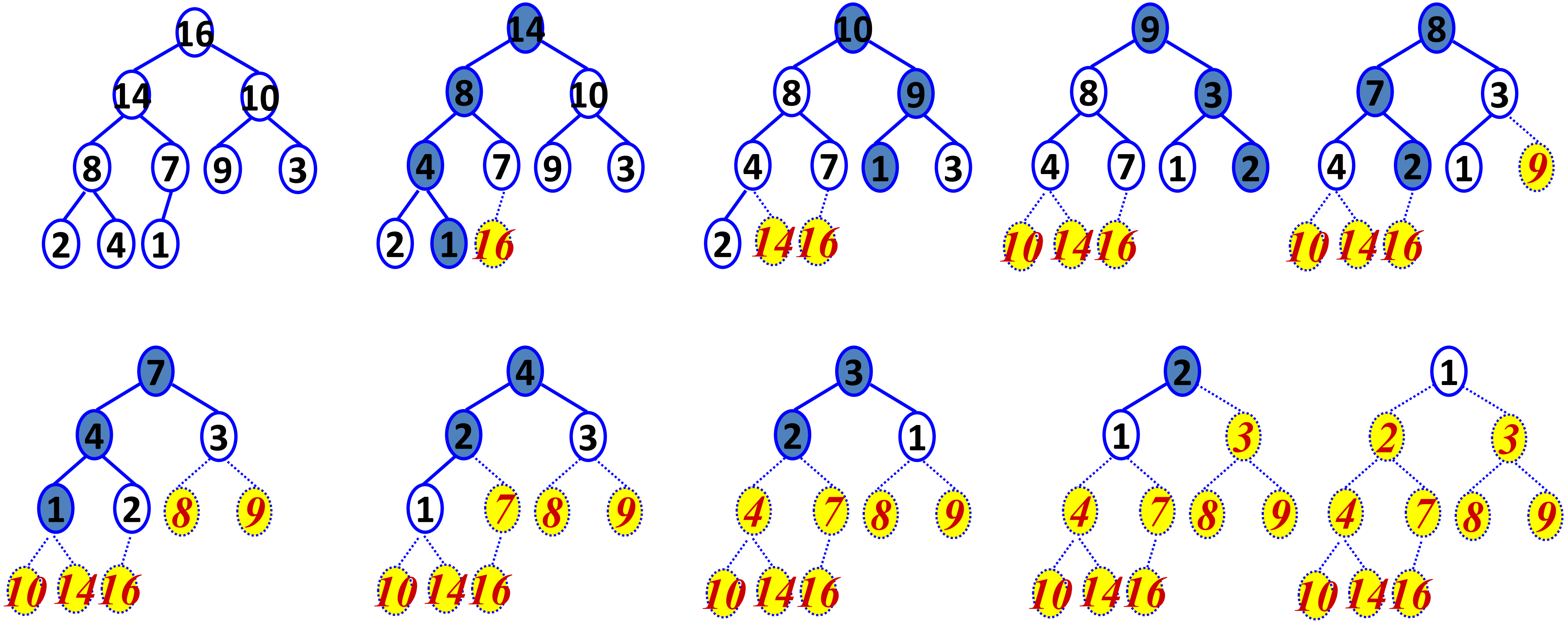


- Step1: Build a Heap according to the input sequence
- Step2: Do DeleteMin() for n times and store the value in a[0],a[1],…a[n-1]

How much storage do we use to do sorting?

Can we improve?

Heapsort

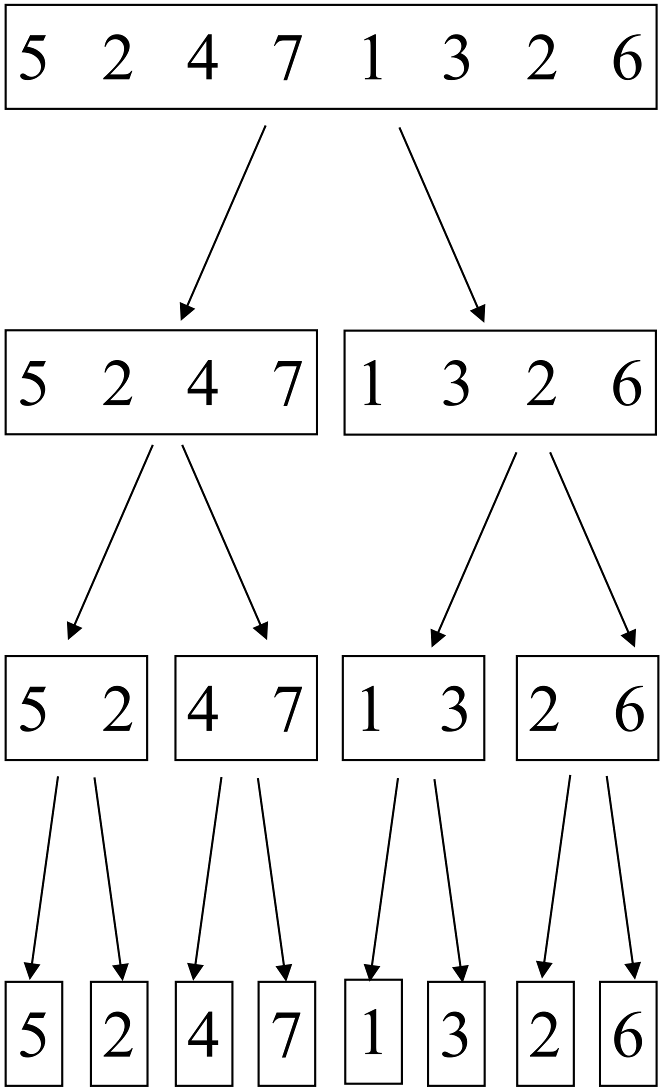


Running time of HeapSort? O( )

Merge Sort

- a ***divide-and-conquer*** approach
- **split** the array into two roughly equal subarrays
- **sort** the subarrays by **recursive** applications of Mergesort and **merge** the sorted subarrays

Merge Sort



At the beginning, a MergeSort is called to sort:

5 2 4 7 1 3 2 6

"So complicated!!, I'll split them and call other MergeSorts to handle."

Then 2 other MergeSorts are called to sort:

5 2 4 7 | 1 3 2 6

Both of them say "Still complicated! I'll split them and call other MergeSorts to handle."

Then 4 other MergeSorts are called to sort:

5 2 | 4 7 | 1 3 | 2 6

All of them say "Still complicated! I'll split them and call other MergeSorts to handle."

Then 8 other MergeSorts are called to sort:

5 | 2 | 4 | 7 | 1 | 3 | 2 | 6

All of them say 'This is easy. No need to do anything.'

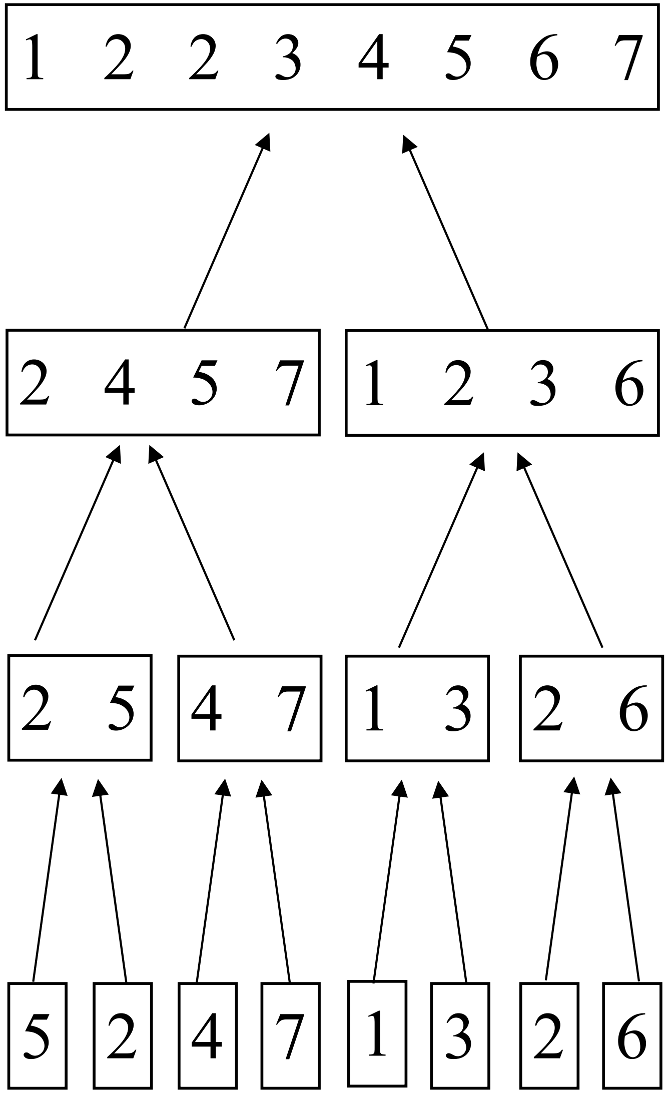
Merge Sort

Then the first MergeSort succeeds and returns.

Then each of the 2 MergeSorts returns the merged numbers.

Then the 4 MergeSorts returns the merged numbers.

Then the 8 MergeSorts return.



The first MergeSort calls Merge to merge the returned numbers

Both MergeSorts call Merge to merge the returned numbers

The 4 MergeSorts call Merge to merge the returned numbers

All of them say ‘This is easy. No need to do anything.’

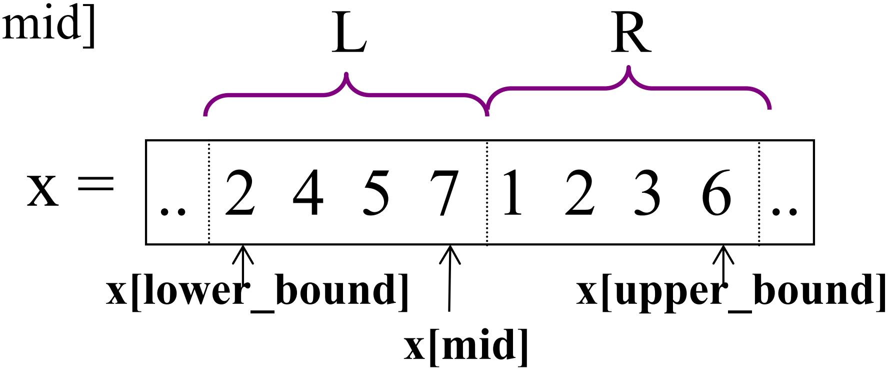
Merge Sort

void merge(**int** *x[ ]*, **int** *lower_bound*, **int** *mid*, **int** *upper_bound*)

-- merges 2 sorted sequences:

L: x[lower_bound], x[lower_bound+1], … , x[mid]

R: x[mid+1], x[mid+2] , …, x[upper_bound]



Step 1: Continuously copy the smallest one from L and R to a result list until either L or R is finished.

Step 2: L may still have some numbers not yet copied. So copy them in order.

Step 3: R may still have some numbers not yet copied. So copy them in order.

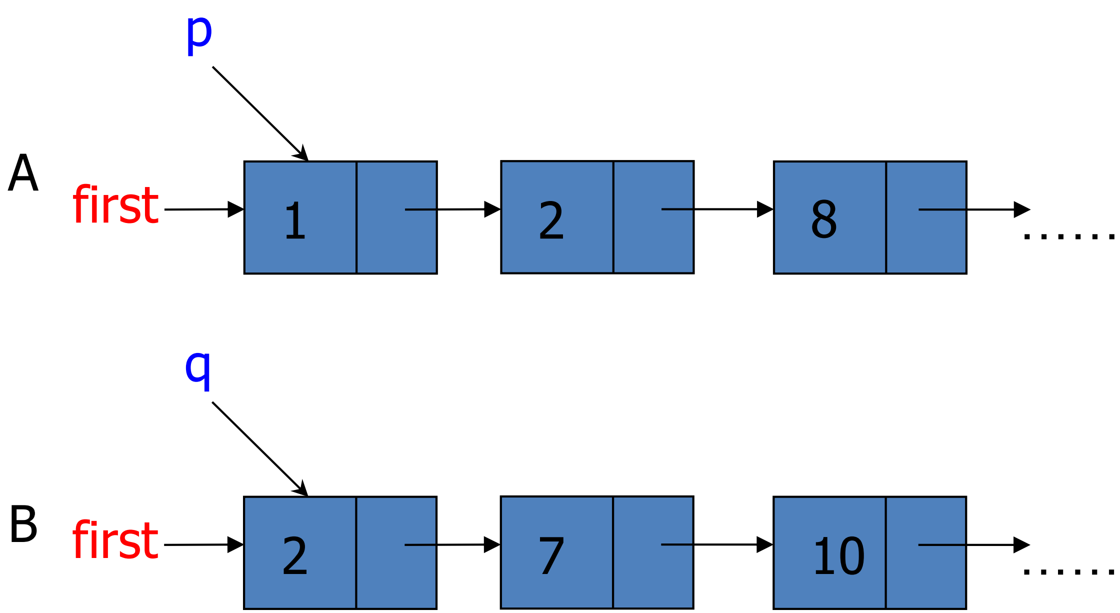
Step 4: Copy the result list back to x.

x = .. 2 4 5 7 1 2 3 6 ..

Result = .. 1 2 2 3 4 5 6 7 ..

Merge Sort

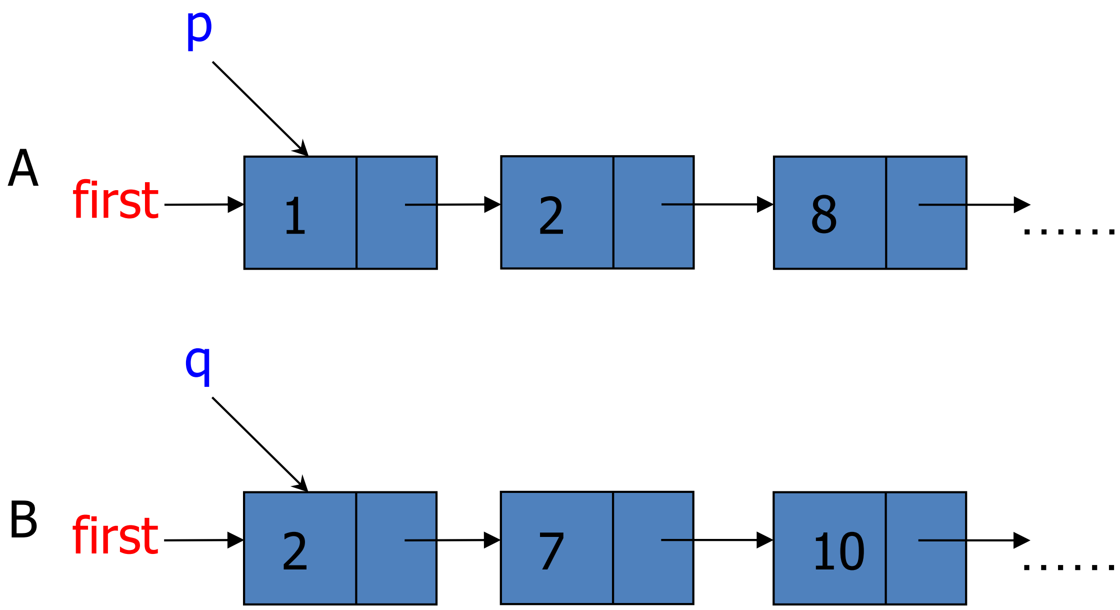
- Linked Implementation



```
p=A->first;
q=B->first;
if (A->first->data < B->first->data) {
            C->first=A->first;
            p=p->next;
}
else{
            C->first=B->first;
            q=q->next;
}
r=C->first;
while (p!=NULL&&q!=NULL){
// Your codes
}
if (p==NULL)
            // Your codes
if (q==NULL)
            // Your codes
```

Merge Sort

- Linked Implementation

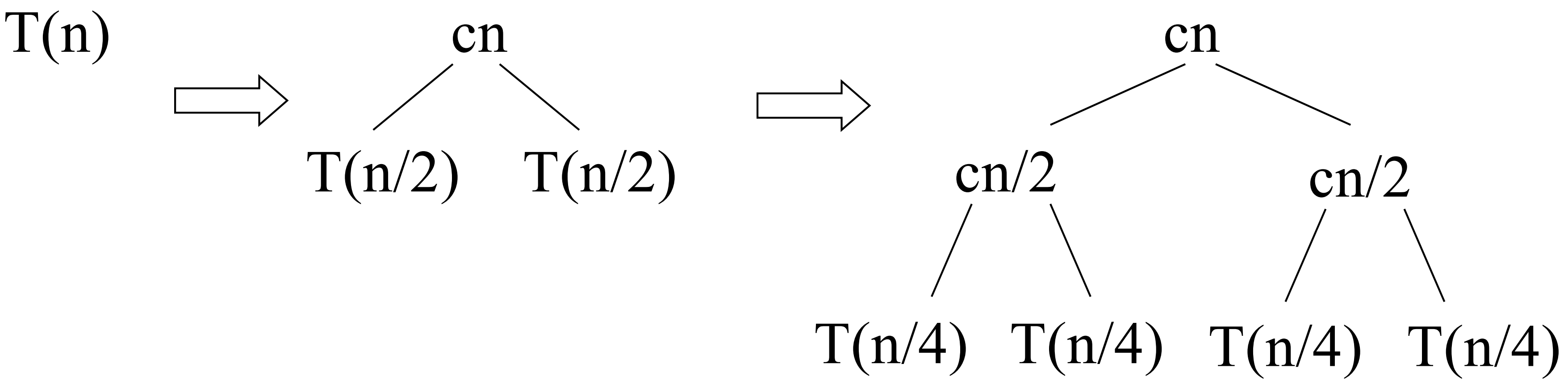


```
…
while (p!=NULL&&q!=NULL){
 if (p->data < q->data) {
     r->next = p;
     p = p->next;
   }
   else {
     r->next = q;
     q = q->next;
   }
   r = r->next;
}
if (p==NULL)
                r->next = q;
if (q==NULL)
                r->next = p;
```

Analysis of Merge Sort

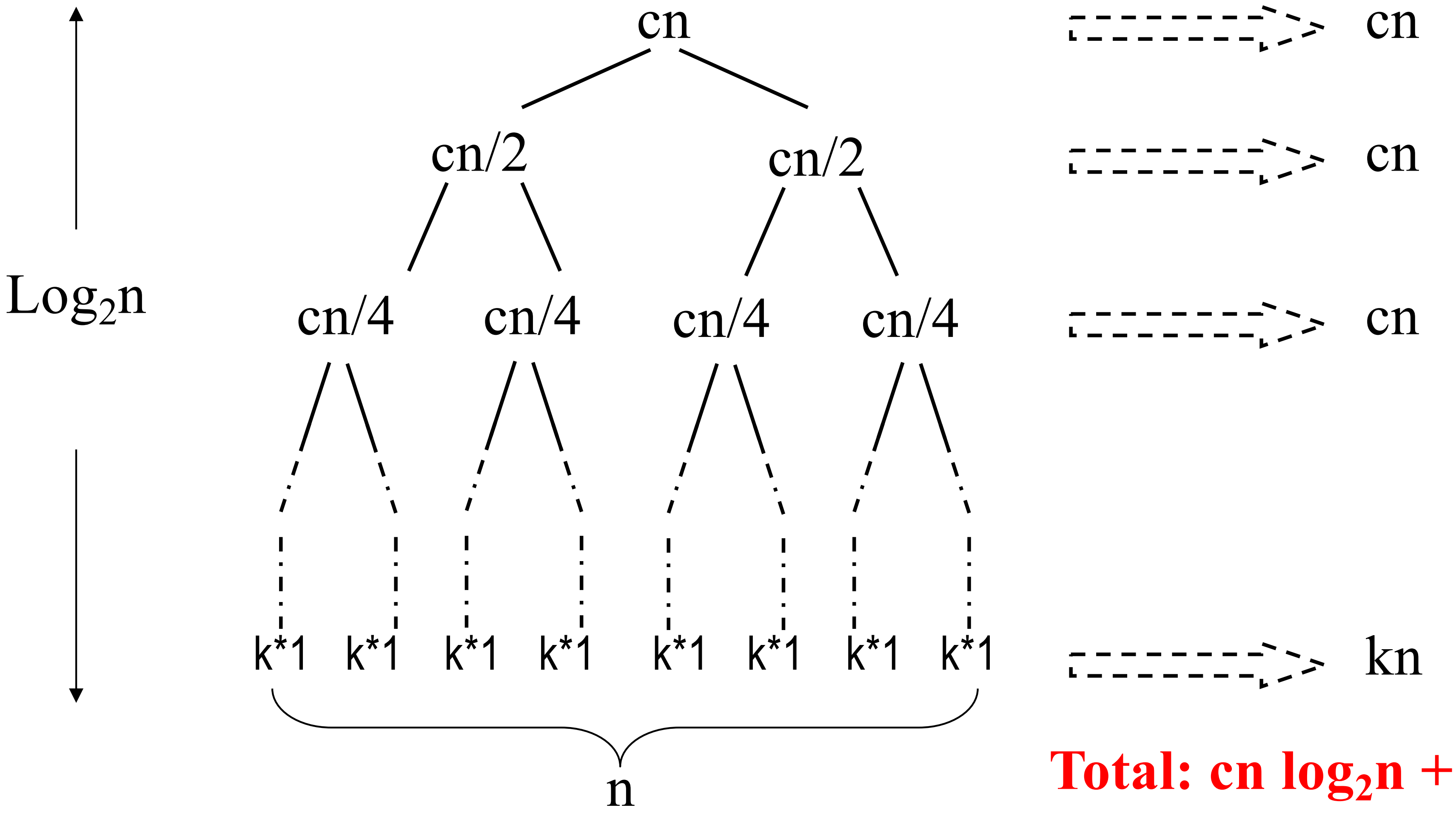
$$T(n) = \begin{cases} k \text{ (a constant)} & \text{if } n=1 \\ 2T(n/2)+cn & \text{if } n>1 \end{cases}$$

Expanding the recursion tree:



Analysis of Merge Sort

Fully Expanded recursion tree:



Total: cn $\log_2 n$ + kn

i.e. T(n) = O(nlogn)

Quicksort

- A “partition-exchange” sorting method:

  Partition an original array into:

  (1) a subarray of small elements

  (2) a single value between (1) and (3)

  (3) a subarray of large elements

  Then partition (1) and (3) independently using the same method.

- For partitioning we need to choose a value ***a***. (simply select ***a*** = ***x***[0])
- During a partition process: pairwise exchanges of elements.

Eg. 25 10 57 48 37 12 92 86 33

=> 12 10 25 48 37 57 92 86 33

A possible arrangement:
simply use first element (ie. 25)
for partitioning

$$\overbrace{\text{Eg. } 25\ 10\ 57\ 48\ 37\ 12\ 92\ 86\ 33}^{x[0..N-1]}$$

=> 12 10 25 48 37 57 92 86 33

a

Quicksort

Original: 25 10 57 48 37 12 92 86 33

Partitioning: Select **a = 25**

Use 2 indices:

down **up**

25 10 57 48 37 12 92 86 33

Move **down** towards **up** until x[**down**]>25

Move **up** towards **down** until x[**up**]<=25 **(*)**

→**down** **up**←

25 10 **57** 48 37 **12** 92 86 33

down **up**

25 10 **12** 48 37 **57** 92 86 33

Swap

up down

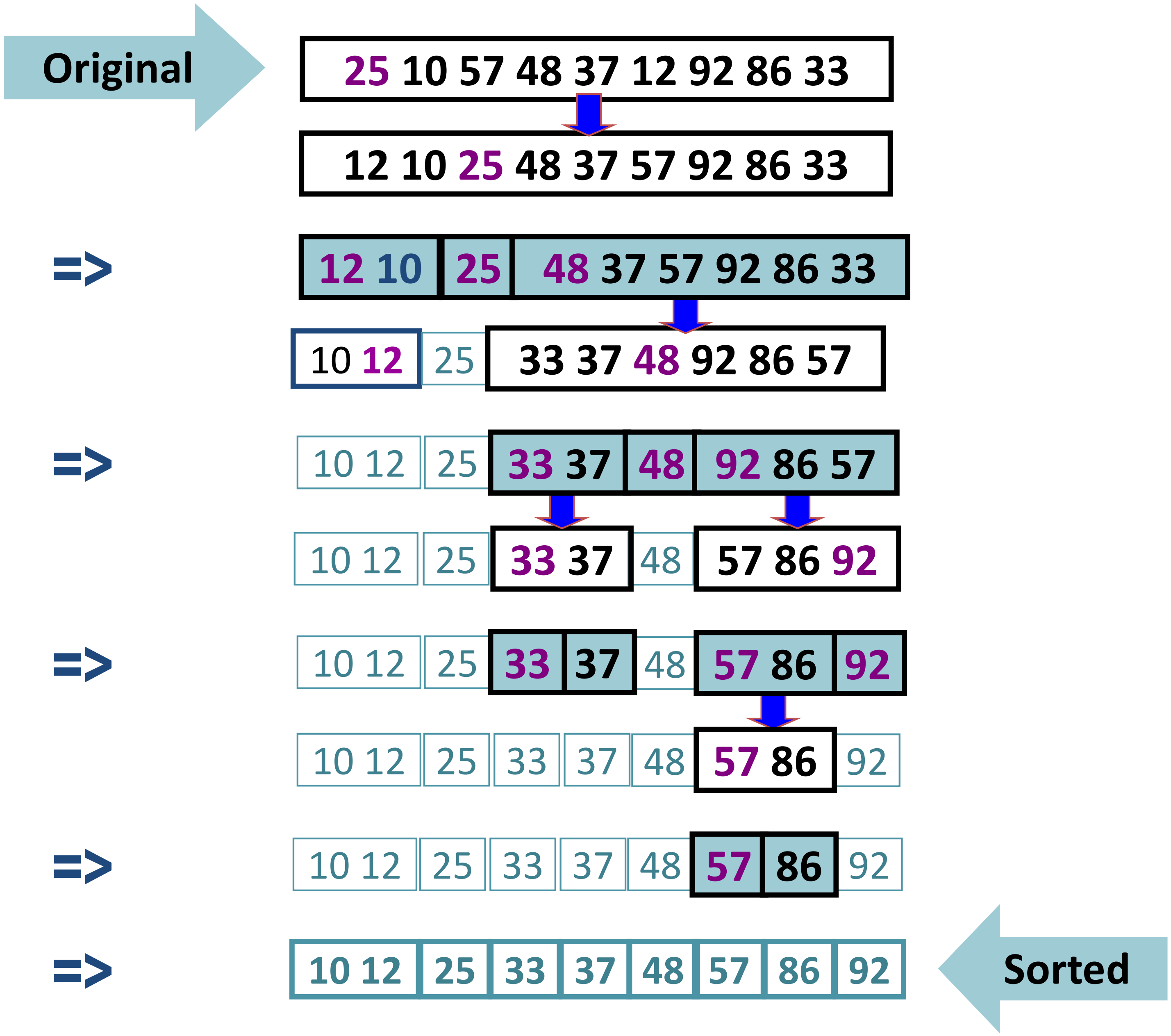
25 10 **12** **48** 37 57 92 86 33

Continue repeat **(*)** until **up** crosses **down** (ie. **down** >= **up**)

12 10 **25** 48 37 57 92 86 33

up is at right-most of smaller partition, so swap **a** with x[**up**]

Quicksort



Original

25 10 57 48 37 12 92 86 33

12 10 25 48 37 57 92 86 33

=> 12 10 25 48 37 57 92 86 33

10 12 25 33 37 48 92 86 57

=> 10 12 25 33 37 48 92 86 57

10 12 25 33 37 48 57 86 92

=> 10 12 25 33 37 48 57 86 92

10 12 25 33 37 48 57 86 92

=> 10 12 25 33 37 48 57 86 92

=> 10 12 25 33 37 48 57 86 92

Sorted

Quicksort

```
void quick_sort(int x[ ], int idLeftmost, int idRightmost)
/* Sort x[idLeftmost].. x[idRightmost] into ascending numerical order. */
{
    int j;

    if (idLeftmost >= idRightmost)
        return; /* array is sorted or empty*/

    partition(x, idLeftmost, idRightmost, &j);
        /* partition the elements of the subarray such that one of the elements
           (possibly x[idLeftmost]) is now at x[j] (j is an output parameter) and
           1) x[i] <= x[j] for idLeftmost <= i < j
           2) x[i] >= x[j] for j<i<= idRightmost
           x[j] is now at its final position */

    quick_sort(x, idLeftmost, j-1);
    /* recursively sort the subarray between positions idLeftmost and j-1 */

    quick_sort(x, j+1, idRightmost);
    /* recursively sort the subarray between positions j+1 and idRightmost */
}
```

Quicksort

```
void partition(int x[ ], int idLeftMost, int idRightMost, int *pj)
{
    int down, up, a, temp;
    a = x[idLeftMost];
    up = idRightMost;
    down = idLeftMost;

    while (down < up)           {
        while ((x[down] <= a) && (down < idRightMost))
            down++;                 /* move up the array */
        while (x[up] > a)
            up--;                   /* move down the array */
        /* interchange x[down] and x[up] */
        temp = x[down];
        x[down] = x[up];
        x[up] = temp;
    }

   x[idLeftMost] = x[up];
   x[up] = a;
   *pj = up;
}
```

Quicksort Analysis

- The best case complexity is O(N log N)

  Each time when a is chosen (as the first element) in a partition, it is the median value in the partition. => the depth of the "tree" is O(log N).

- In worst case, it is O(N^2).
  For most straightforward implementation of Quicksort, the worst case is achieved for an input array that is already in order.

  Each time when a is chosen (as the first element) in a partition, it is the smallest (or largest) value in the partition.

  => the depth of the "tree" is O(N).

- Improvement: choose the pivot at random

  Expected O(N log N) running time

Bucket Sort

- Comparison-based sorting algorithms require Ω(nlogn) time
- But we can sort in O(n) time using more powerful operations
  - When elements are integers in {0,…, M-1}, bucket sort needs O(M+n) time and O(M) space
  - When M=O(n), bucket sort needs O(2n)=O(n) time
- Note: Some books call this *counting sort*

Bucket Sort (cont.)

- **Idea**: Require a counter (auxiliary) array *C[0..M-1]* to count the number of occurrences of each integer in {0,…,M-1}
- **Algorithm**:
  - **Step 1:** initialize all entries in C[0..M-1] to 0
  - **Step 2:** For i=0 to n-1
    - Use A[i] as an array index and increase C[A[i]] by one
  - **Step 3:** For j=0 to M-1
    - Write C[j] copies of value j into appropriate places in A[0..n-1]

Bucket Sort (Example)

- Input: 3, 4, 6, 9, 4, 3 where M=10

Counter array A:

0	1	2	3	4	5	6	7	8	9

- Step 1: Initialization

0	1	2	3	4	5	6	7	8	9
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

- Step 2: Read 3

(A[3] = A[3] + 1)

0	1	2	3	4	5	6	7	8	9
0	0	0	**1**	0	0	0	0	0	0

- Read 4

(A[4] = A[4] + 1)

0	1	2	3	4	5	6	7	8	9
0	0	0	1	**1**	0	0	0	0	0

- Read 6

(A[6] = A[6] + 1)

0	1	2	3	4	5	6	7	8	9
0	0	0	1	1	0	**1**	0	0	0

- Read 9

(A[9] = A[9] + 1)

0	1	2	3	4	5	6	7	8	9
0	0	0	1	1	0	1	0	0	**1**

- Read 4

(A[4] = A[4] + 1)

0	1	2	3	4	5	6	7	8	9
0	0	0	1	**2**	0	1	0	0	1

- Read 3

(A[3] = A[3] + 1)

0	1	2	3	4	5	6	7	8	9
0	0	0	**2**	2	0	1	0	0	1

- Step 3: Print the result (from index 0 to 9)

- Result: 3, 3

0	1	2	3	4	5	6	7	8	9
0	0	0	**2**	2	0	1	0	0	1

- Result: 3, 3, 4, 4

0	1	2	3	4	5	6	7	8	9
0	0	0	0	**2**	0	1	0	0	1

- Result: 3, 3, 4, 4, 6

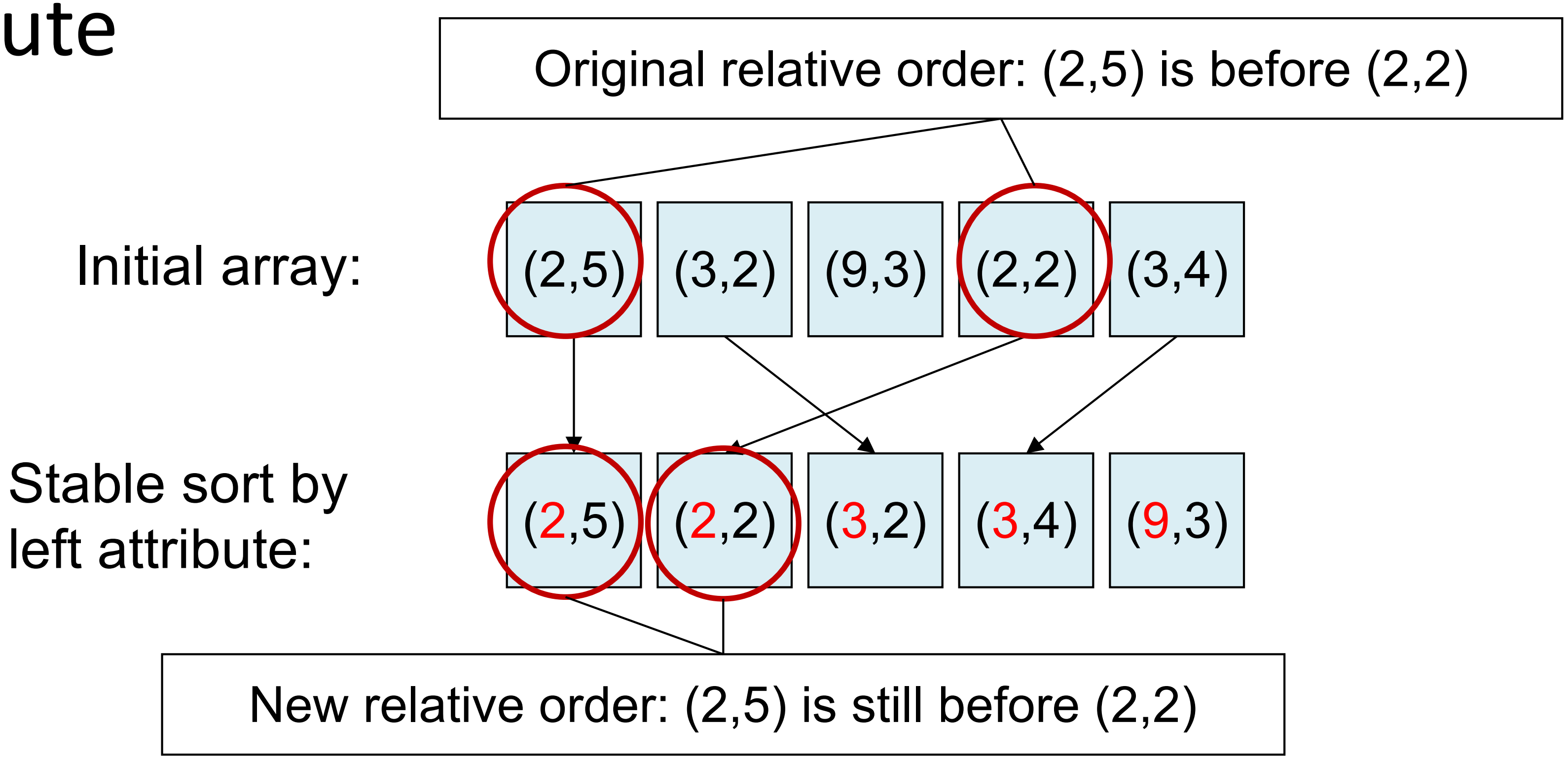
0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	**1**	0	0	1

- Result: 3, 3, 4, 4, 6, 9

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	**1**

Stable Sort

- Definition: A *stable* sorting algorithm is one that preserves the original relative order of elements with equal key
- E.g., suppose the left attribute is the key attribute



Original relative order: (2,5) is before (2,2)

Initial array: (2,5) (3,2) (9,3) (2,2) (3,4)

Stable sort by left attribute: (2,5) (2,2) (3,2) (3,4) (9,3)

New relative order: (2,5) is still before (2,2)

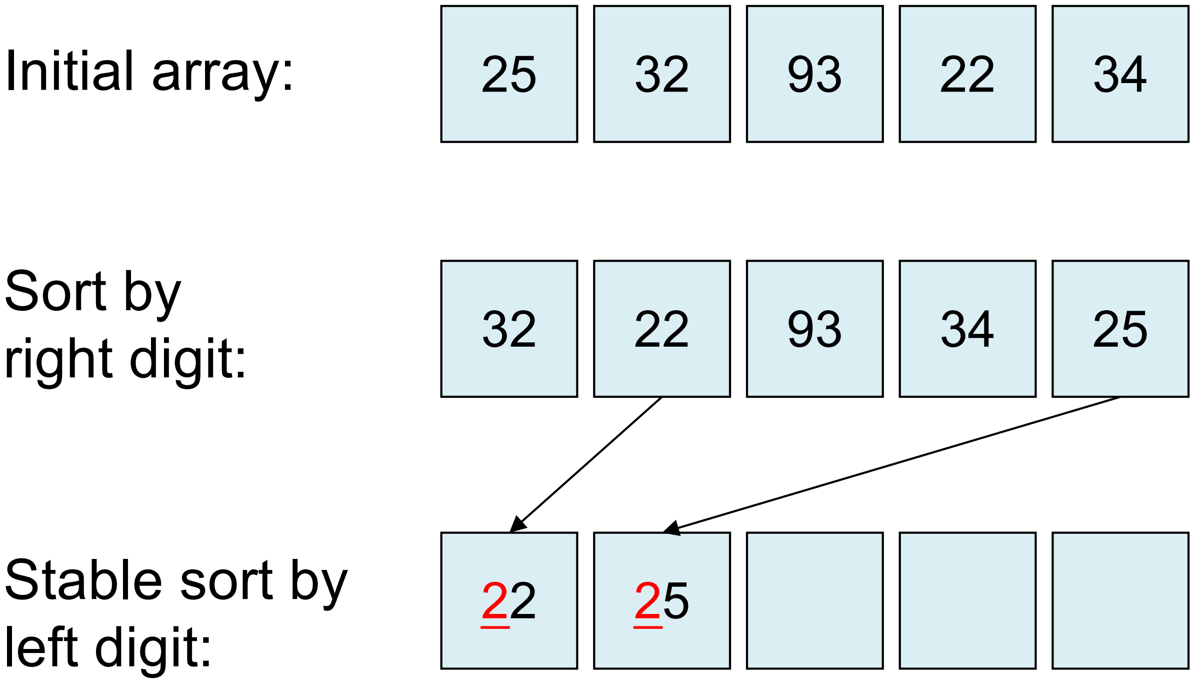
Using Stable Sort (1/4)

- Suppose we sort some 2-digit integers
- **Phase 1: Stable sort by the right digit (the least significant digit)**

Initial array:	25	32	93	22	34
Sort by right digit:	32	22	93	34	25

Using Stable Sort (2/4)

- Suppose we sort some 2-digit integers
- **Phase 2: Stable sort by the left digit (the second least significant digit)**



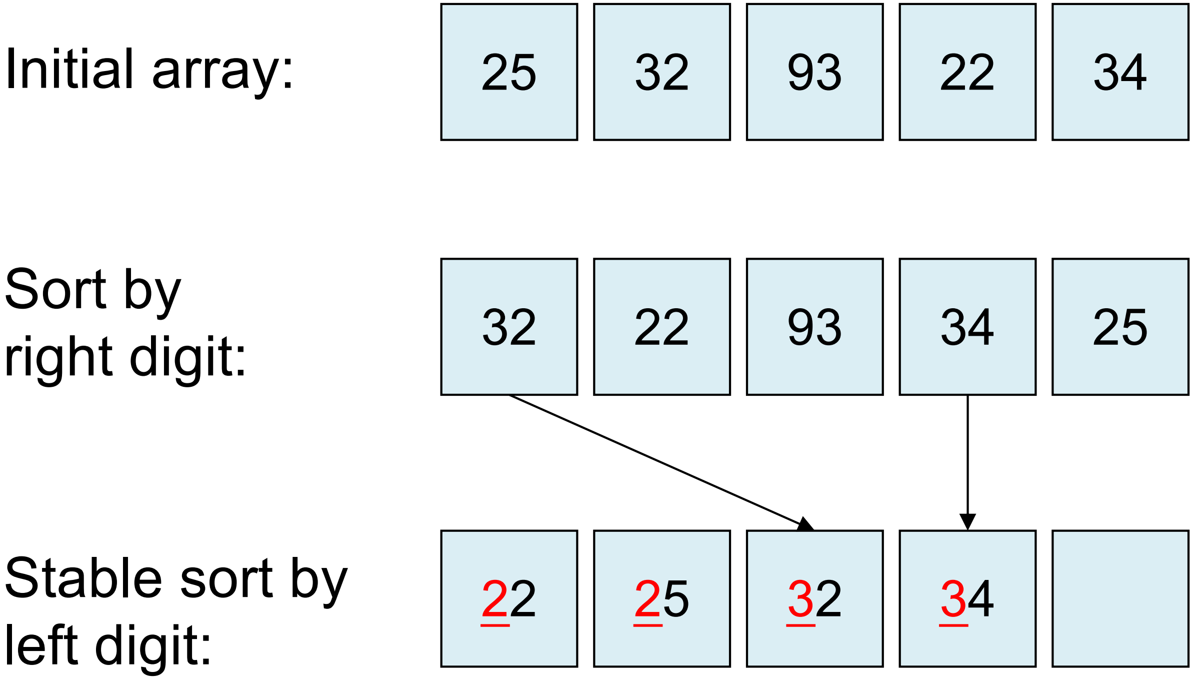
Initial array: 25 32 93 22 34

Sort by right digit: 32 22 93 34 25

Stable sort by left digit: 22 25

Using Stable Sort (3/4)

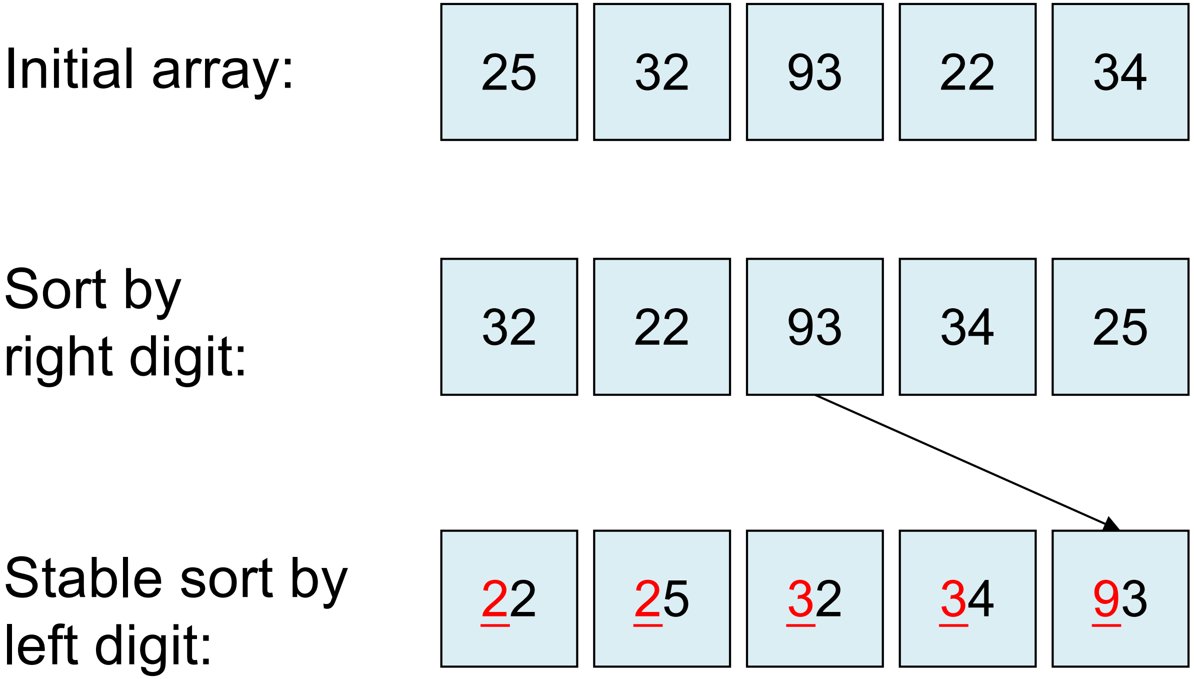
- Suppose we sort some 2-digit integers
- **Phase 2: Stable sort by the left digit (the second least significant digit)**



Initial array:	25	32	93	22	34
Sort by right digit:	32	22	93	34	25
Stable sort by left digit:	22	25	32	34	

Using Stable Sort (4/4)

- Suppose we sort some 2-digit integers
- **Phase 2: Stable sort by the left digit (the second least significant digit)**



Initial array:	25	32	93	22	34
Sort by right digit:	32	22	93	34	25
Stable sort by left digit:	22	25	32	34	93

Radix Sort

- Bucket sort is not efficient if M is large
- The idea of radix sort:
  - Apply stable bucket sort on each digit (from Least Significant Digit to Most Significant Digit)
- A complication:
  - Just keeping the count is not enough
  - Need to keep the actual elements
  - Use a queue for each digit

Radix Sort (Example) (1/3)

- Input: 170, 045, 075, 090, 002, 024, 802, 066
- The **first** pass
  - Consider **the least significant digits** as keys and move the keys into their buckets

0	170, 090
1	
2	002, 802
3	
4	024
5	045, 075
6	066
7	
8	
9	

  - Output: 170, 090, 002, 802, 024, 045, 075, 066

Radix Sort (Example) (2/3)

- The **second** pass
- Input: $1\underline{7}0$, $0\underline{9}0$, $0\underline{0}2$, $8\underline{0}2$, $0\underline{2}4$, $0\underline{4}5$, $0\underline{7}5$, $0\underline{6}6$
  - Consider **the second least significant digits** as keys and move the keys into their buckets

0	$0\underline{0}2$, $8\underline{0}2$
1	
2	$0\underline{2}4$
3	
4	$0\underline{4}5$
5	
6	$0\underline{6}6$
7	$1\underline{7}0$, $0\underline{7}5$
8	
9	$0\underline{9}0$

  - Output: $0\underline{0}2$, $8\underline{0}2$, $0\underline{2}4$, $0\underline{4}5$, $0\underline{6}6$, $1\underline{7}0$, $0\underline{7}5$, $0\underline{9}0$

Radix Sort (Example) (3/3)

- The **third** pass
- Input: 002, 802, 024, 045, 066, 170, 075, 090
  - Consider **the third least significant digits** as keys and move the keys into their buckets

0	002, 024, 045, 066, 075, 090
1	170
2	
3	
4	
5	
6	
7	
8	802
9	

  - Output: 002, 024, 045, 066, 075, 090, 170, 802 (Sorted)

Radix Sort Code (1/2)

```
// item is the type: {0,…,10^d-1},
// i.e., the type of d-digit integers
void radixsort(item A[], int n, int d)
{
   int i;
   for (i=0; i<d; i++)
      bucketsort(A, n, i);
}

// To extract d-th digit of x
int digit(item x, int d)
{
   int i;
   for (i=0; i<d; i++)
      x /= 10; // integer division
   return x%10;
}
```

Radix Sort Code (2/2)

```
void bucketsort(item A[], int n, int d)
// stable-sort according to d-th digit
{
   int i, j;
   Queue *C = new Queue[10];
   for (i=0; i<10; i++) C[i].makeEmpty();
   for (i=0; i<n; i++)
      C[digit(A[i],d)].EnQueue(A[i]);
   for (i=0, j=0; i<10; i++)
      while (!C[i].empty())
      { // copy values from queues to A[]
         C[i].DeQueue(A[j]);
         j++;
      }
}
```

Radix Sort: Worst-case Time Complexity

- Assume ***d*** digits, each digit comes from {0,…,M-1}
- For each digit,
  - O(M) time to initialize M queues,
  - O(n) time to distribute n numbers into M queues
- Total time = O(d(M+n))
- When *d* is constant and M = O(n), we can make radix sort run in linear time, i.e., O(n).

Learning Objectives

1. Know insertion sort, Radix sort, Bucket sort, Quicksort
2. Able to do heap sort manually
3. Catch the idea behind “merge-sort” and able to use the idea to solve some other related problems
4. Able to implement Various sorting algorithms

D:1; C:1,2; B:1,2,3; A:1,2,3,4

Exercise 1

Suppose we are sorting an array of eight integers using heapsort, and we have just finished some deletions (either deleteMax or deleteMin) operations. The array now looks like this: 16 14 15 10 12 27 28.

How many deletions have been performed on root of heap?

a. 1
b. 2
c. 3 or 4
d. 5 or 6

Exercise 2

Which sorting algorithms is most efficient to sort string consisting of ASCII characters?

a. Quick sort
b. Heap sort
c. Merge sort
d. Bucket Sort

Exercise 3

Assume that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64.

Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

a. 256

b. 512

c. 1024

d. 2048

Exercise 4

Applying a QuickSort Program to sort numbers in ascending order using the first element as pivot.

Let t1 and t2 be the number of comparisons made by the Quicksort program for the inputs {1, 2, 3, 4, 5} and {4, 1, 5, 3, 2} respectively. Which one of the following is true?

a. t1=5
b. t1<t2
c. t1>t2
d. t1=t2