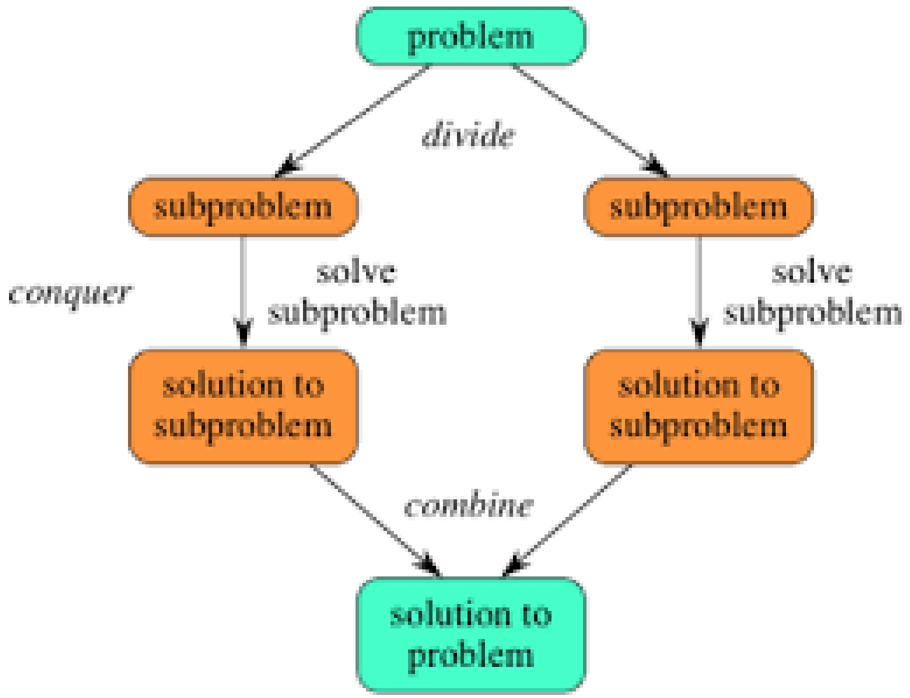
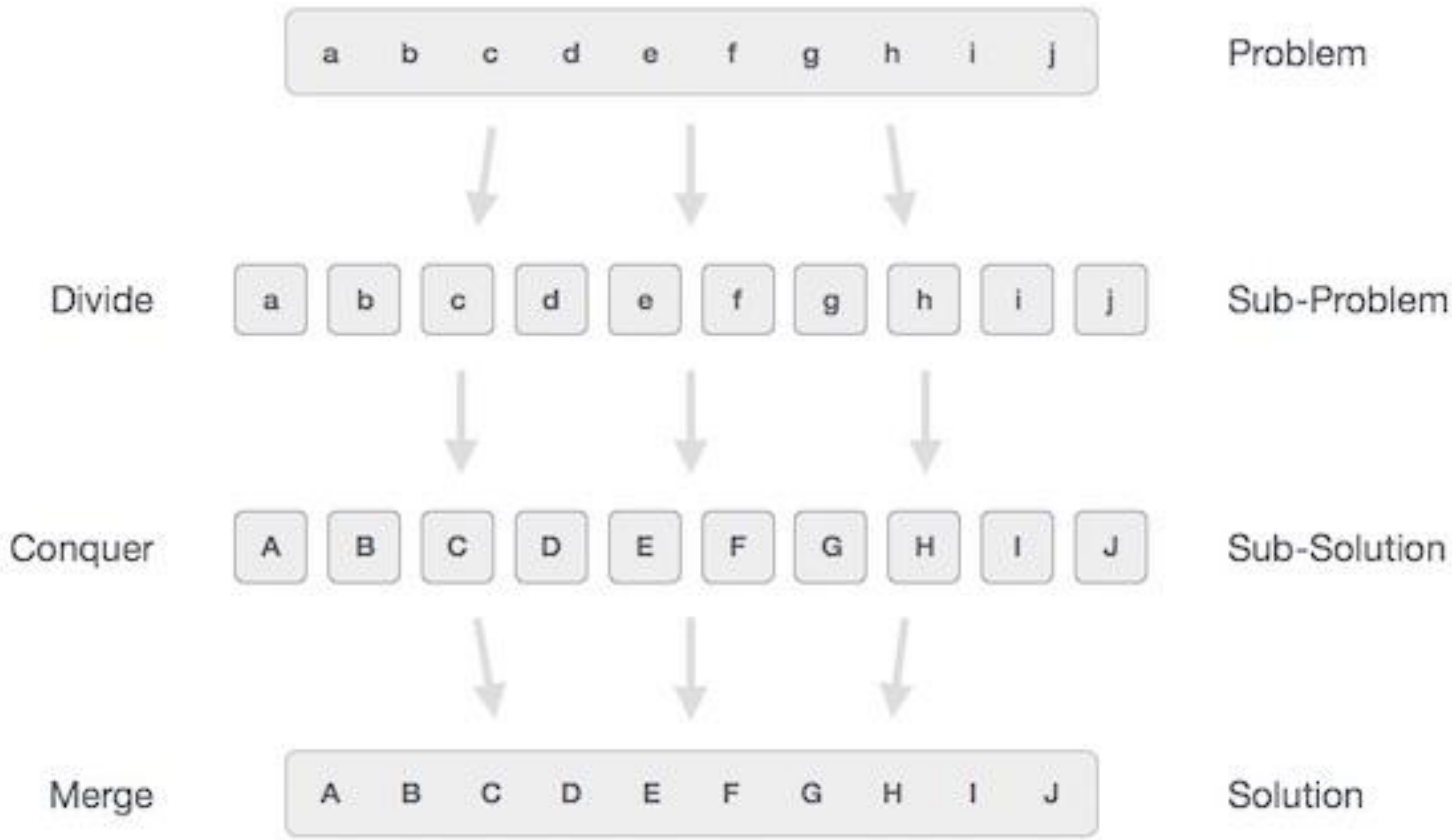
Unit 2

Design Techniques I

Divide and Conquer



Problem divides like



Examples

- The following computer algorithms are based on **divide-and-conquer** programming approach –
  - Merge Sort
  - Quick Sort
  - Binary Search
  - Strassen's Matrix Multiplication
  - Closest pair (points)

Greedy Algorithm

- An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.
- **Counting Coins**
  - This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −
    - Select one ₹ 10 coin, the remaining count is 8
    - Then select one ₹ 5 coin, the remaining count is 3
    - Then select one ₹ 2 coin, the remaining count is 1
    - And finally, the selection of one ₹ 1 coins solves the problem

Merge sort

- Apply divide-and-conquer to sorting problem
- Problem: Given *n* elements, sort elements into non-decreasing order
- Divide-and-Conquer:
  - If n=1 terminate (every one-element list is already sorted)
  - If n>1, partition elements into two or more sub-collections; sort each; combine into a single sorted list
- How do we partition?

Partitioning - Choice 2

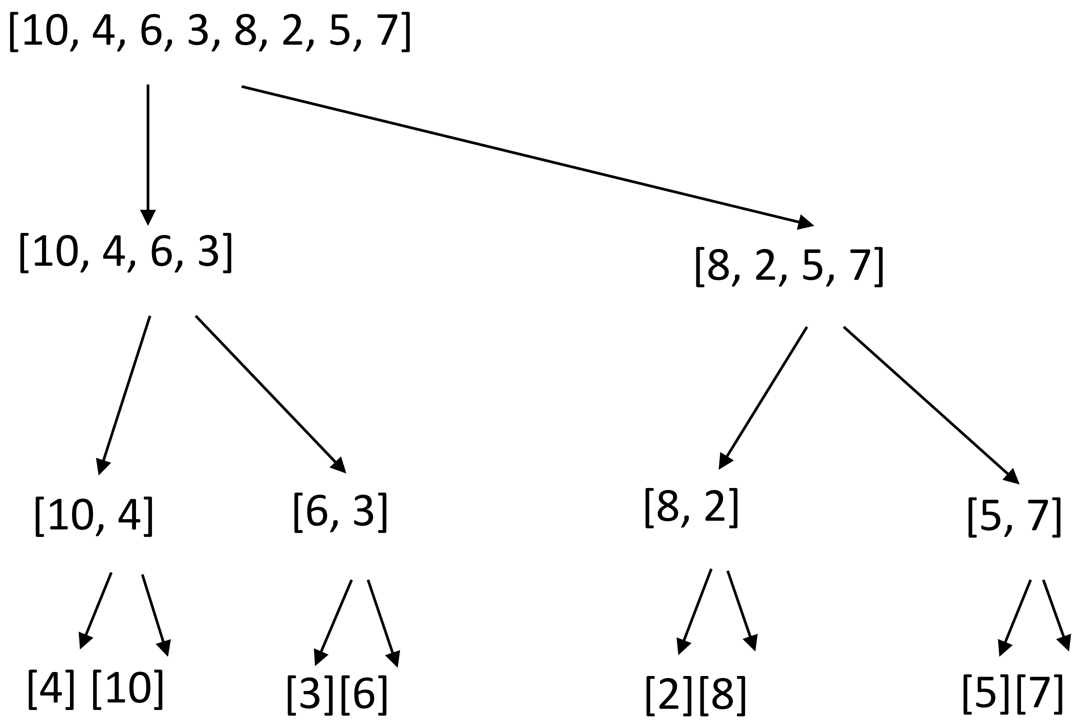
- Put element with largest key in B, remaining elements in A
- Sort A recursively
- To combine sorted A and B, append B to sorted A
  - Use Max() to find largest element $\rightarrow$ recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position $\rightarrow$ recursive BubbleSort()
- All $O(n^2)$

Partitioning - Choice 3

- Let's try to achieve balanced partitioning
- A gets *n/2* elements, B gets rest half
- Sort A and B recursively
- Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
  - How? We will see soon

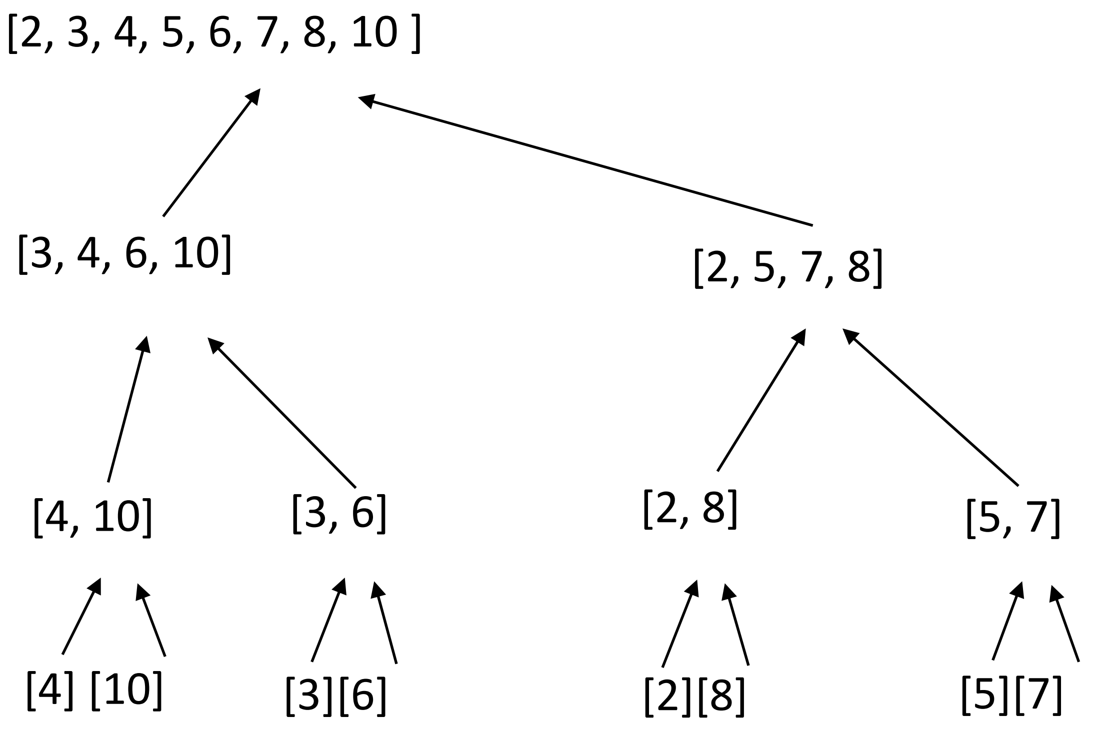
Example

- Partition into lists of size n/2



Example Cont'd

- Merge



```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Merge Function

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
```

```
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

A={85,76,46,92,30,41,42,12,19,93,3,50,11}

Pass 1: 85,76, 46,92,30,41| 42,12,19,93,3,50,11

Pass 2: 85,76,46|92,30,41| 42,12,19|93,3,50,11

Pass 3: 85|76,46 | 92|30,41|42|12,19|93,3|50,11

Pass 4:
85|76|46|92|30|41|42|12|19|93|3|50|11

Sort+merge:
76,85|46,92|30,41|12,42|19,93|3,50|11

46,76,85, 92|30,41|12, 19,42,93|3,11,50

30,41,46,76,85,92|3,11,12,19,42,50,93

3,11,12,19,30,41,42,46,50,76,85,92,93

Evaluation

- Recurrence equation:
- Assume n is a power of 2

$$T(n) = \begin{cases} c_1 & \text{if } n=1 \\ 2T(n/2) + c_2 n & \text{if } n>1,\ n=2^k \end{cases}$$

Quick sort

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

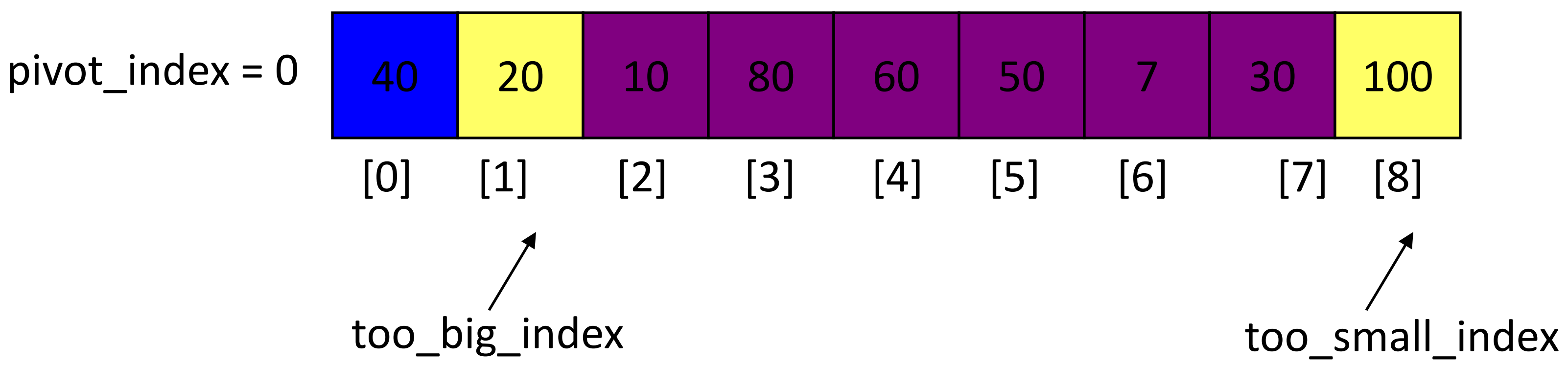
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

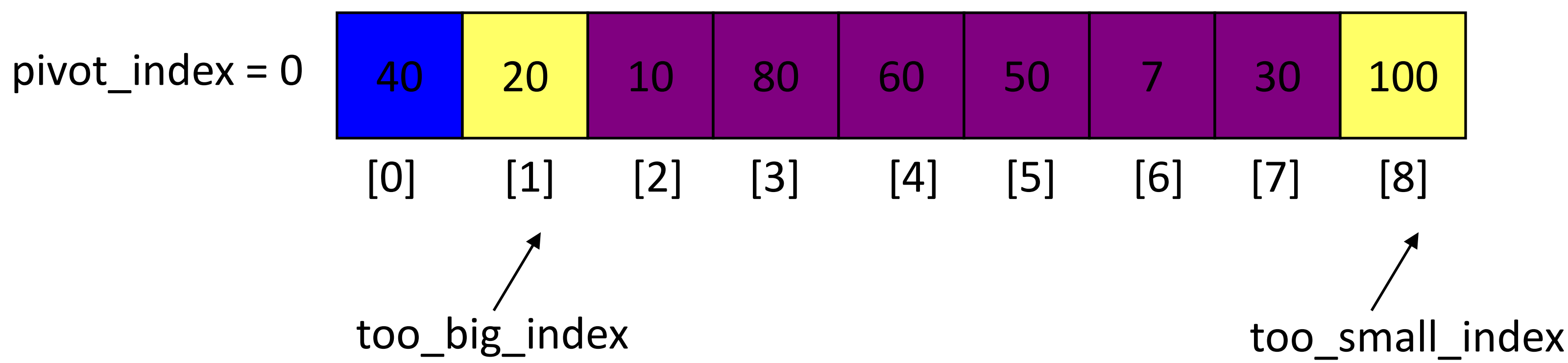
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

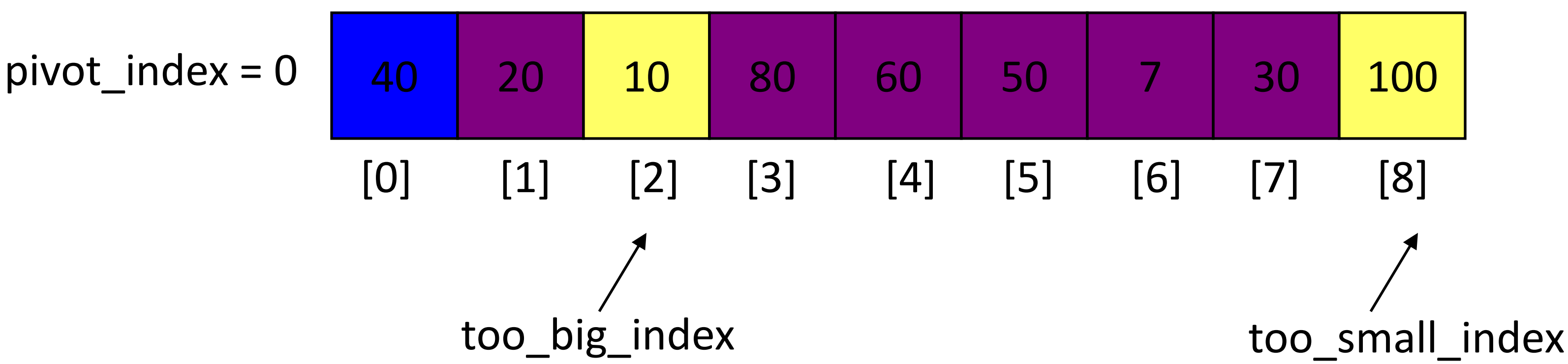
Partitioning loops through, swapping elements below/above pivot.


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

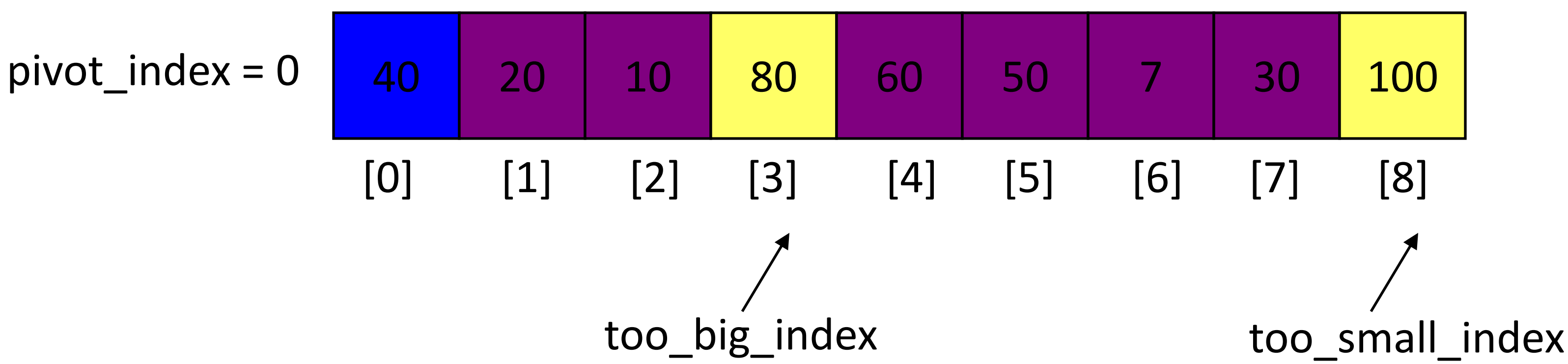
```
1.    While data[too_big_index] <= data[pivot]
          ++too_big_index
```



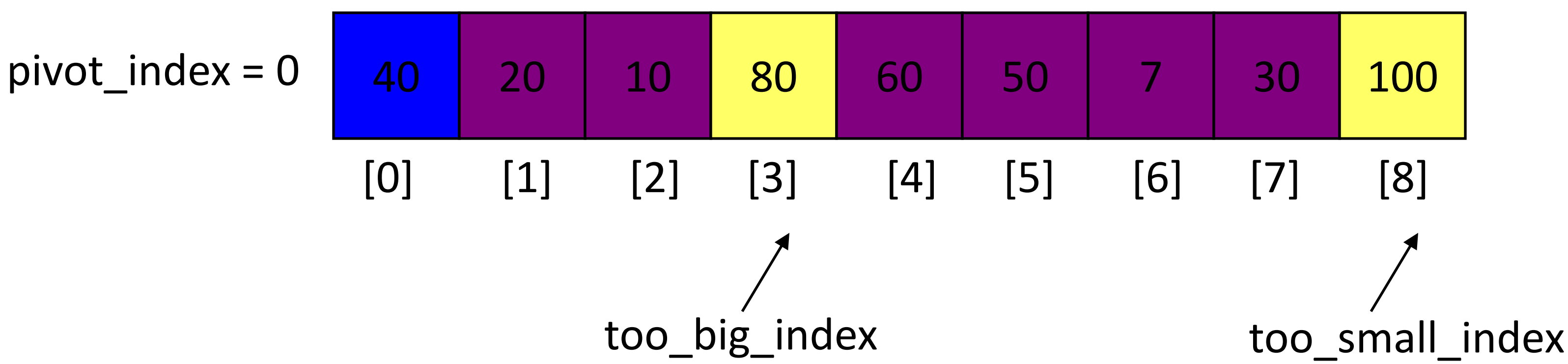
1. While data[too_big_index] <= data[pivot]
++too_big_index



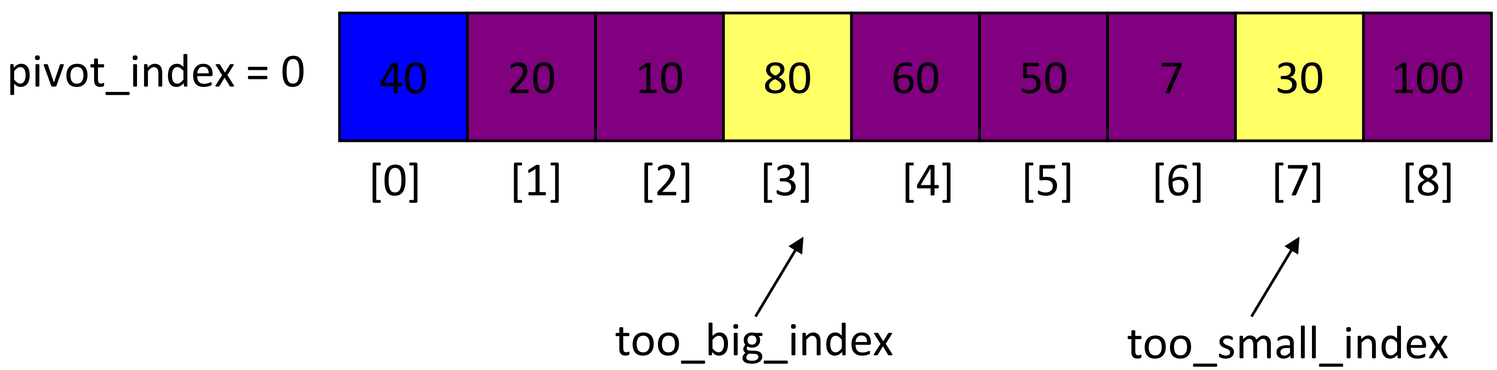
1. While data[too_big_index] <= data[pivot]
    ++too_big_index



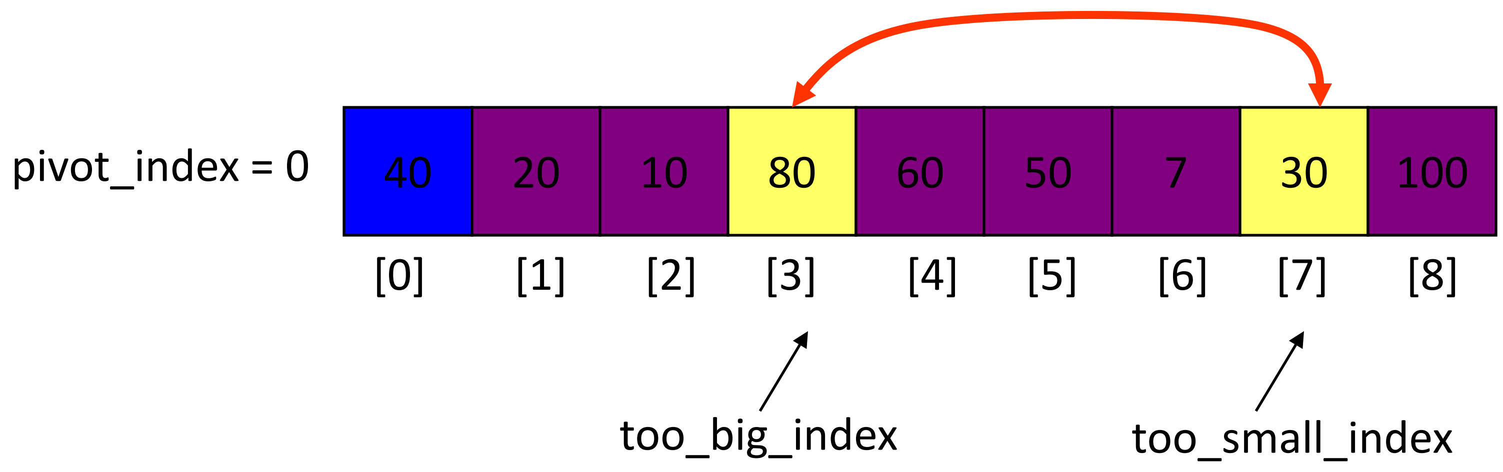
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



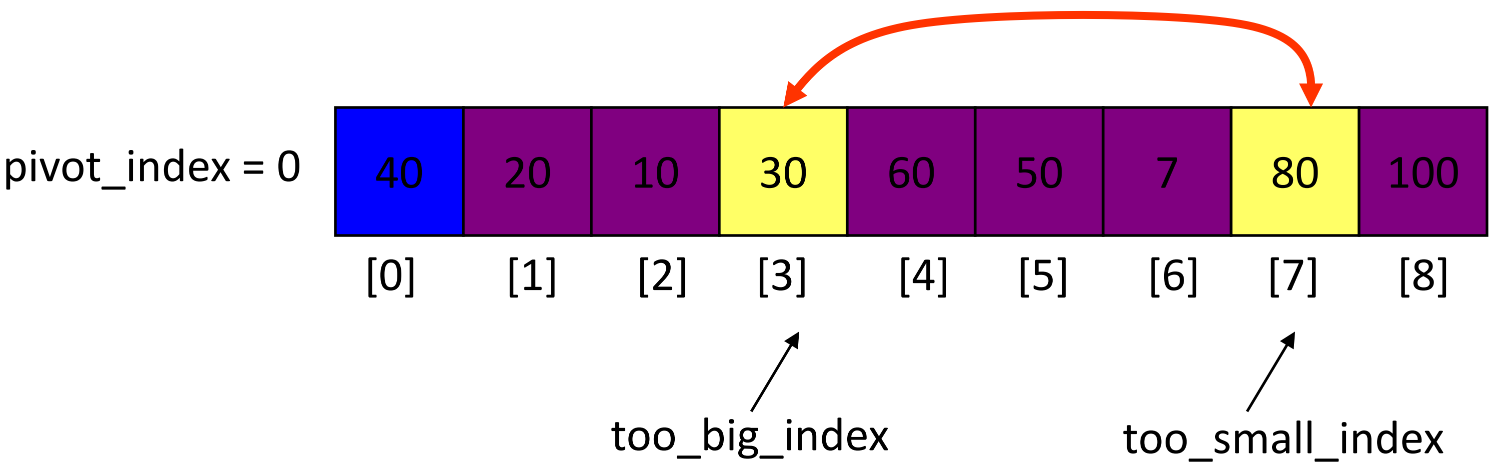
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



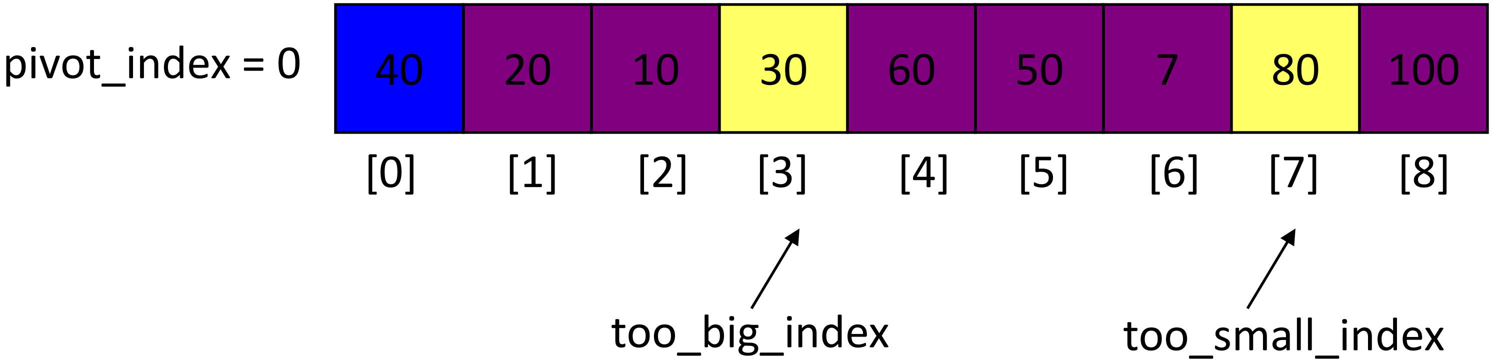
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



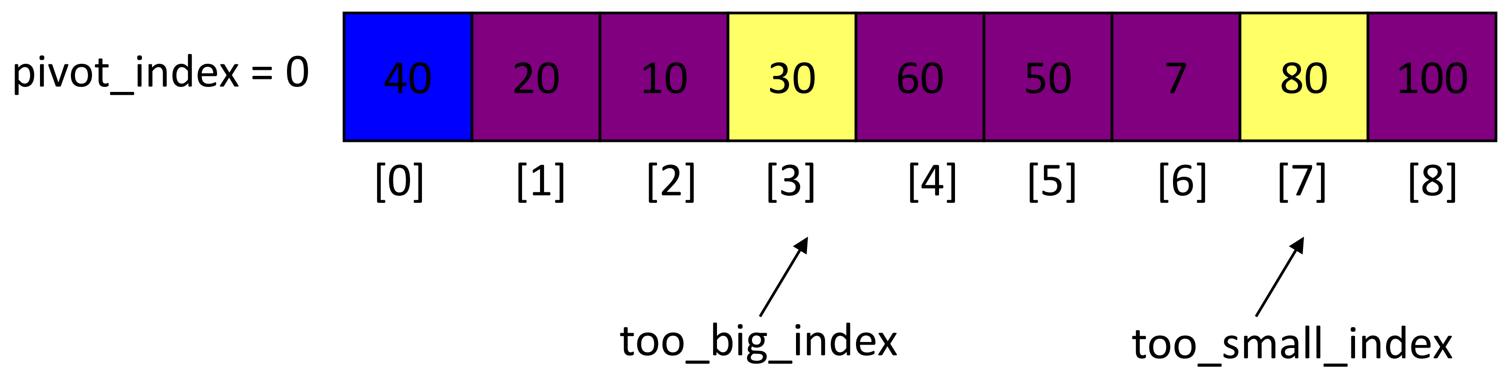
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



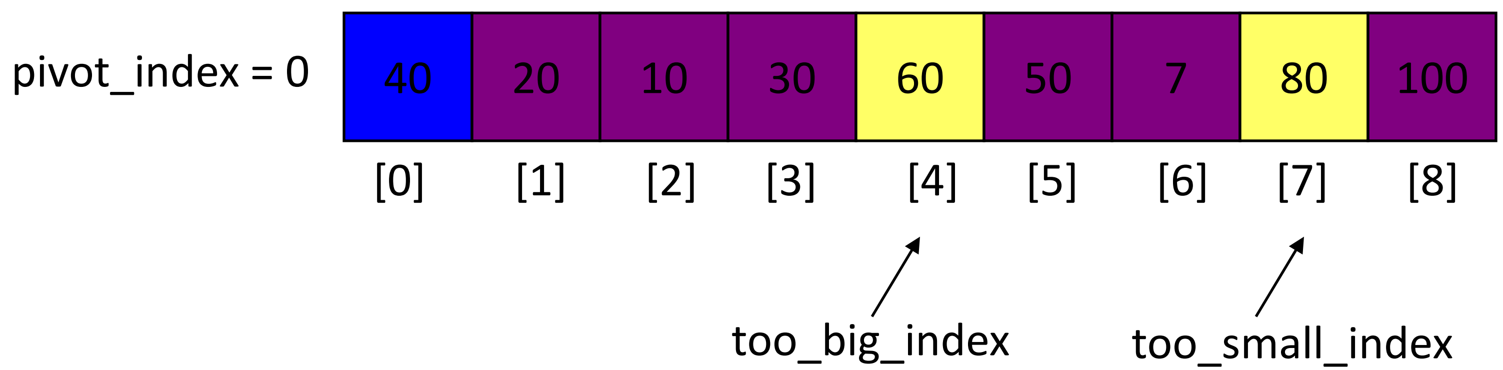
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



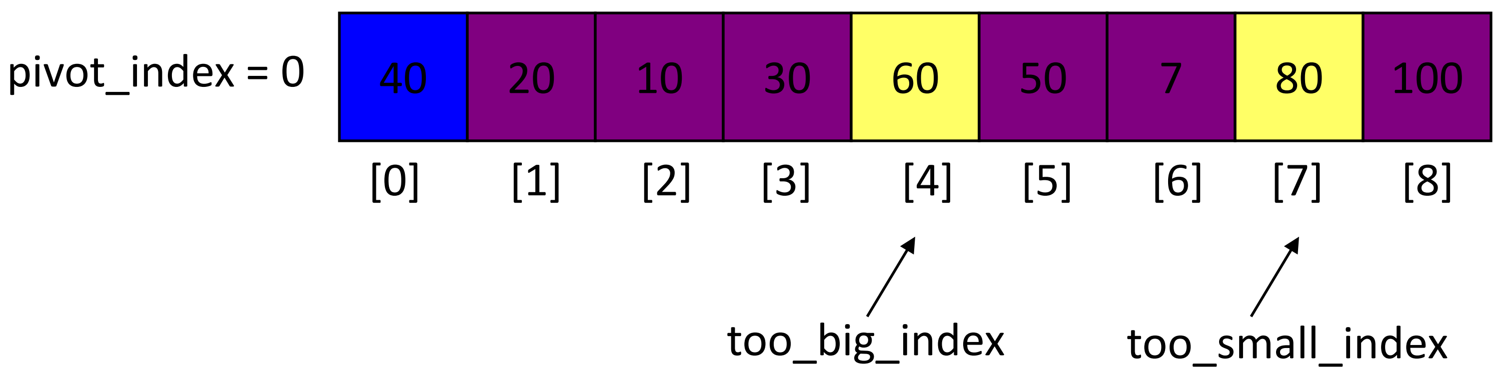
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



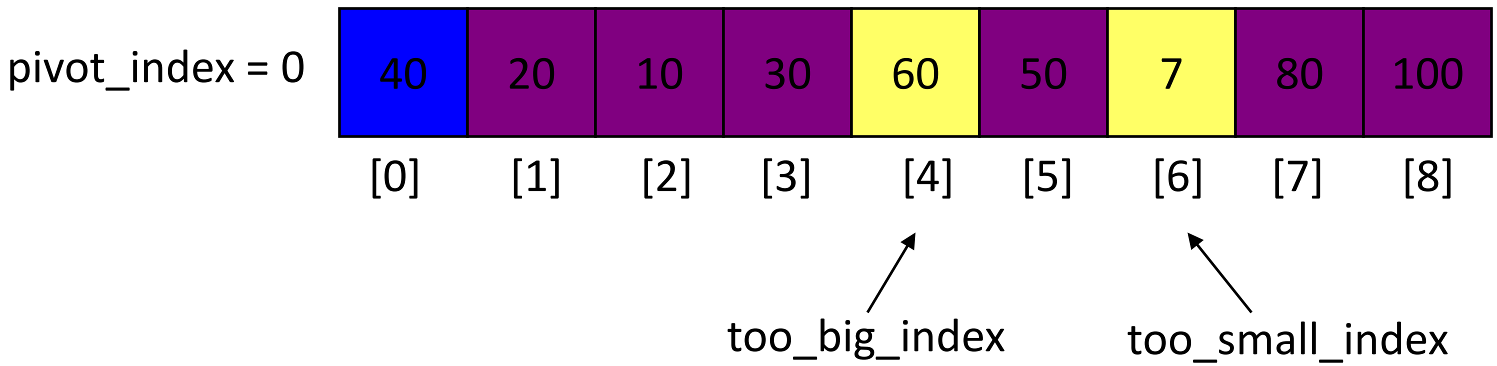
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



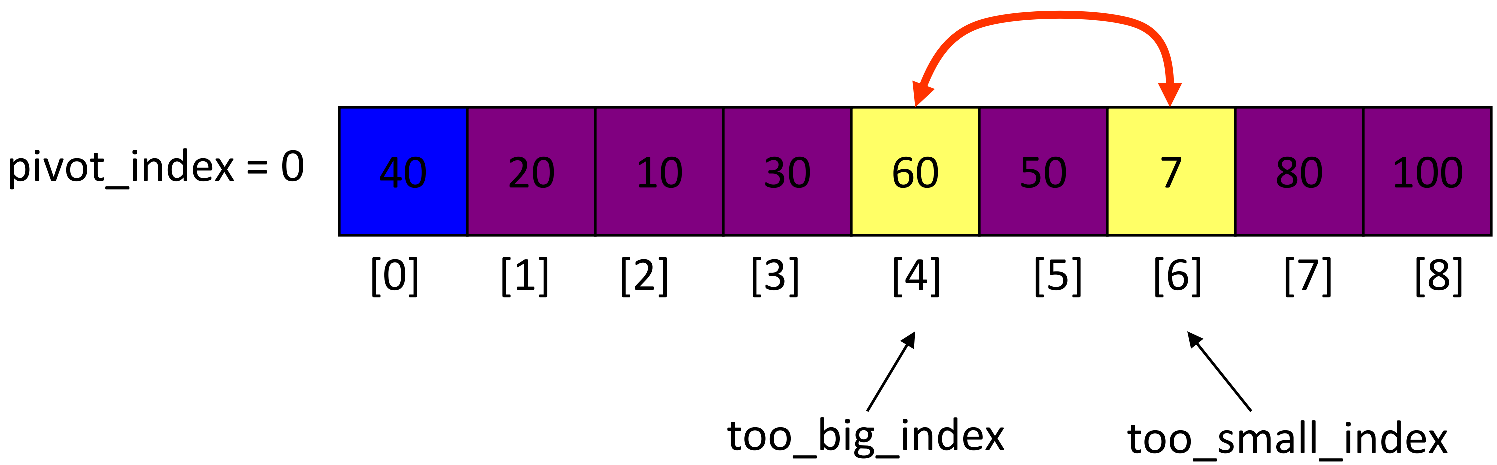
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



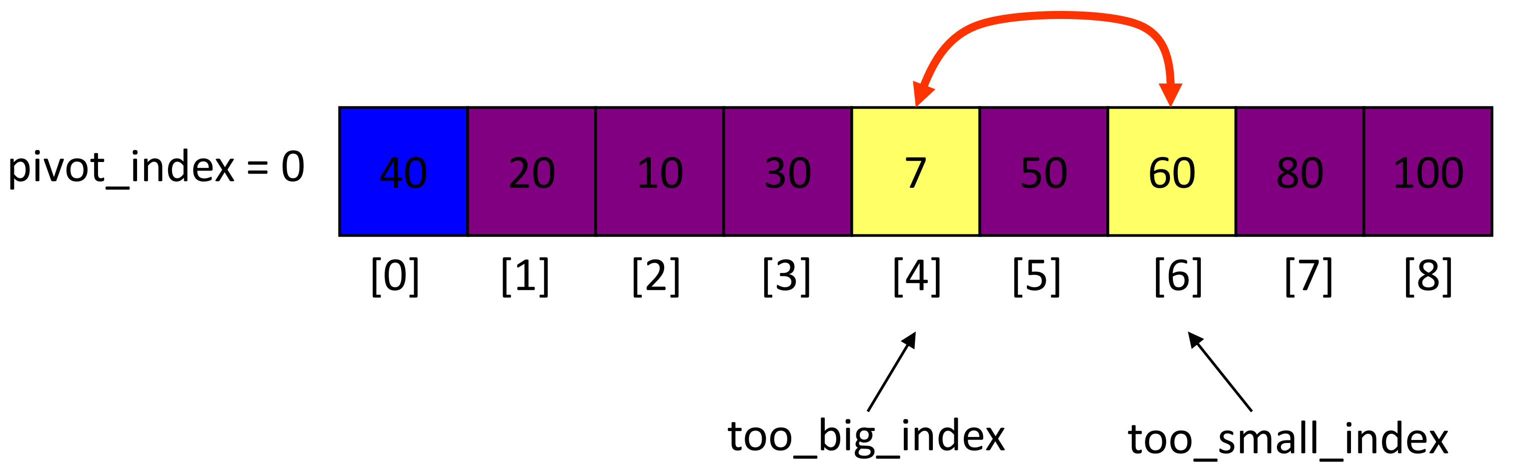
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

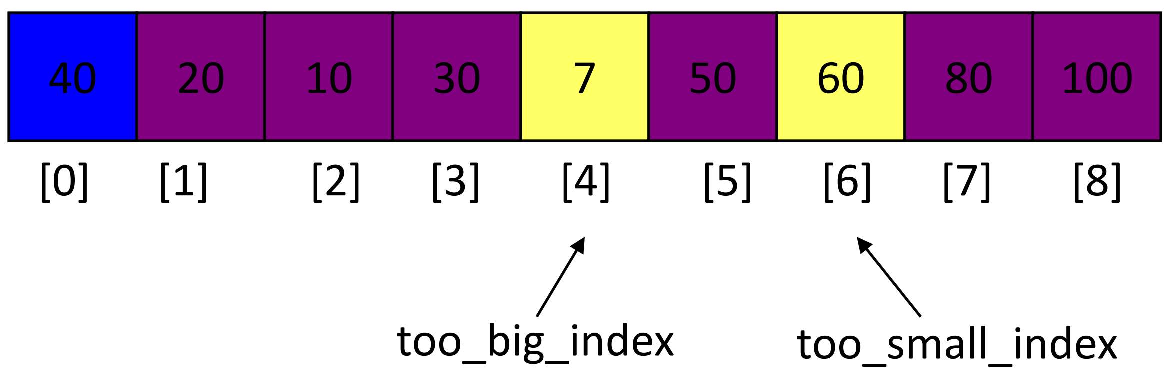


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

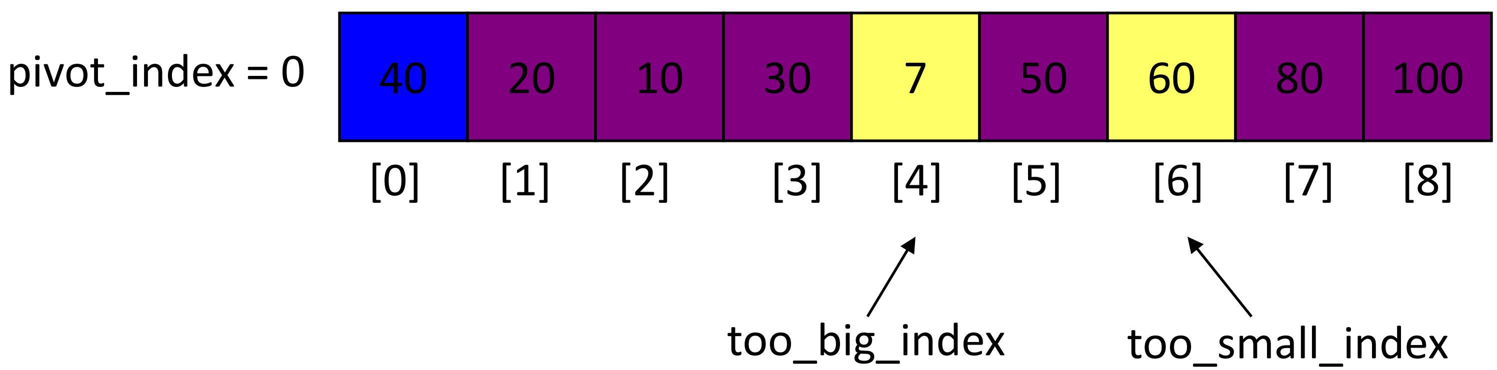


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

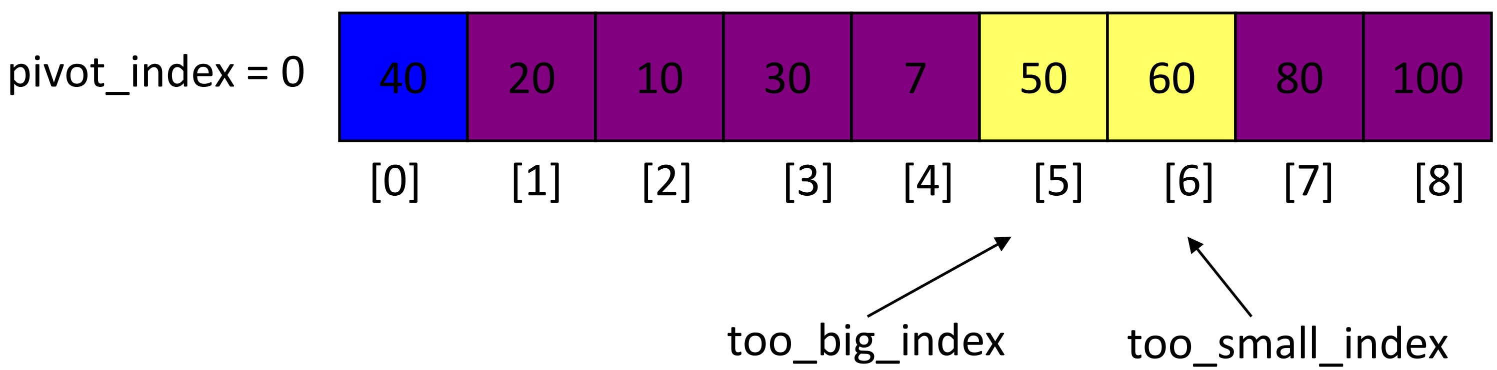


1. While data[too_big_index] <= data[pivot]

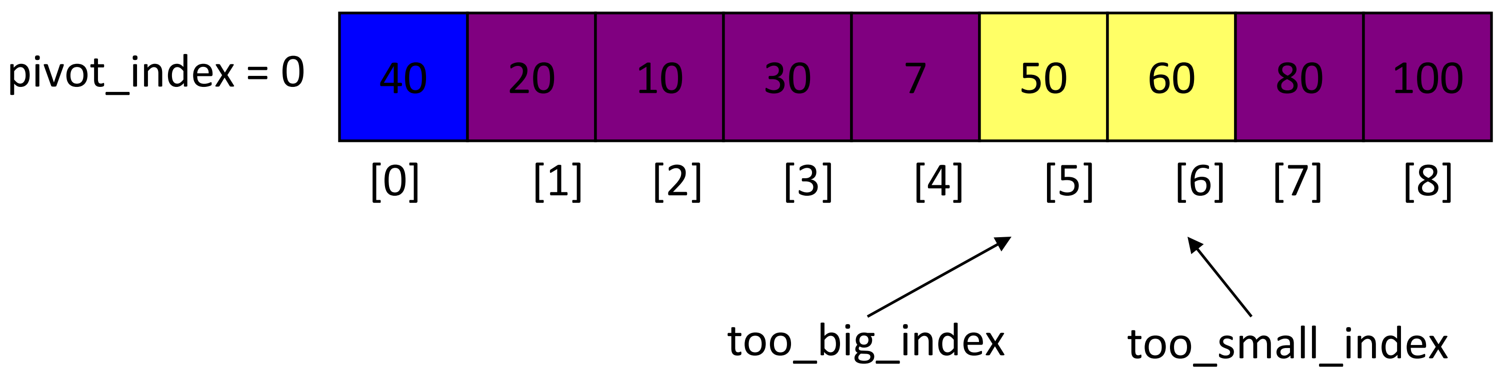
   ++too_big_index
2. While data[too_small_index] > data[pivot]

   --too_small_index
3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]

   ++too_big_index

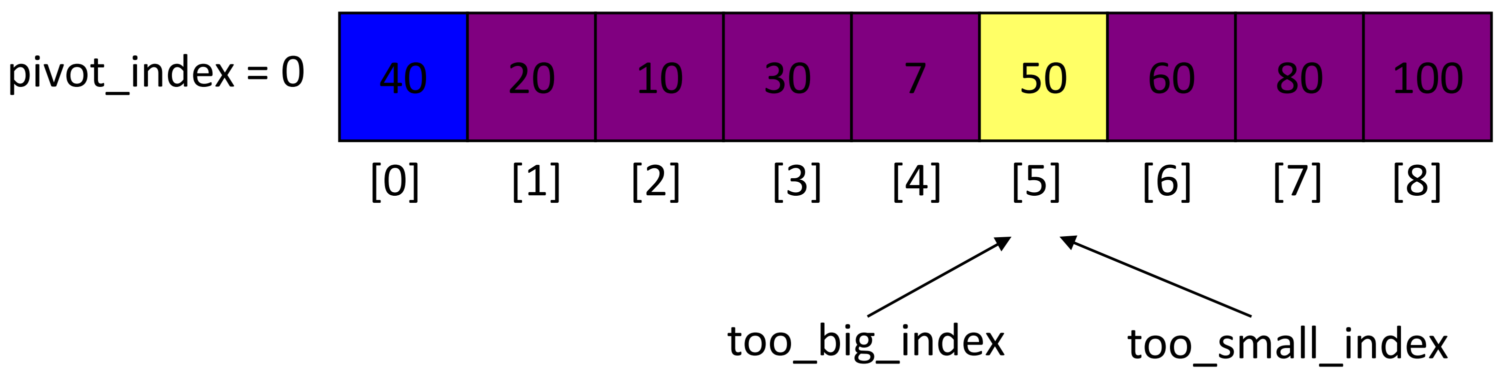
2. While data[too_small_index] > data[pivot]

   --too_small_index

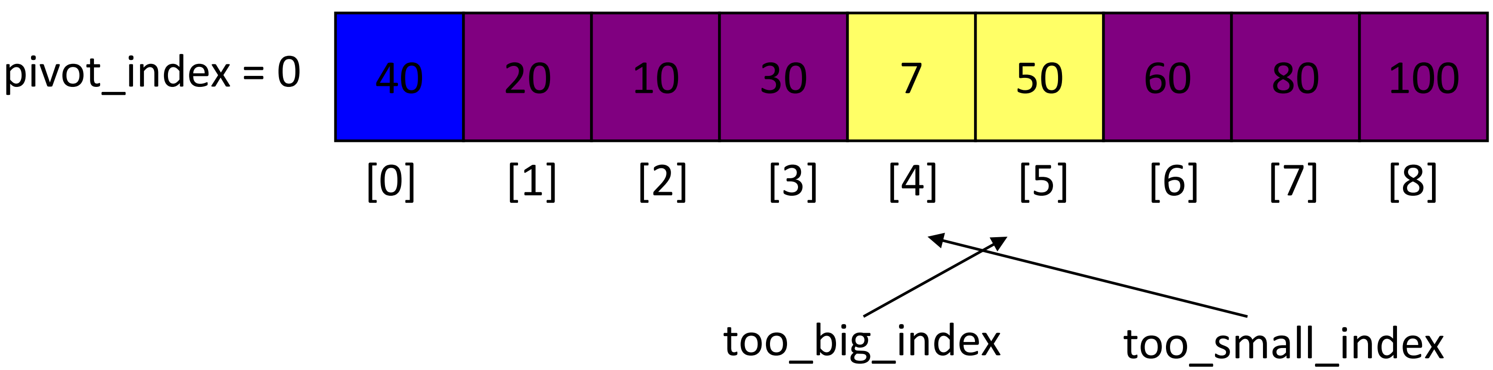
3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

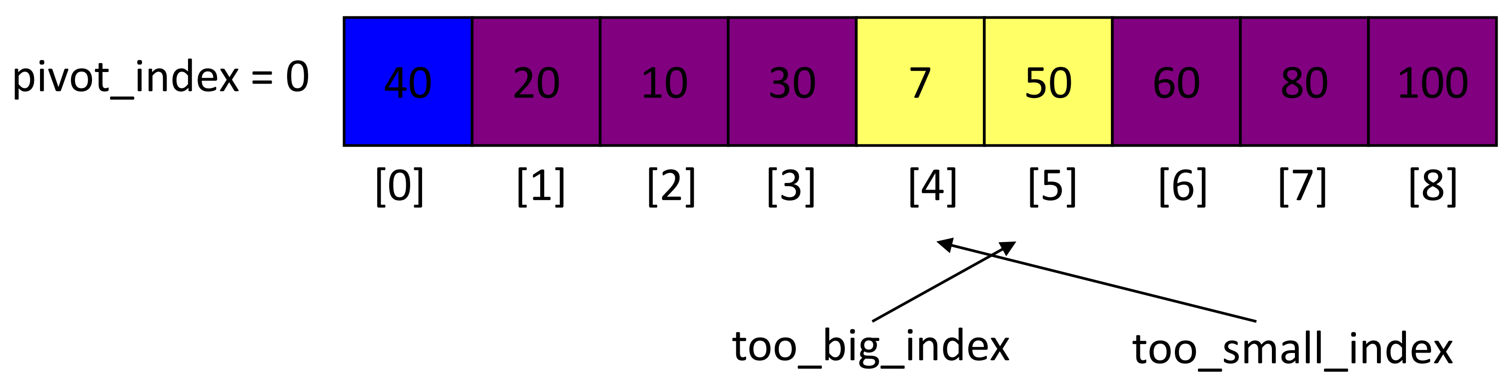
4. While too_small_index > too_big_index, go to 1.



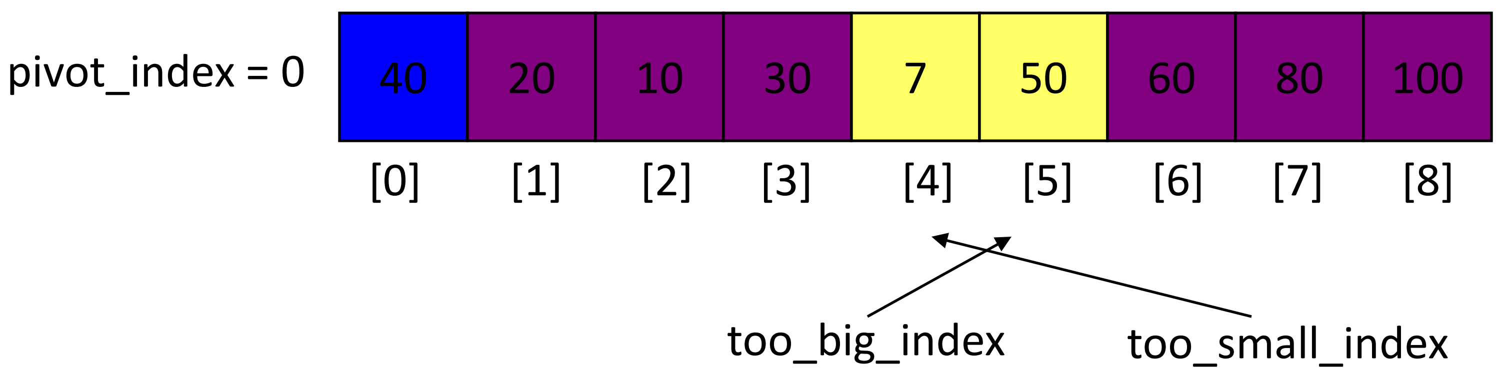
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



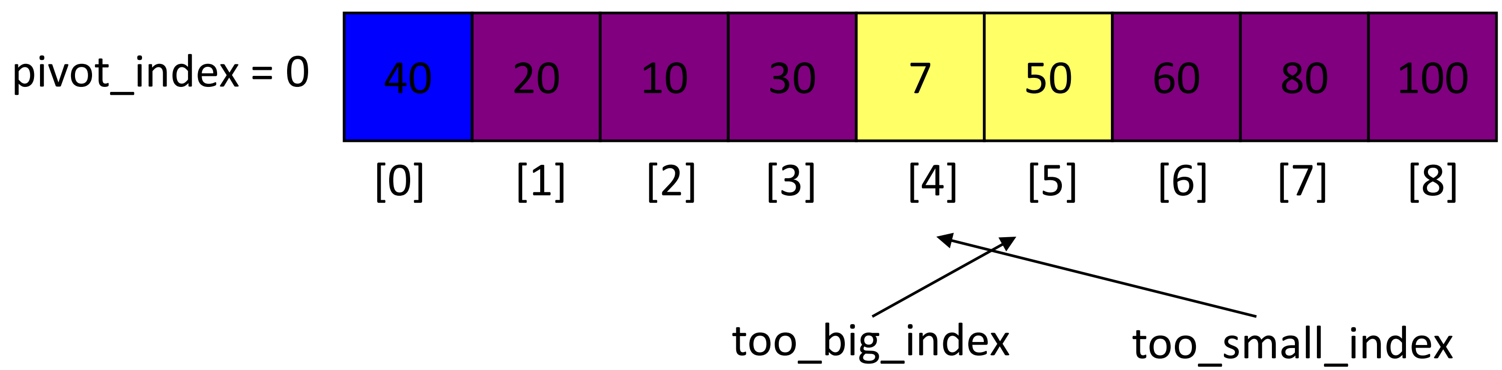
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



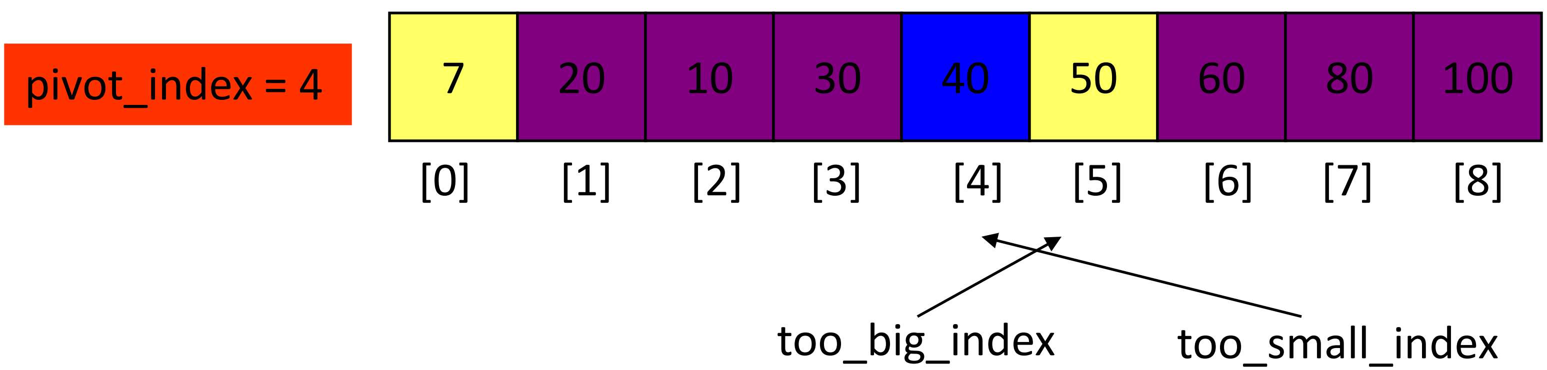
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



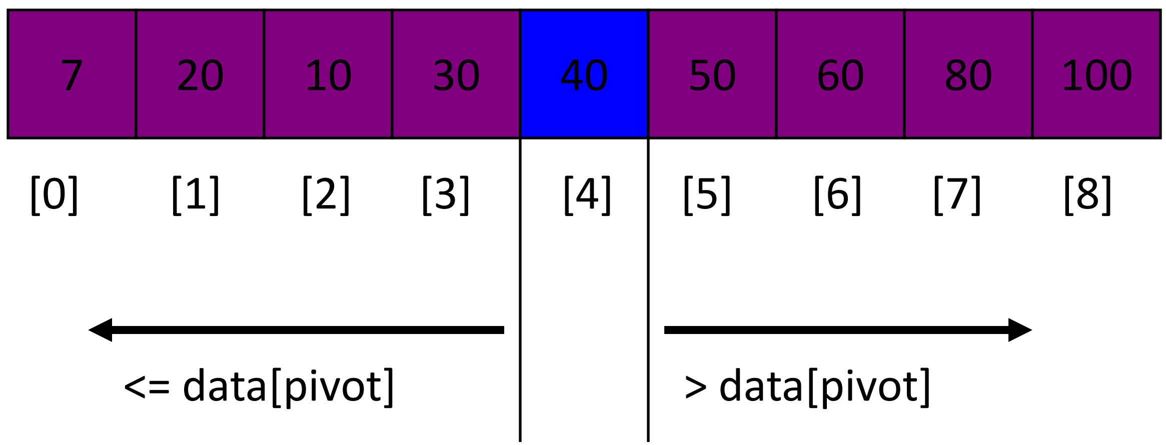
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



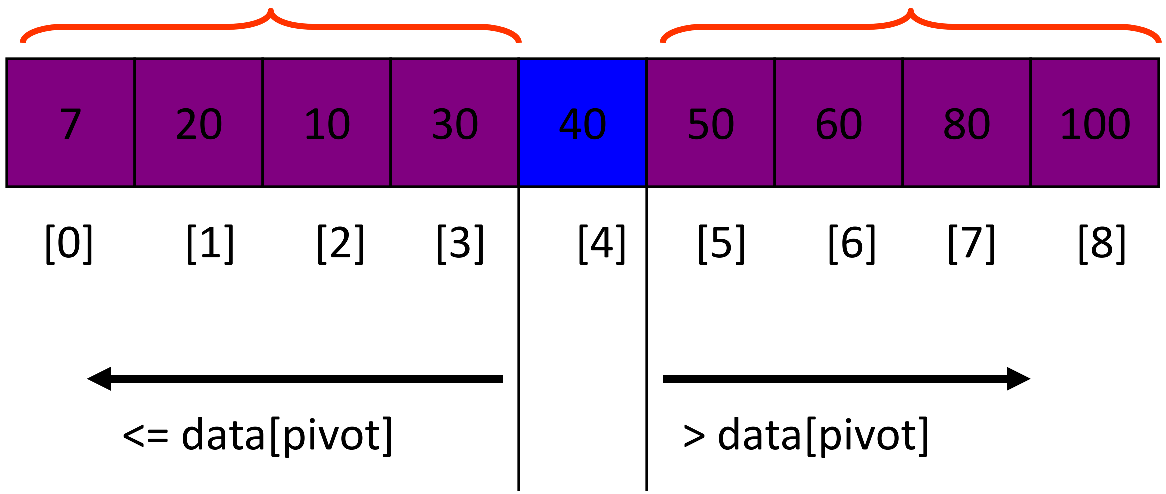
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



Partition Result



Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

Quicksort Analysis

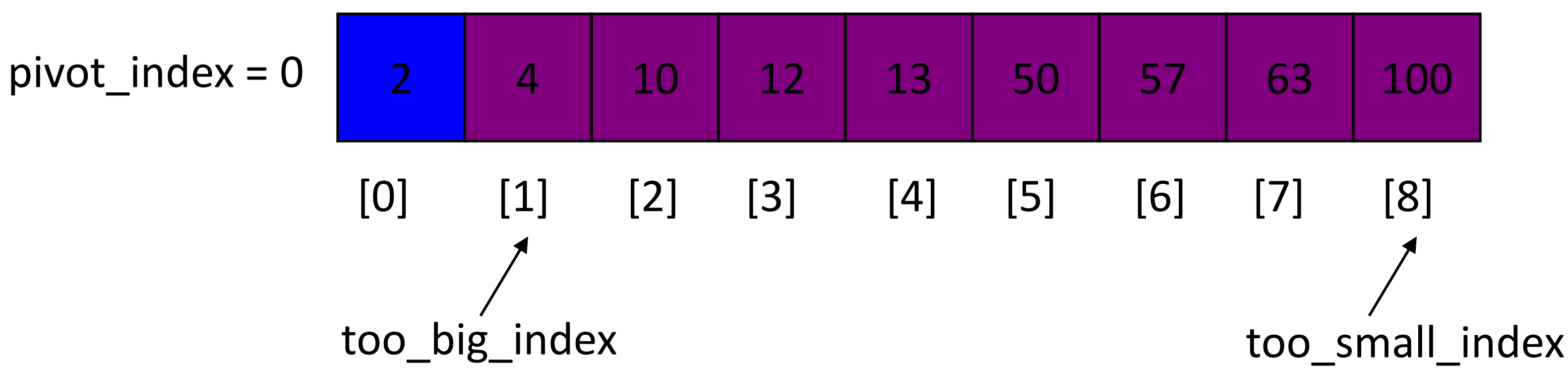
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quick sort

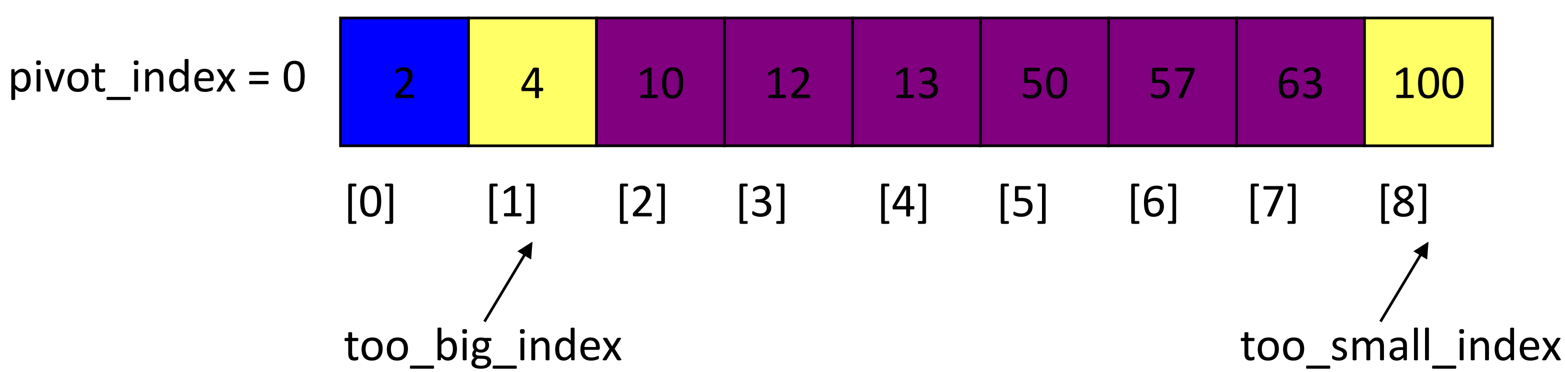
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Quicksort: Worst Case

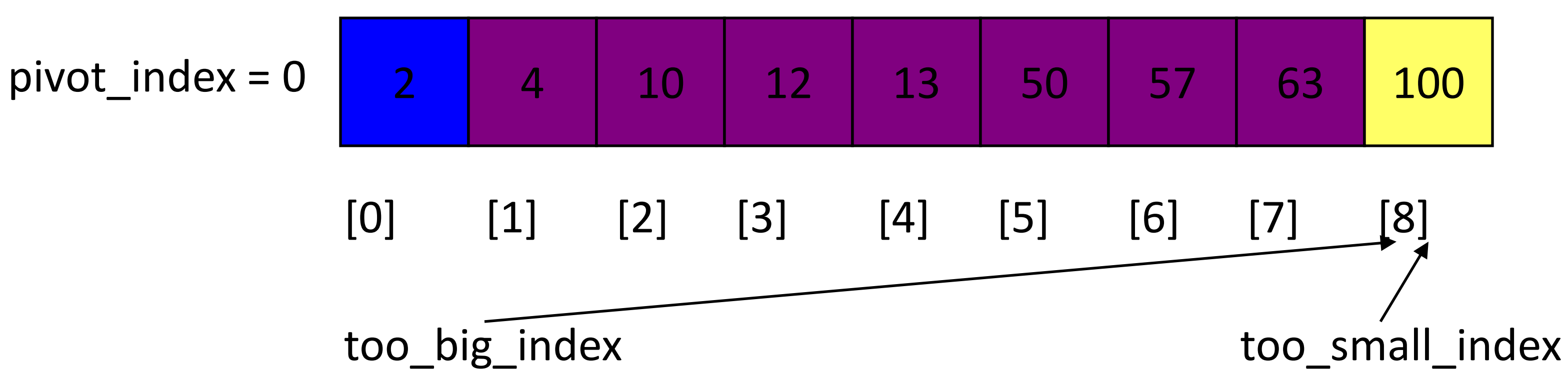
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

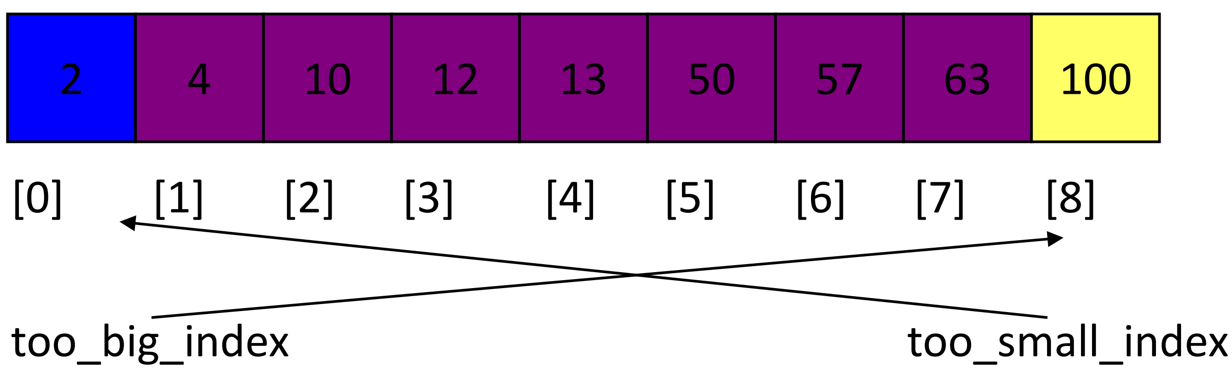


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

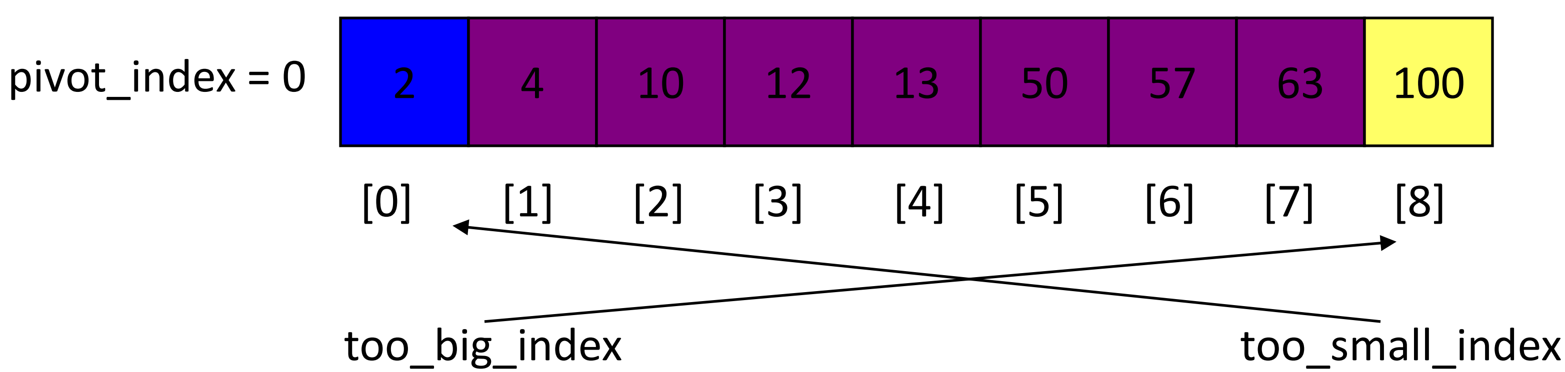


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

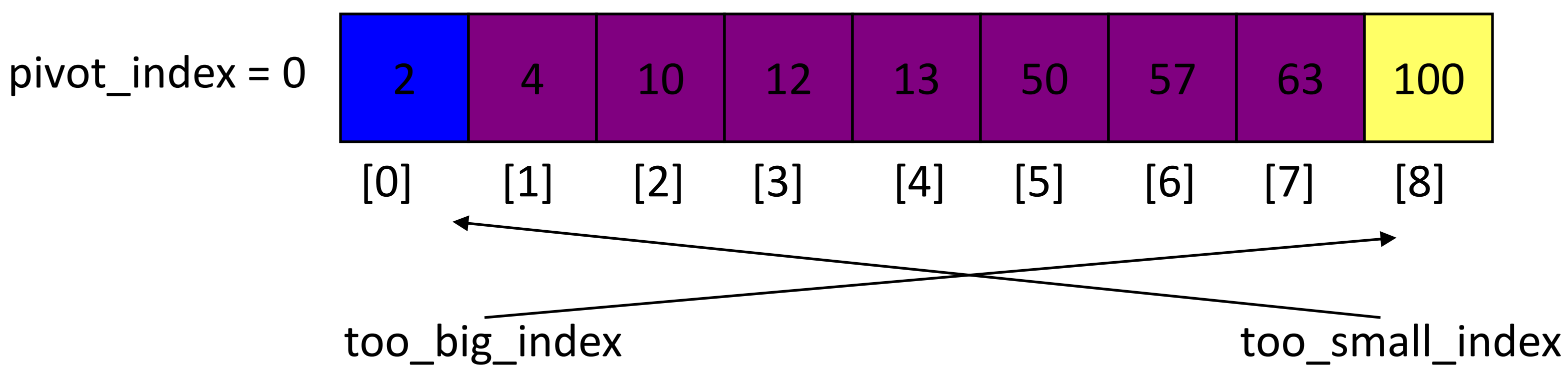
pivot_index = 0



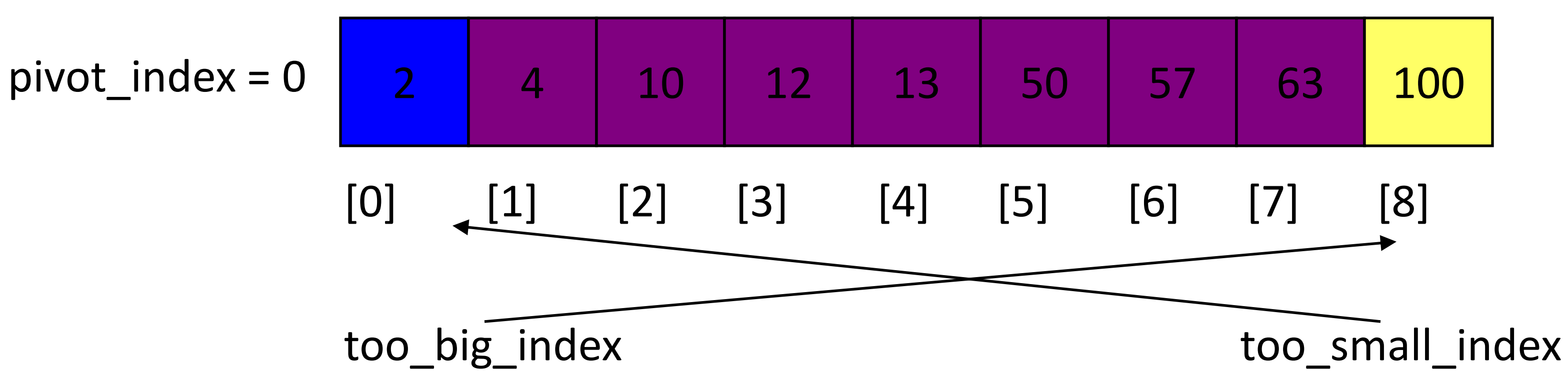
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



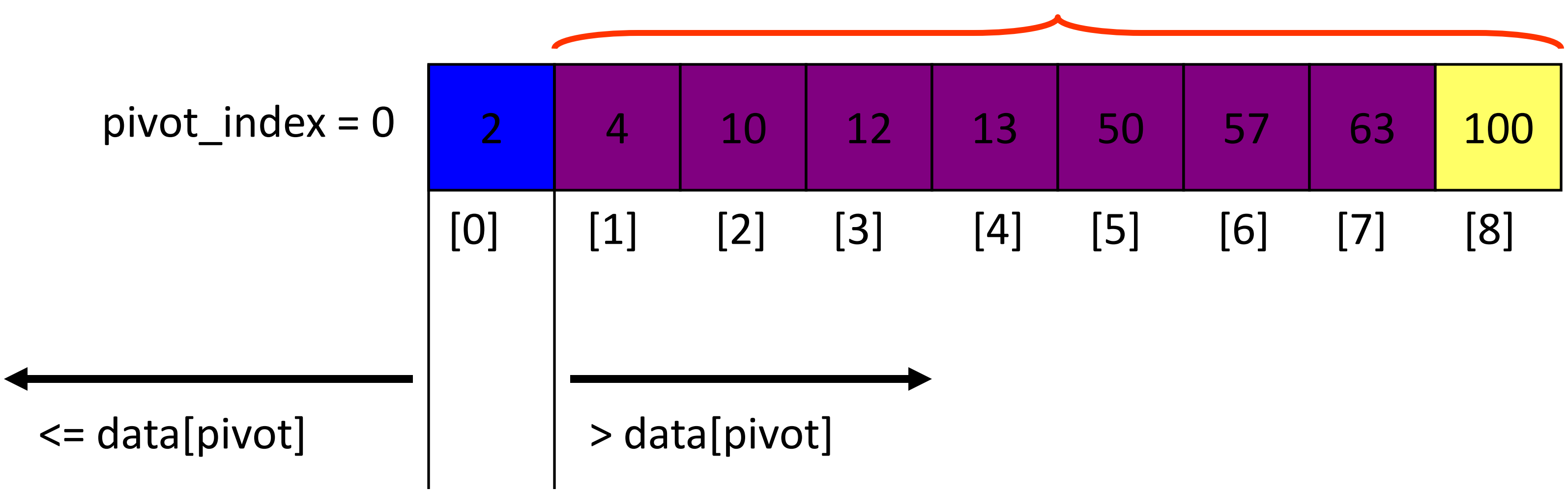
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], . . . , a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j - 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

data[0], data[n/2], and data[n-1].

Use this median value as pivot.

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
  - Sub-array of size 3: left as an exercise.

Iterative version of QuickSort

```
Algorithm QuickSort2(p, q)
// Sorts the elements in a[p : q].
{
    // stack is a stack of size 2 log(n).
    repeat
    {
        while (p < q) do
        {
            j := Partition(a, p, q + 1);
            if ((j - p) < (q - j)) then
            {
                Add(j + 1); // Add j + 1 to stack.
                Add(q); q := j - 1; // Add q to stack
            }
            else
            {
                Add(p); // Add p to stack.
                Add(j - 1); p := j + 1; // Add j - 1 to stack
            }
        } // Sort the smaller subfile.
        if stack is empty then return;
        Delete(q); Delete(p); // Delete q and p from stack.
    } until (false);
}
```

Matrix Multiplication

- Consider two n by n matrices A and B
- Definition of AxB is n by n matrix C whose (i,j)-th entry is computed like this:
  - consider row i of A and column j of B
  - multiply together the first entries of the row and column, the second entries, etc.
  - then add up all the products
- Number of scalar operations (multiplies and adds) in straightforward algorithm is $O(n^3)$.
- Can we do it faster?

Divide-and-Conquer

$$A \times B = C$$

$$\begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} = \begin{bmatrix} A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \end{bmatrix}$$

- Divide matrices A and B into four submatrices each
- We have 8 smaller matrix multiplications and 4 additions. Is it faster?

Divide-and-Conquer

Let us investigate this recursive version of the matrix multiplication.

Since we divide *A*, *B* and *C* into 4 submatrices each, we can compute the resulting matrix C by

- 8 matrix multiplications on the submatrices of *A* and *B*,
- plus $\Theta(n^2)$ scalar operations

Divide-and-Conquer

- Running time of recursive version of straightfoward algorithm is
  - $T(n) = 8T(n/2) + \Theta(n^2)$
  - $T(2) = \Theta(1)$

  where $T(n)$ is running time on an n x n matrix
- Master theorem gives us:

  $$T(n) = \Theta(n^3)$$
- Can we do fewer recursive calls (fewer multiplications of the $n/2$ x $n/2$ submatrices)?

Strassen's Matrix Multiplication

$$A \quad \times \quad B \quad = \quad C$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22}) * B_{11}$
$P_3 = A_{11} * (B_{12} - B_{22})$
$P_4 = A_{22} * (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) * B_{22}$
$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

Strassen's Matrix Multiplication

- Strassen found a way to get all the required information with only 7 matrix multiplications, instead of 8.

- Recurrence for new algorithm is
  - $T(\mathrm{n}) = 7T(n/2) + \Theta(n^2)$

Solving the Recurrence Relation

Applying the Master Theorem to

$T(n) = a\,T(n/b) + f(n)$

with a=7, b=2, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log_b(a)-\varepsilon}) = O(n^{\log_2(7)-\varepsilon})$,

case a) applies and we get

$T(n)= \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = O(n^{2.81})$.

Discussion of Strassen's Algorithm

- Not always practical
  - constant factor is larger than for naïve method
  - specially designed methods are better on sparse matrices
  - issues of numerical (in)stability
  - recursion uses lots of space
- Not the fastest known method
  - Fastest known is $O(n^{2.376})$
  - Best known lower bound is $\Omega(n^2)$

Convex Hull

- The **Convex Hull** is the line completely enclosing a set of points in a plane so that there are no concavities in the line. More formally, we can describe it as the smallest convex polygon which encloses a set of points such that each point in the set lies within the polygon or on its perimeter.

Convex Hull

- **Input** = a set S of n points
  Assume that there are at least 2 points in the input set S of points
- **QuickHull** (S)
  {
  // Find convex hull from the set S of n points
- Convex Hull := {}
  Find left and right most points, say A & B, and add A & B to convex hull
  Segment AB divides the remaining (n-2) points into 2 groups S1 and S2
  where S1 are points in S that are on the right side of the oriented line from A to B,
  and S2 are points in S that are on the right side of the oriented line from B to A
  FindHull (S1, A, B)
  FindHull (S2, B, A)
  }

- **FindHull** (Sk, P, Q)

  {

  // Find points on convex hull from the set Sk of points

  // that are on the right side of the oriented line from P to Q

- If Sk has no point,

  then return.

  From the given set of points in Sk, find farthest point, say C, from segment PQ

  Add point C to convex hull at the location between P and Q

  Three points P, Q, and C partition the remaining points of Sk into 3 subsets: S0, S1, and S2

  where S0 are points inside triangle PCQ, S1are points on the right side of the oriented line from P to C, and S2 are points on the right side of the oriented line from C to Q.

  FindHull(S1, P, C)

  FindHull(S2, C, Q)

  }

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array

- **Algorithm: Max-Min-Element (numbers[])**

```
max := numbers[1]
min := numbers[1]
for i = 2 to n do
     if numbers[i] > max then
          max := numbers[i]
     if numbers[i] < min
          then min := numbers[i]
return (max, min)
```

NUMBER OF COMPARISONS 2n-2

Analysis

- The number of comparison in this method is **2n - 2**.
- The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide in Conquer

- In this approach, the array is divided into two halves.
- Then using recursive approach maximum and minimum numbers in each halves are found.
- Later, return the maximum of two maxima of each half and the minimum of two minima of each half.
- In this given problem, the number of elements in an array is y−x+1y−x+1, where **y** is greater than or equal to **x**

Divide and Conquer Approach

- **Algorithm: Max - Min(x, y)**

  if y – x ≤ 1 then

  return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))

  else

  (max1, min1):= maxmin(x, $\lfloor((x + y)/2)\rfloor$)

  (max2, min2):= maxmin($\lfloor((x + y)/2) + 1)\rfloor$,y)

  return (max(max1, max2), min(min1, min2))

Analysis

- Let ***T(n)*** be the number of comparisons made by Max–Min(x,y), where the number of elements n=y–x+1
- If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$\boxed{T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\, n > 2 \\ 1 & for\, n = 2 \\ 0 & for\, n = 1 \end{cases}}$$

O(N)

Greedy Algorithm

- An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.
- **Counting Coins**
  - This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −
    - Select one ₹ 10 coin, the remaining count is 8
    - Then select one ₹ 5 coin, the remaining count is 3
    - Then select one ₹ 2 coin, the remaining count is 1
    - And finally, the selection of one ₹ 1 coins solves the problem

Greedy Solution

- Find feasible solutions from different solutions
- Find optimal solution from that.
  - Examples:
  - Knapsack problem using greedy method
  - Job sequencing with deadlines.
  - Dijkstra’s Algorithm
  - Prim’s Algorithm
  - Huffman’s tree
  - Single source shortest path.

General Greedy Method

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Job Sequencing with deadlines

- Problem: n jobs, S={1, 2, …, n}, each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit p_i if job i is completed by its deadline.

i	1	2	3	4	5
p_i	20	15	10	5	1
d_i	2	2	1	3	3

The optimal solution = {1, 2, 4}.

The total profit = 20 + 15 + 5 = 40.

Algorithm:

Step 1: Sort p_i into nonincreasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n$.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that $1 \leq r \leq d_i$ and [r-1, r] is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

e.g.

i	p_i	d_i	
1	20	2	assign to [1, 2]
2	15	2	assign to [0, 1]
3	10	1	reject
4	5	3	assign to [2, 3]
5	1	3	reject

solution = {1, 2, 4}

total profit = 20 + 15 + 5 = 40

Job Sequencing with deadlines

Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1)  step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
$\{1\}$	[1, 2]	2	assign to [0, 1]	20
$\{1, 2\}$	[0, 1], [1, 2]	3	cannot fit; reject	35
$\{1, 2\}$	[0, 1], [1, 2]	4	assign to [2, 3]	35
$\{1, 2, 4\}$	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. $\square$

The knapsack algorithm

- Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the **maximum total value in the knapsack.**
- In the **0-1 Knapsack problem**, we are not allowed to break items. We either take the whole item or don’t take it.
- In **Fractional Knapsack**, we can break items for maximizing the total value/profit of knapsack.

Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance.

The knapsack algorithm

- The greedy algorithm:
  Step 1: Sort p_i/w_i into nonincreasing order.
  Step 2: Put the objects into the knapsack according
  to the sorted sequence as possible as we can.
- e. g.
  $n = 3$, $M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$
  $(w_1, w_2, w_3) = (18, 15, 10)$
  Sol: $p_1/w_1 = 25/18 = 1.39$
  $p_2/w_2 = 24/15 = 1.6$
  $p_3/w_3 = 15/10 = 1.5$
  Optimal solution: $x_1 = 0$, $x_2 = 1$, $x_3 = 1/2$
  total profit = 24 + 7.5 = 31.5

Greedy-fractional-knapsack (***w***, ***v***, ***W***)

```
FOR i =1 to n
    do x[i] =0
weight = 0
while weight < W
    do i = best remaining item
        IF weight + w[i] ≤ W
            then x[i] = 1
                weight = weight + w[i]
            else
                x[i] = (w - weight) / w[i]
                weight = W
return x
```

Profit Calculation

```
    profit=0
        for i= 1 to n
profit=profit+(P[i]*x[i])
        End of Algorithm
```

Complexity

Worst Case: n^2

Average case: nlogn

Analysis

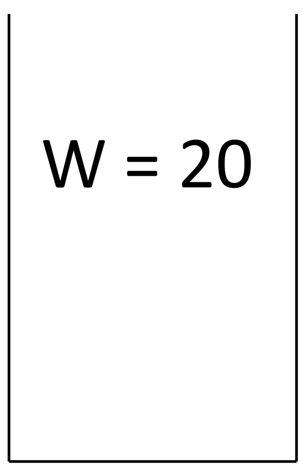
- If the provided items are already sorted into a decreasing order of **piwi** , then the while loop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

0-1 Knapsack problem

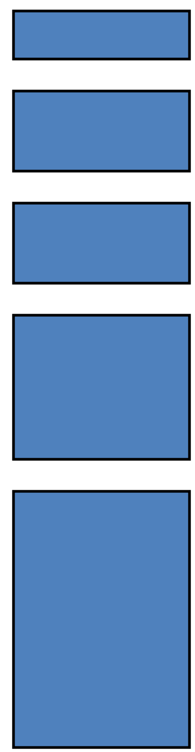
- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i , b_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20



Items



Weight w_i	Benefit value b_i
2	3
3	4
4	5
5	8
9	10

[0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \leq m$$

$$\text{and} \quad x_i = 0 \text{ or } 1, \; 1 \leq i \leq n$$

The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

0-1 Knapsack problem: brute-force approach

Let’s first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

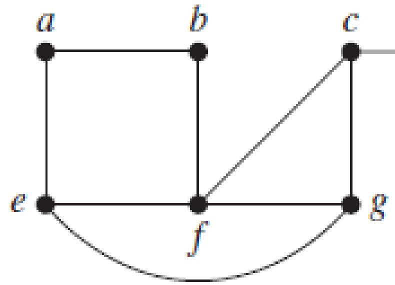
If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

S_k = *{items labeled 1, 2, .. k}*

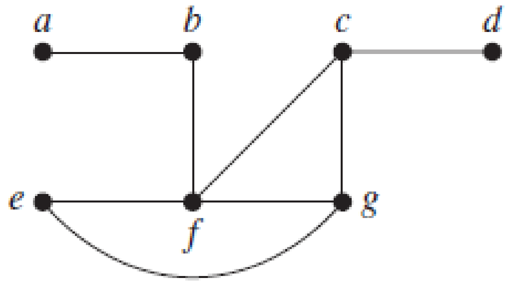
Spanning tree

- Let *G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing* every vertex of *G.*
- A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices. The converse is also true; that is, every connected simple graph has a spanning tree.

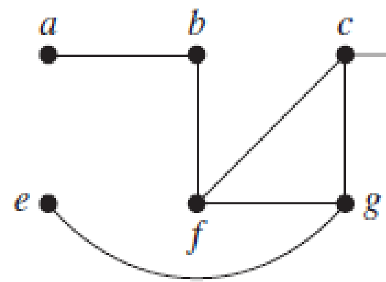
- Find a spanning tree of the simple graph *G*



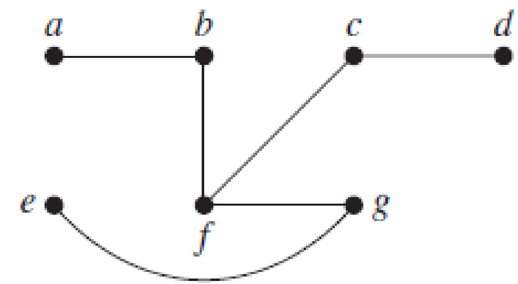
- *solution*



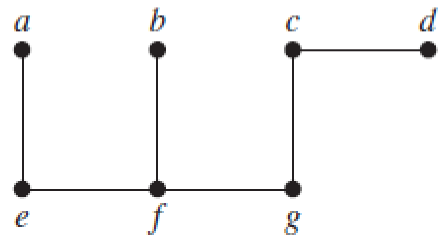
Edge removed: {*a*, *e*}

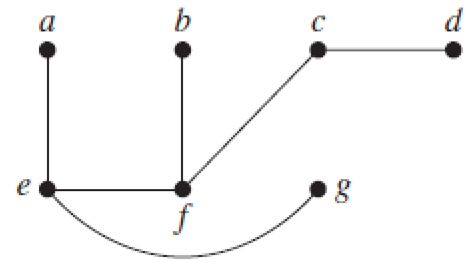


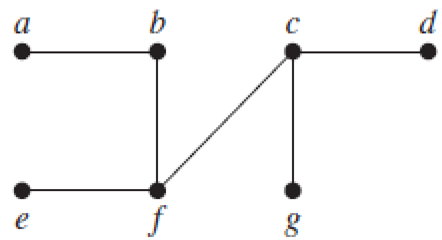
{*e*, *f*}

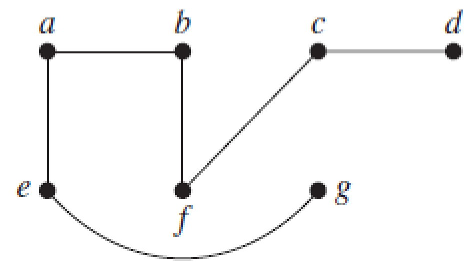


{*c*, *g*}









Minimum Spanning Tree (MST)

- A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

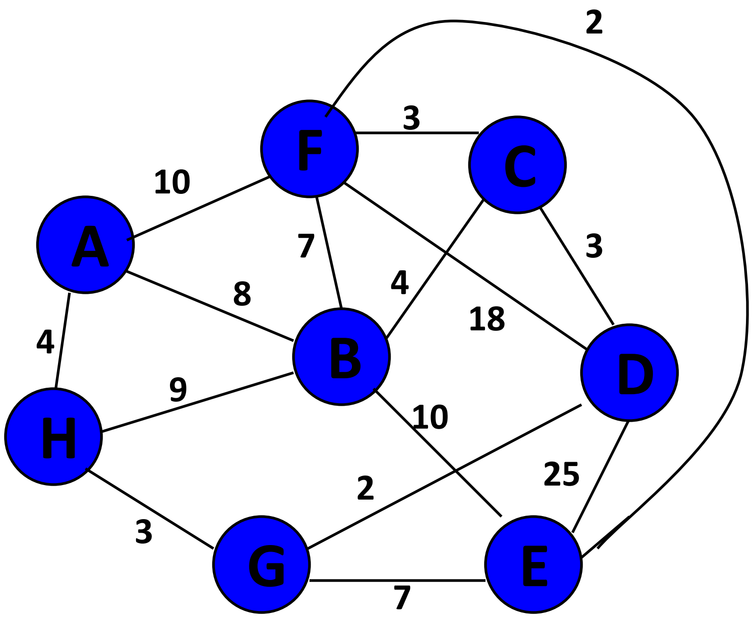
Algorithm Characteristics

- Both Prim's and Kruskal's Algorithms work with undirected graphs
- Both work with weighted and unweighted graphs but are more interesting when edges are weighted
- Both are greedy algorithms that produce optimal solutions

Prim's Algorithm

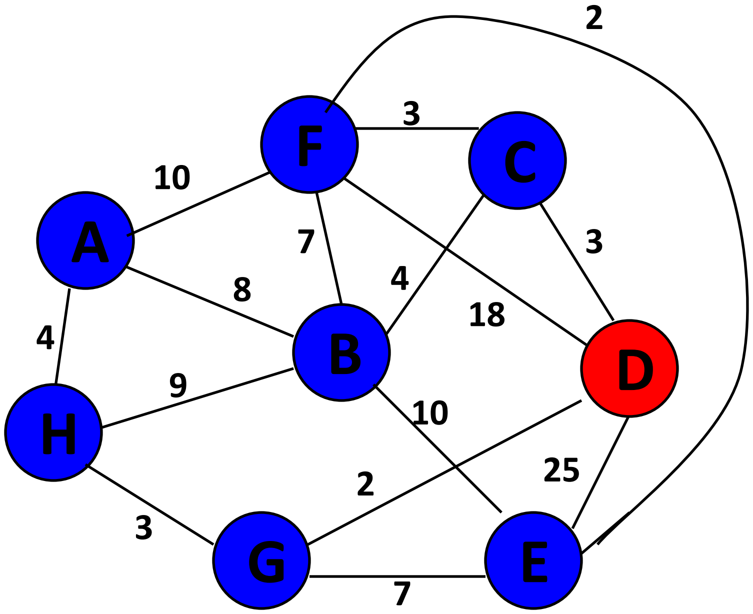
- Similar to Dijkstra's Algorithm except that d_v records edge weights, not path lengths

Walk-Through



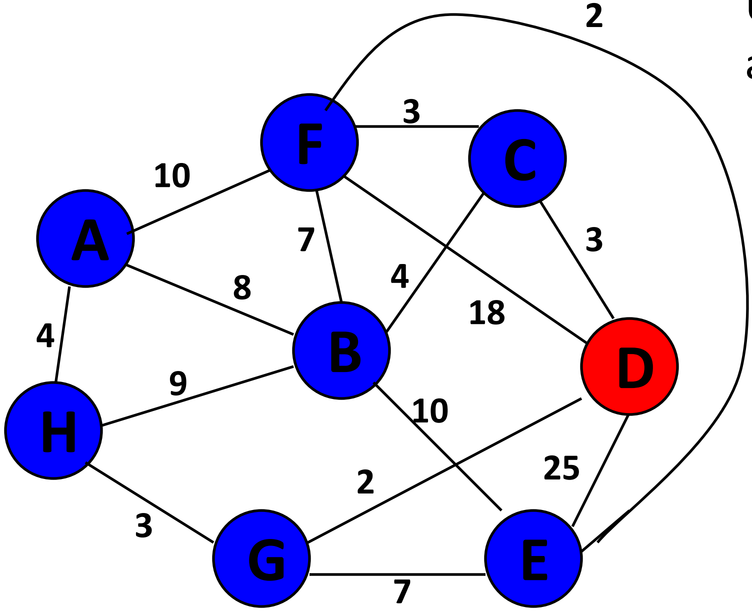
Initialize array

	K	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–



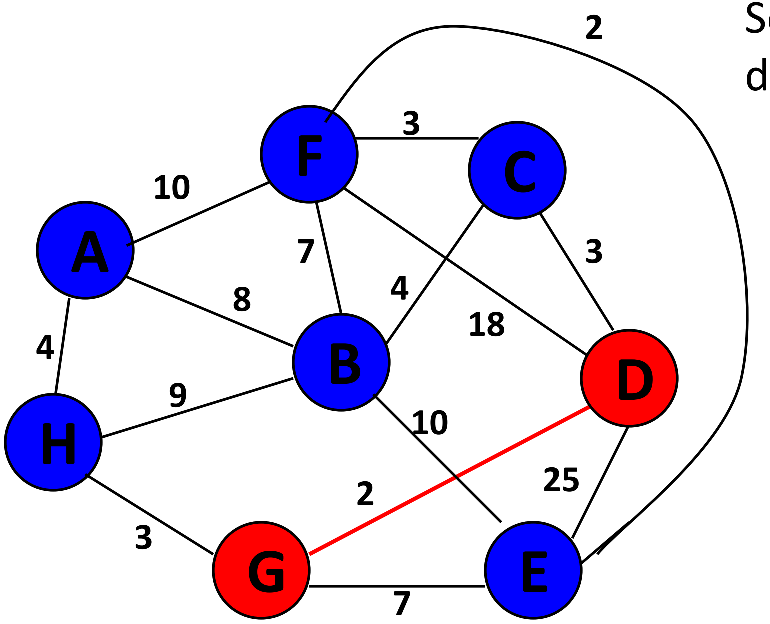
Start with any node, say D

	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			



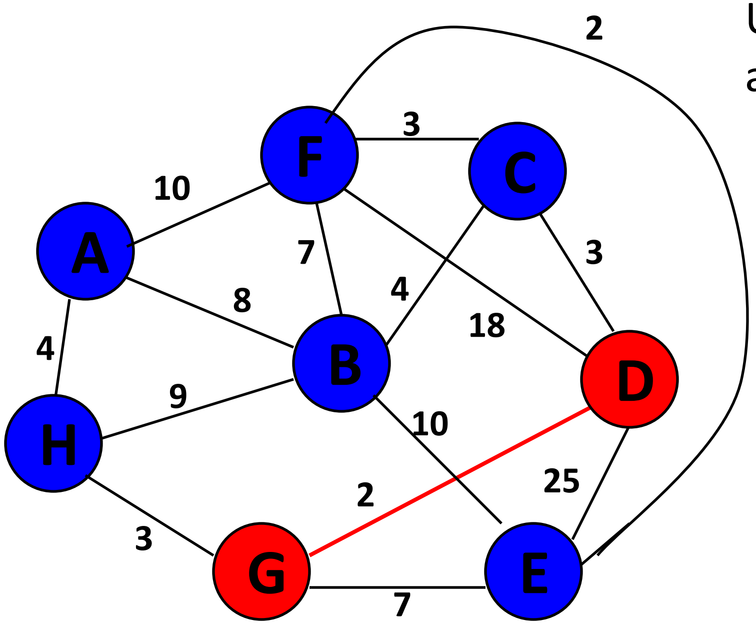
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			



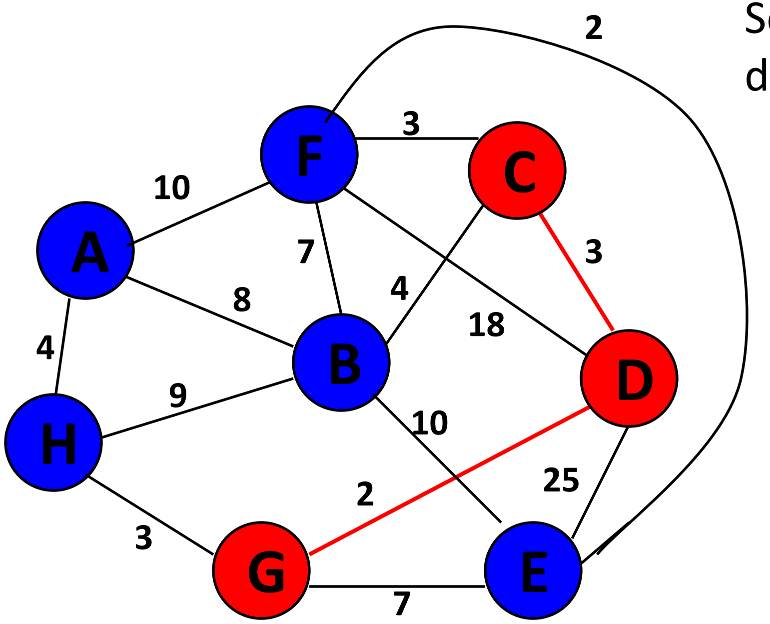
Select node with minimum distance

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			



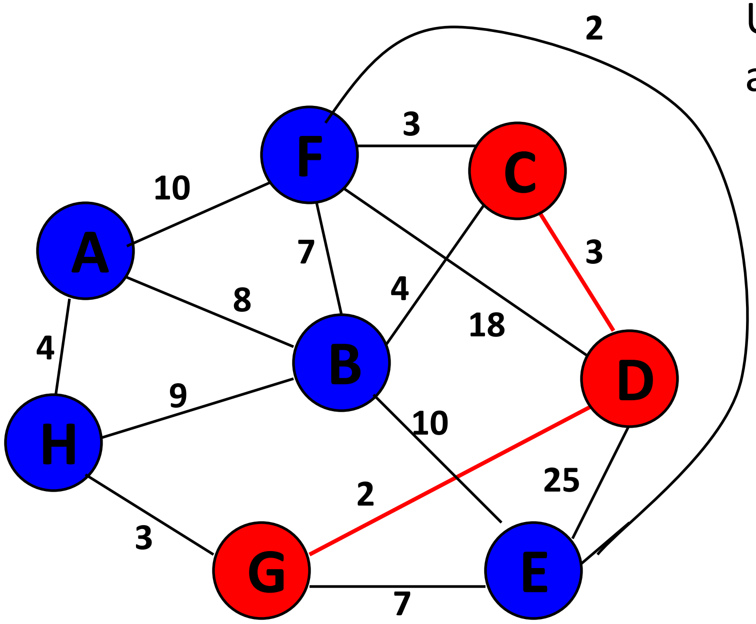
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



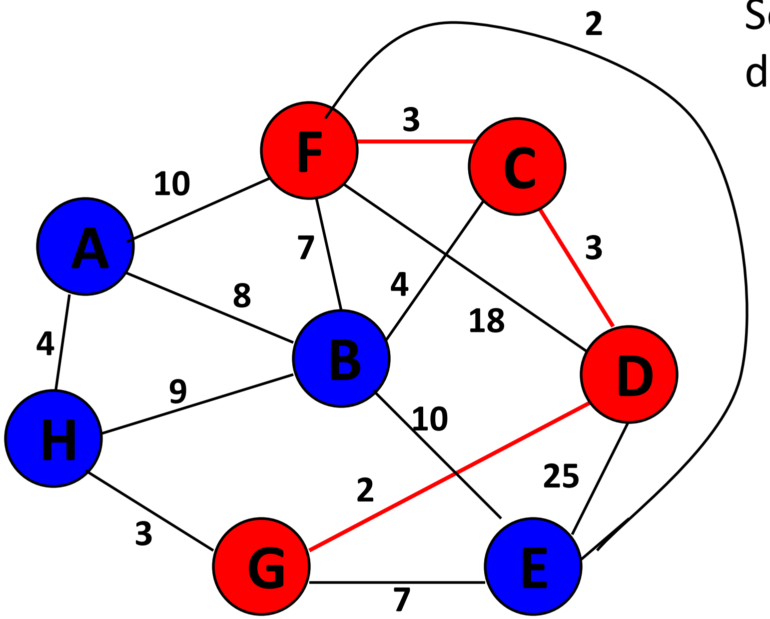
Select node with minimum distance

	K	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



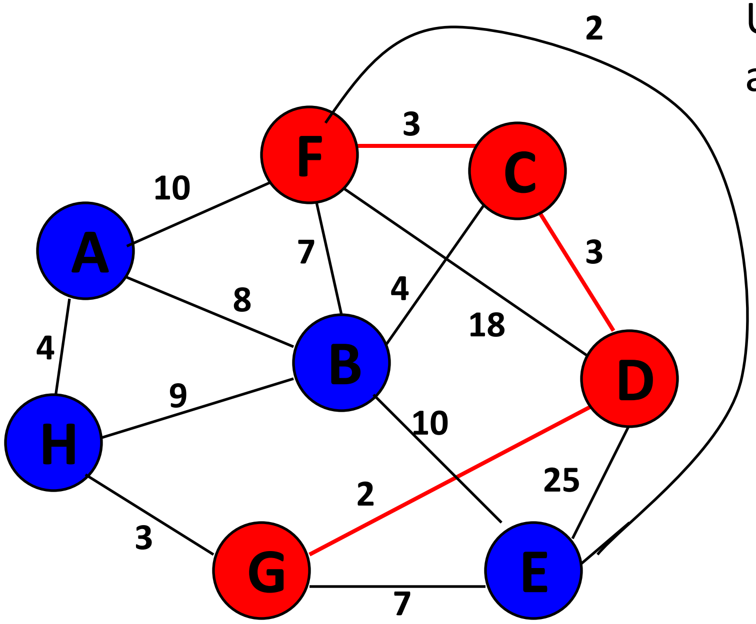
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G



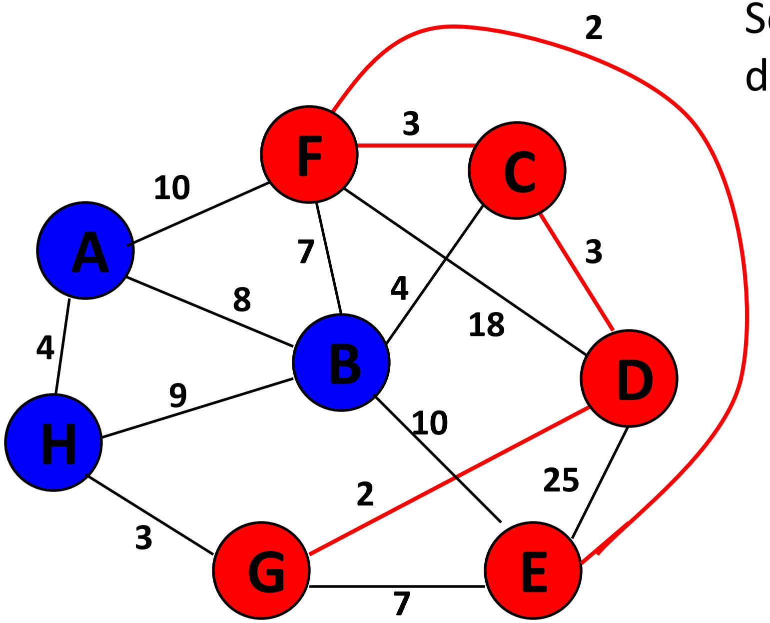
Select node with minimum distance

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G



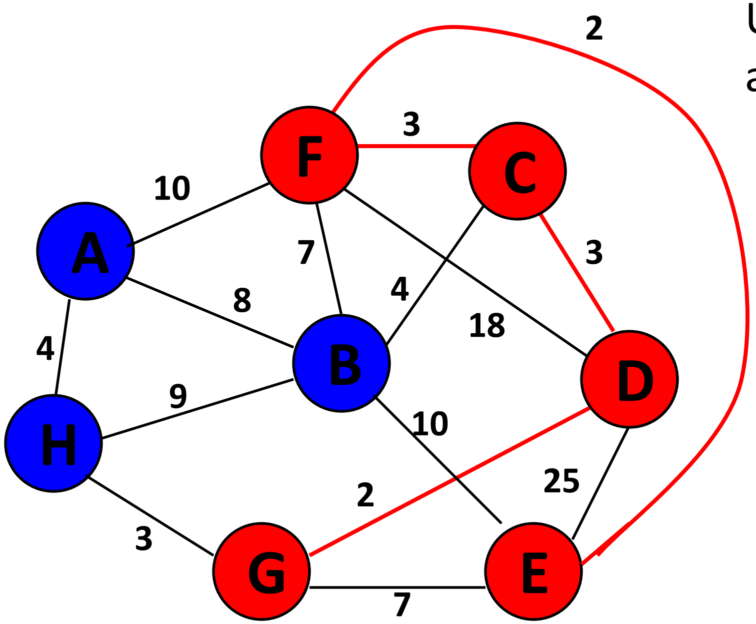
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

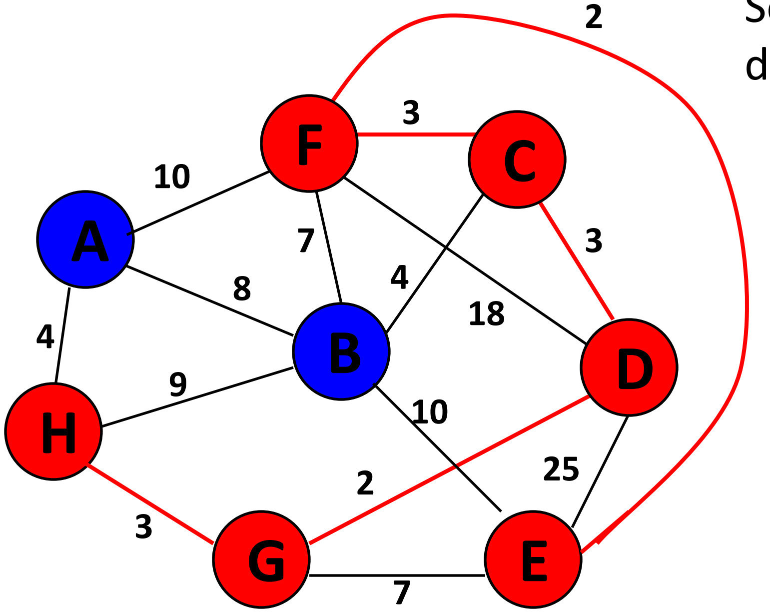
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

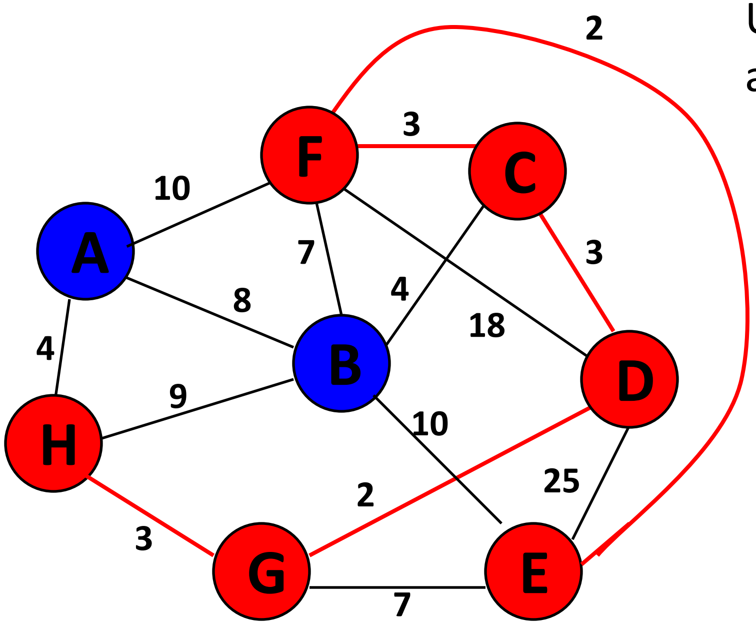
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



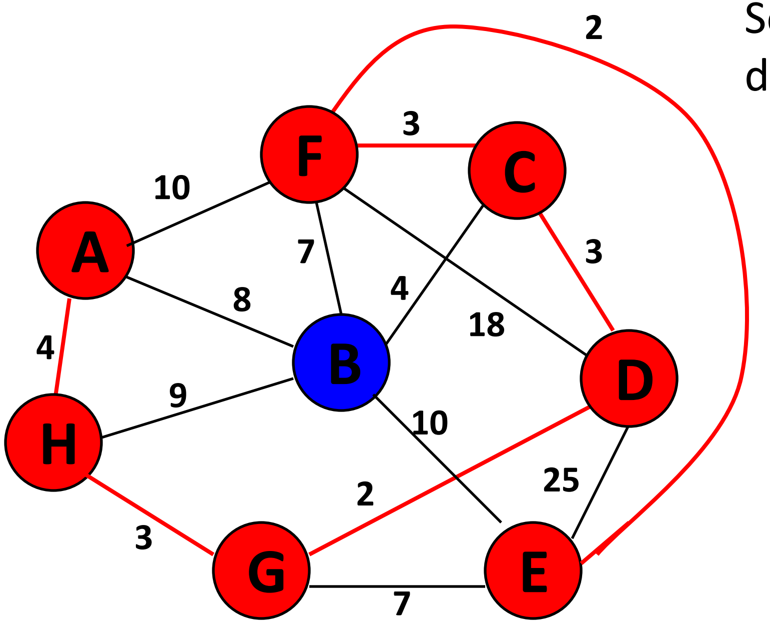
Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



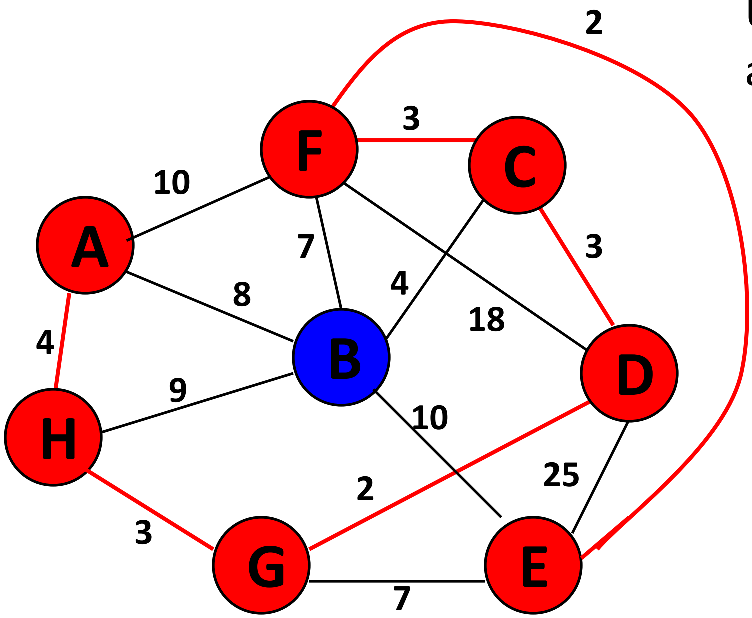
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

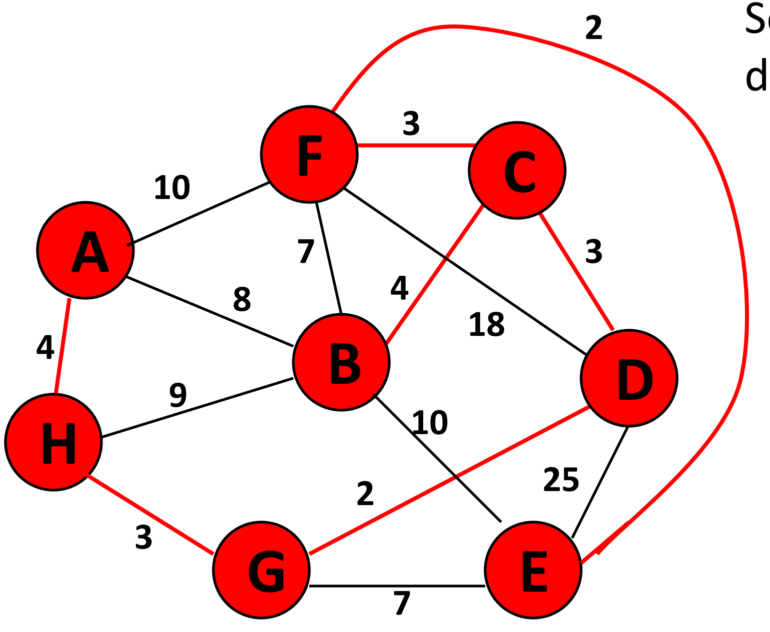
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of
adjacent, unselected nodes

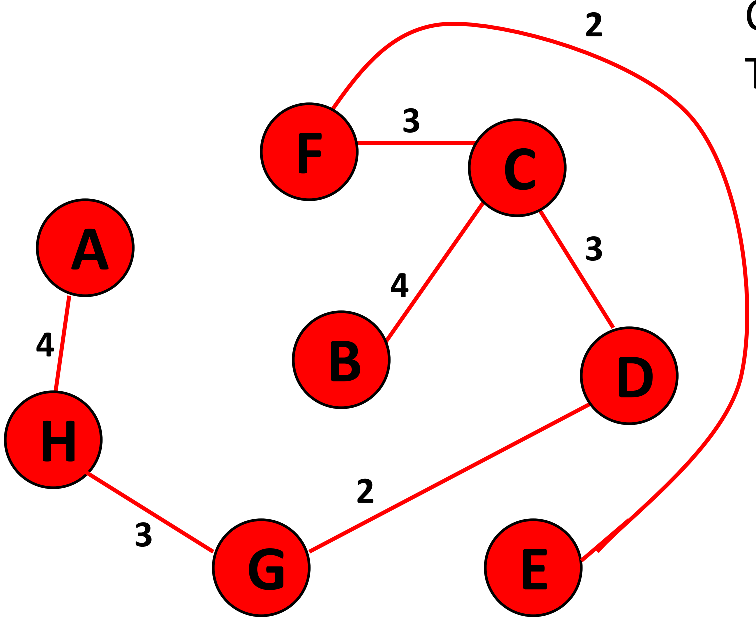
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

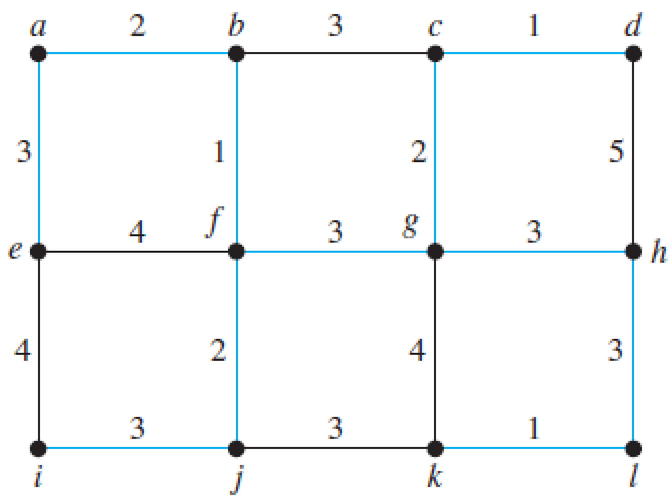
	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = Σ d_v = 21

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done



(a)

Choice	Edge	Weight
1	$\{b,\ f\}$	1
2	$\{a,\ b\}$	2
3	$\{f,\ j\}$	2
4	$\{a,\ e\}$	3
5	$\{i,\ j\}$	3
6	$\{f,\ g\}$	3
7	$\{c,\ g\}$	2
8	$\{c,\ d\}$	1
9	$\{g,\ h\}$	3
10	$\{h,\ l\}$	3
11	$\{k,\ l\}$	1
	Total:	24

(b)

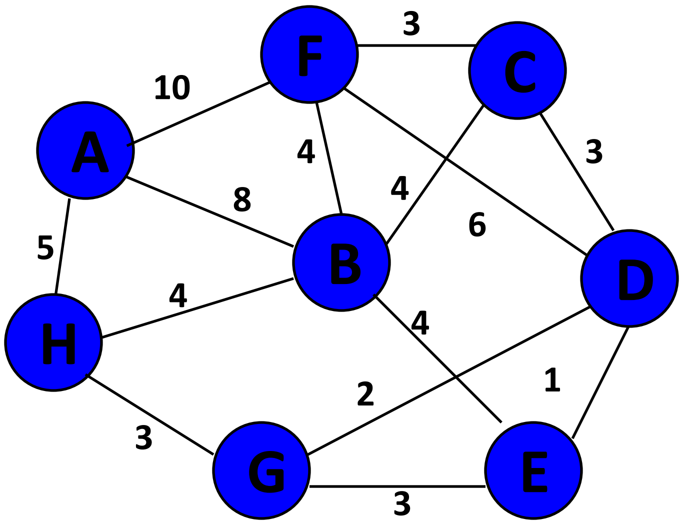
Kruskal's Algorithm

Work with edges, rather than nodes

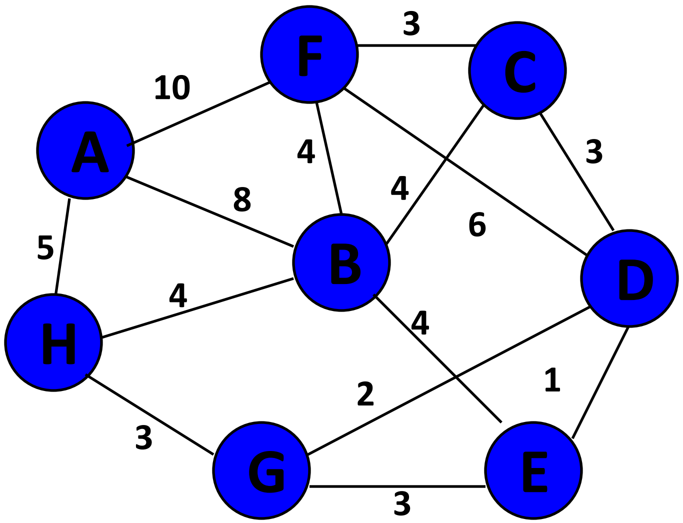
Two steps:

- Sort edges by increasing edge weight
- Select the first $|V| - 1$ edges that do not generate a cycle

Walk-Through



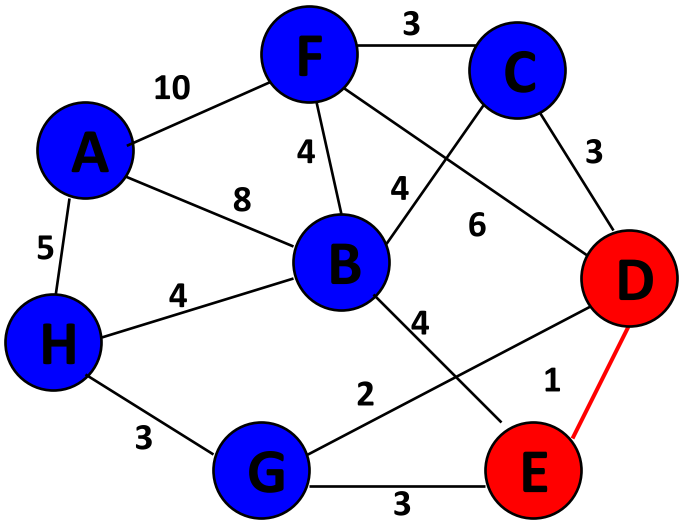
Consider an undirected, weight graph



Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

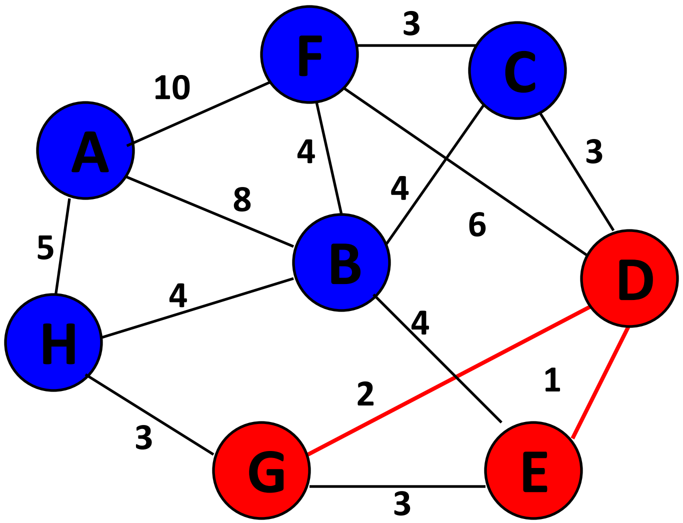
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

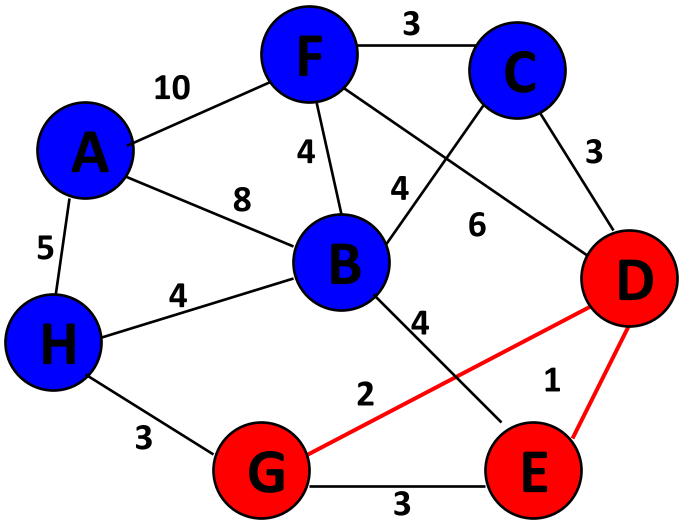
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

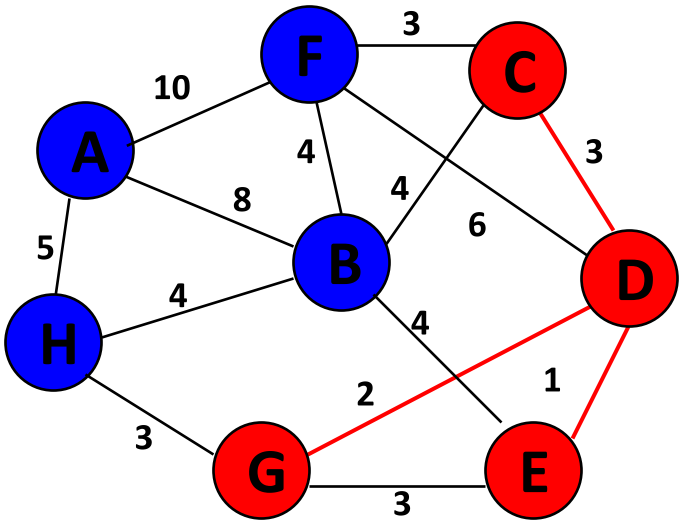


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

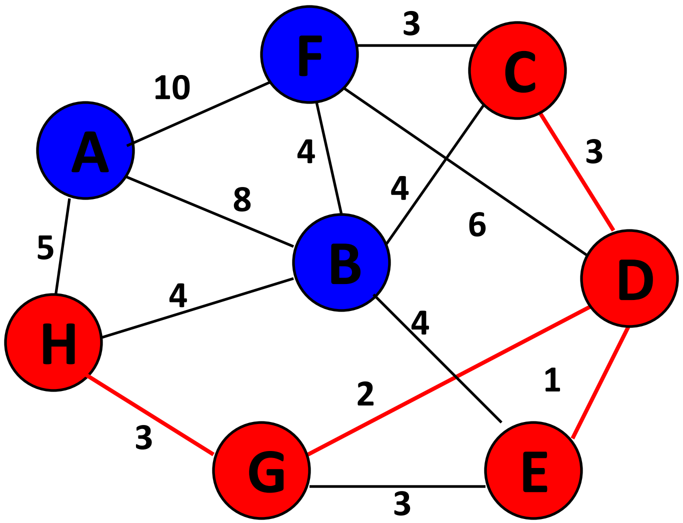
Accepting edge (E,G) would create a cycle



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

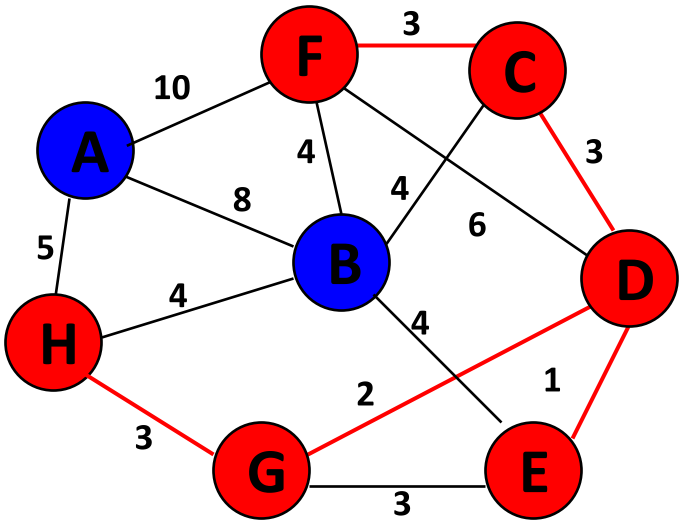
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

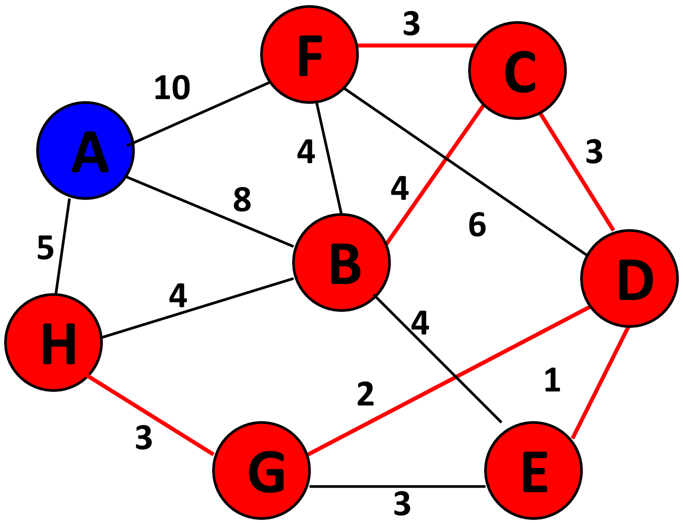
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

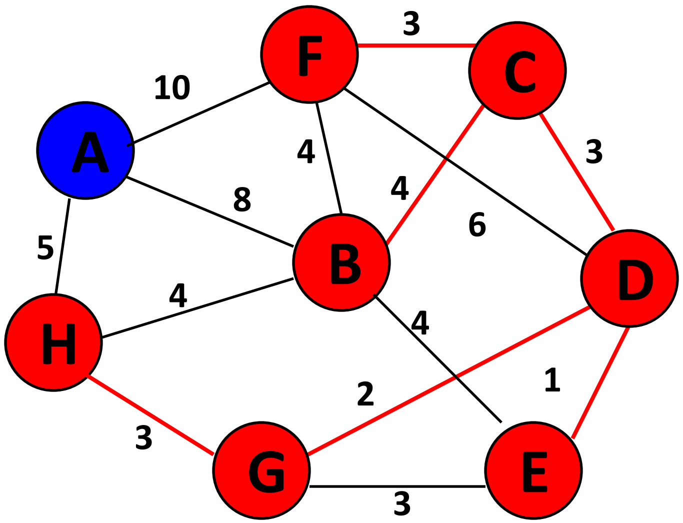
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

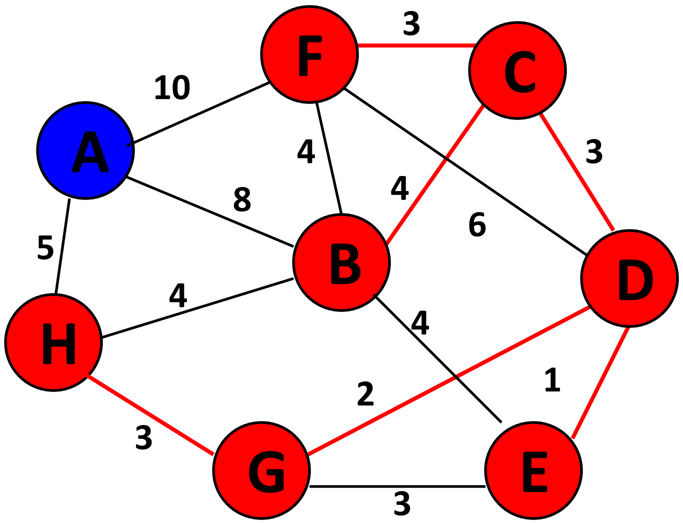
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

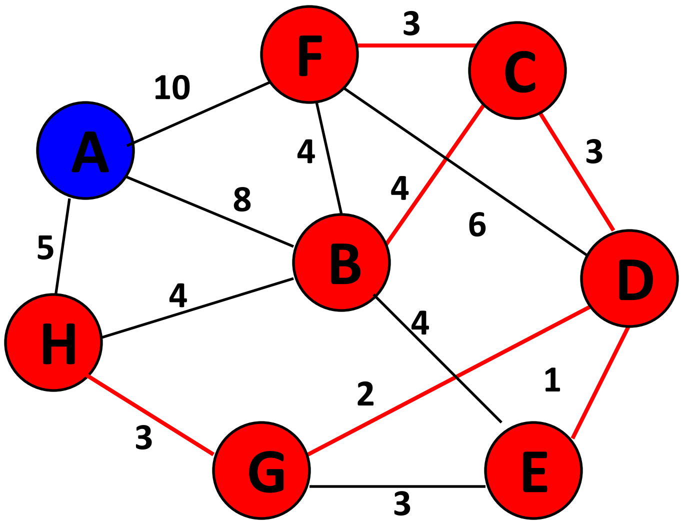
edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

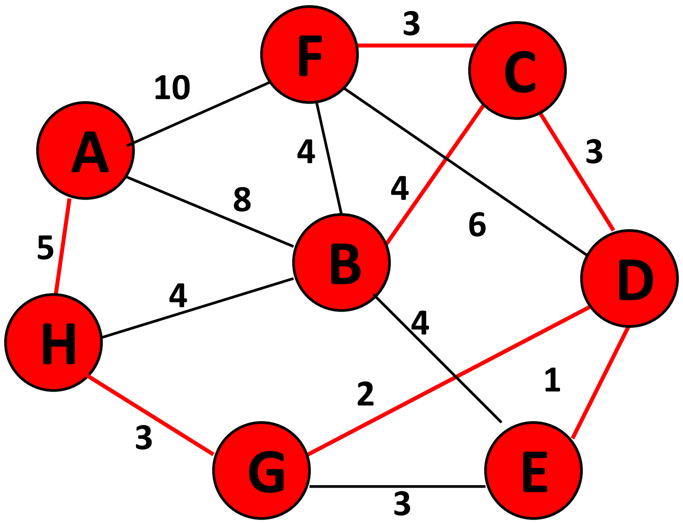
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

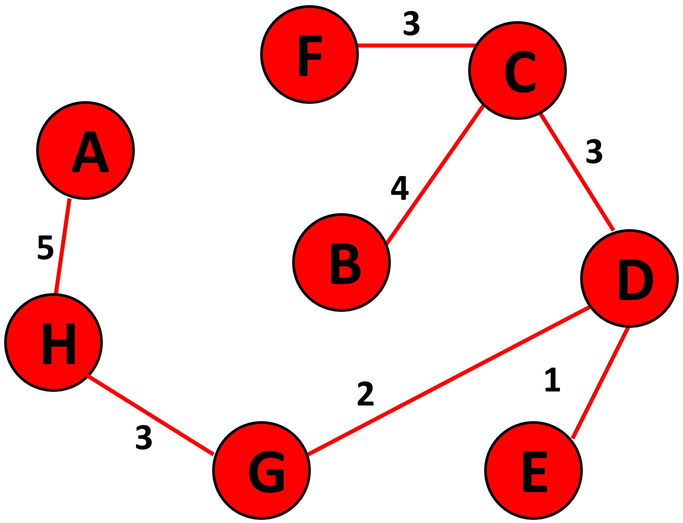
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

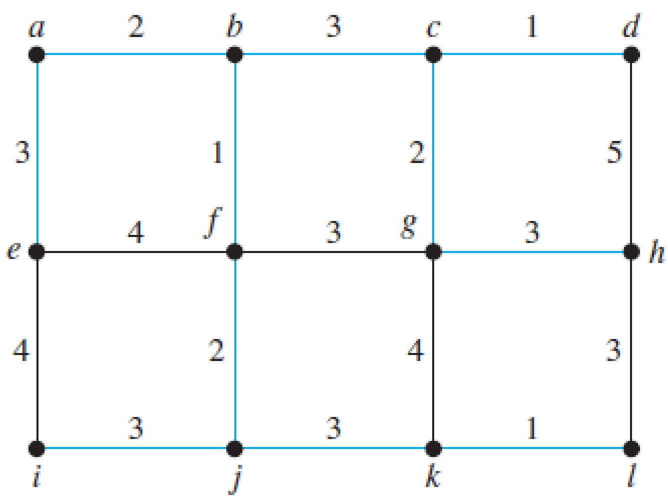
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

not considered

Done

Total Cost = $\sum d_v = 21$



(a)

Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

(b)

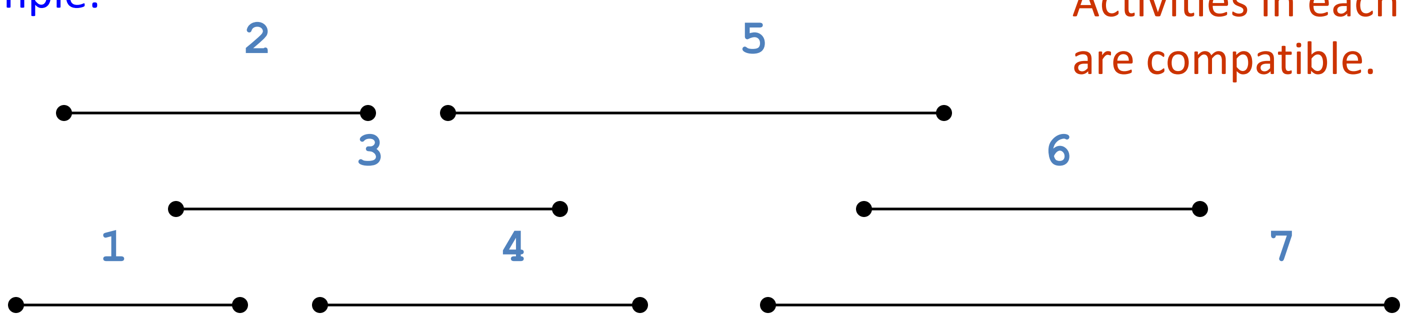
Activity-Selection Problem

- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Activity-selection Problem

- Input: Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don't overlap.

Example:



Activities in each line are compatible.

Greedy Choice Property

- Activity selection problem exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal sol'n
  - Theorem: if S is an activity selection problem sorted by finish time, then $\exists$ optimal solution $A \subseteq S$ such that $\{1\} \in A$
    - Sketch of proof: if $\exists$ optimal solution B that does not contain $\{1\}$, can always replace first activity in B with $\{1\}$. Same number of activities, thus optimal.

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time.

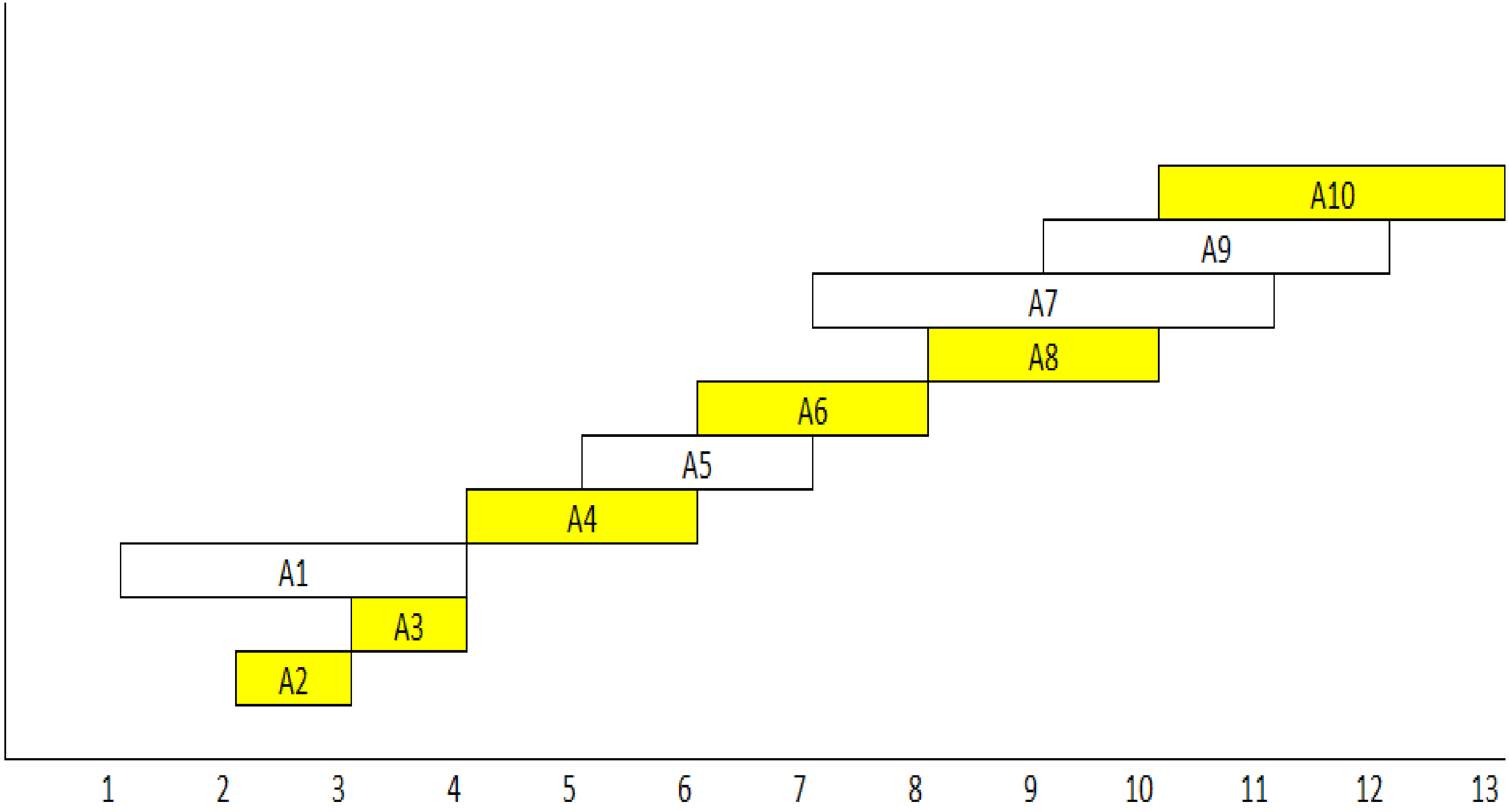
Example

Given 10 Activities with their start and finish time

- $S=< A_1,A_2,A_3,A_4,A_5,A_6,A_7,A_8,A_9,A_{10}>$
- Si=<1,2,3,4,5,6,7,8,9,10>
- Fi=<5,3,4,6,7,8,11,10,12,13>
- Sort in increasing order of finish time

Activity	A_2	A_3	A_1	A_4	A_5	A_6	A_8	A_7	A_9	A_{10}
Start	2	3	1	4	5	6	8	7	9	10
Finish	3	4	5	6	7	8	10	11	12	13

Example



Solution: <$A_{2,}$ $A_{3,}$ $A_{4,}$ $A_{6,}$ $A_{8,}$,A_{10}>

Example

Given 10 Activities with their start and finish time

Activity	1	2	3	4	5	6	7	8	9	10	11
Start	1	3	0	5	3	5	6	8	8	2	12
Finish	4	5	6	7	9	9	10	11	13	14	16

Example

Given 10 Activities with their start and finish time

Activity	1	2	3	4	5	6	7	8	9	10	11
Start	1	3	0	5	3	5	6	8	8	2	12
Finish	4	5	6	7	9	9	10	11	13	14	16

Solution: <1,4,8,11>

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

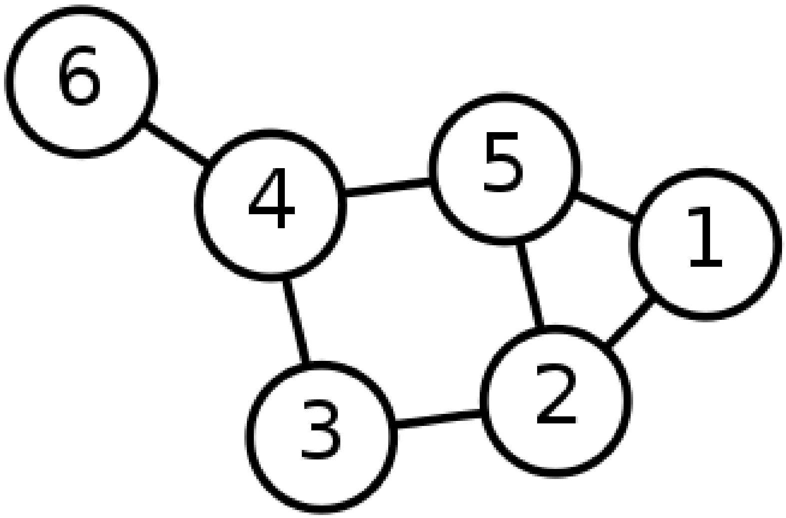
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n ← length[s]
A ← {a_1}
i ← 1
for m ← 2 to n
        do if s_m ≥ f_i
              then A ← A ∪ {a_m}
                   i ← m
return A
```

Elements of Greedy Algorithms

- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

Single-Source Shortest Path Problem

<u>**Single-Source Shortest Path Problem**</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

- The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v and u.
- the algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C
- Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between v and u. The algorithm will update a D[u] value when it finds a shorter path from v to u.
- When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex v and vertex u.

1. Initialize the cost of each node to ∞
2. Initialize the cost of the source to 0
3. While there are unknown nodes left in the graph
   1. Select the unknown node N with the *lowest cost* (greedy choice)
   2. Mark N as known
   3. For each node A adjacent to N
      If (N's cost + cost of (N, A)) < A's cost
         A's cost = N's cost + cost of (N, A)
         Prev[A] = N *//store preceding node*

Analysis

- Main loop:

  While there are unknown nodes left in the graph $\longleftarrow |V|$ times
  1. Select the unknown node N with the *lowest cost* $\longleftarrow O(|V|)$
  2. Mark N as known
  3. For each node A adjacent to N $\longleftarrow O(|E|)$ total

     If (N's cost + cost of (N, A)) < A's cost

       A's cost = N's cost + cost of (N, A)

Total time = $|V|\ (O(|V|)) + O(|E|) = O(|V|^2 + |E|)$

Dense graph: $|E| = \Theta(|V|^2) \rightarrow$ Total time $= O(|V|^2) = O(|E|)$ √

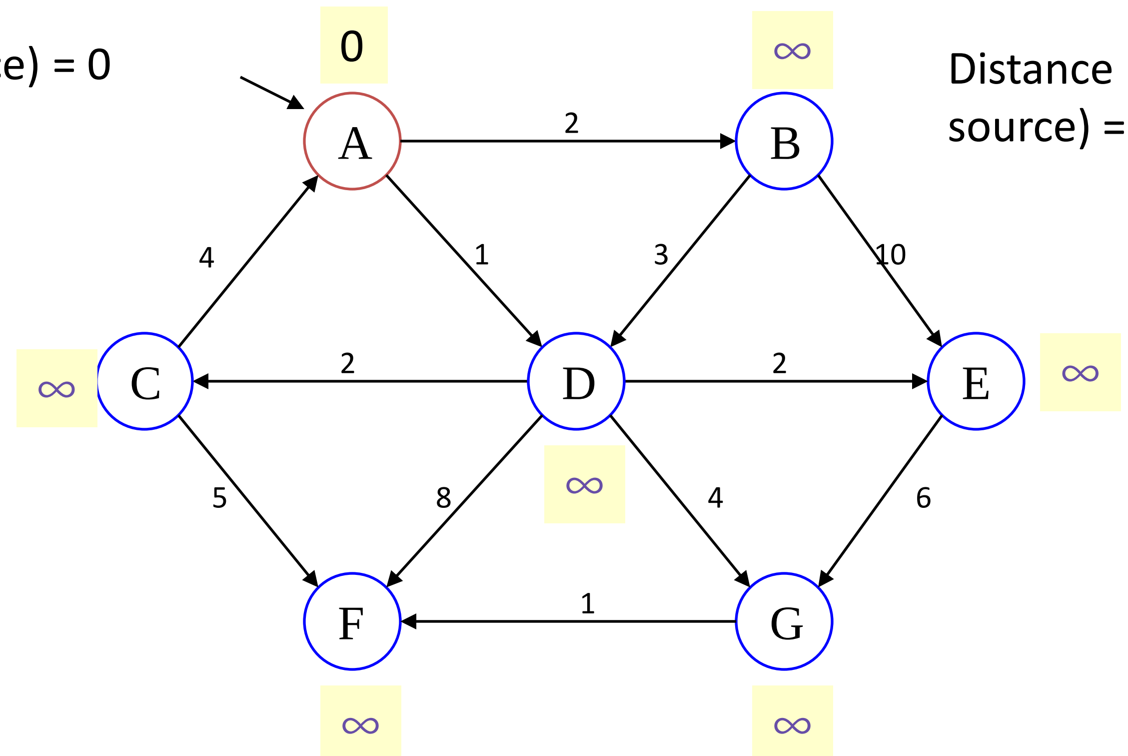
Sparse graph: $|E| = \Theta(|V|) \rightarrow$ Total time $= O(|V|^2) = O(|E|^2)$ χ

Quadratic! Can we do better?

Example: Initialization

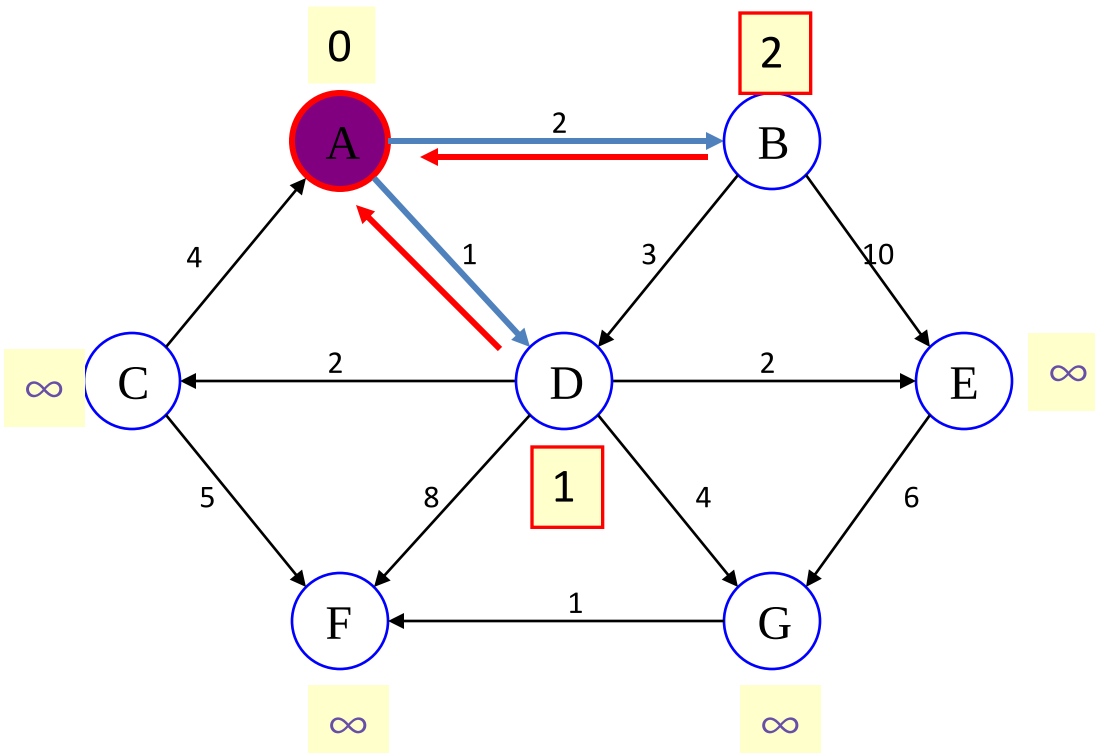
Distance(source) = 0

Distance (all vertices but source) = ∞



Pick vertex in List with minimum distance.

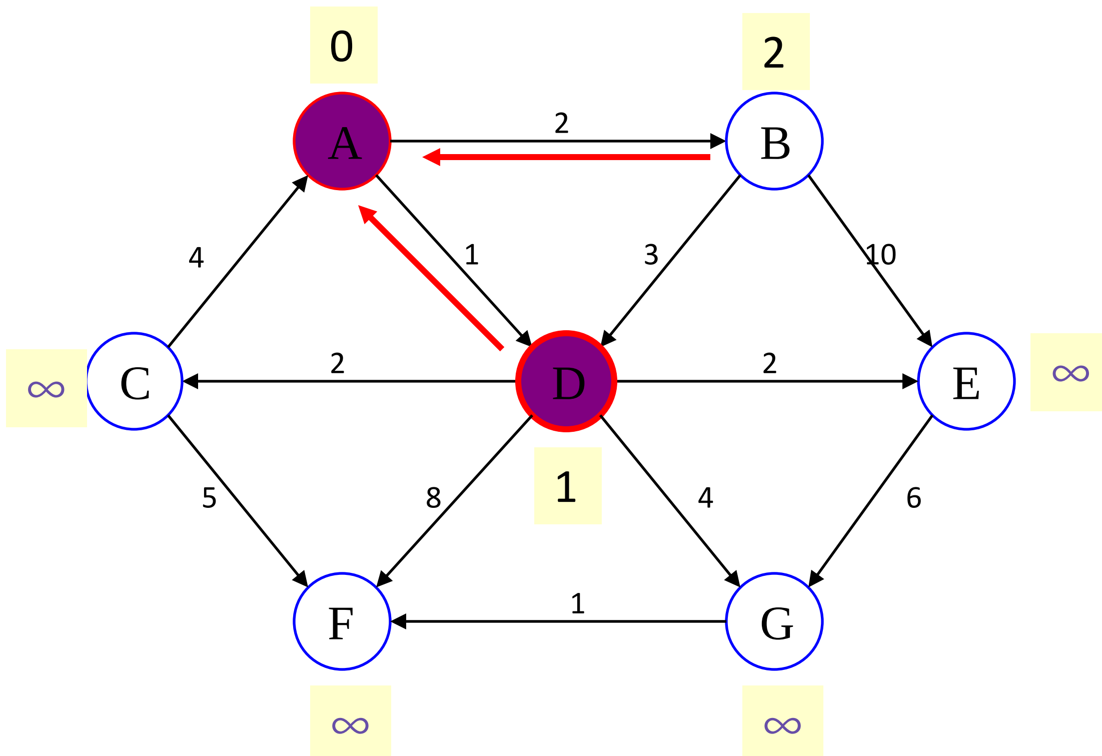
Example: Update neighbors' distance



Distance(B) = 2

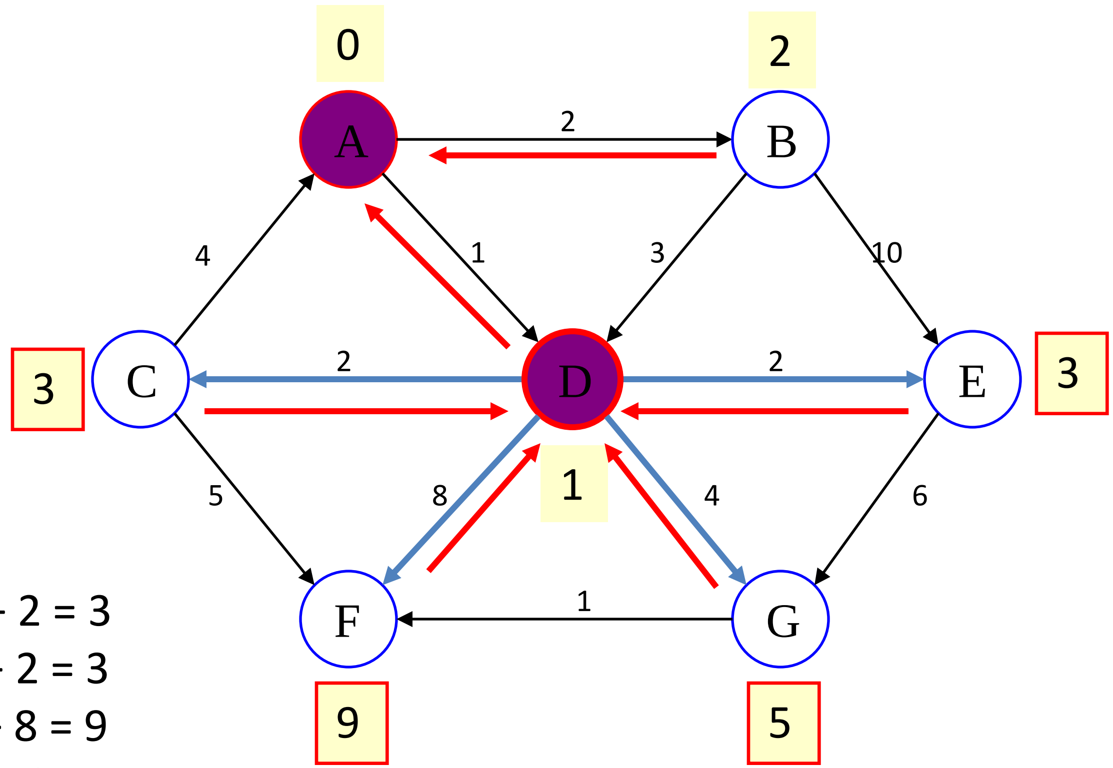
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

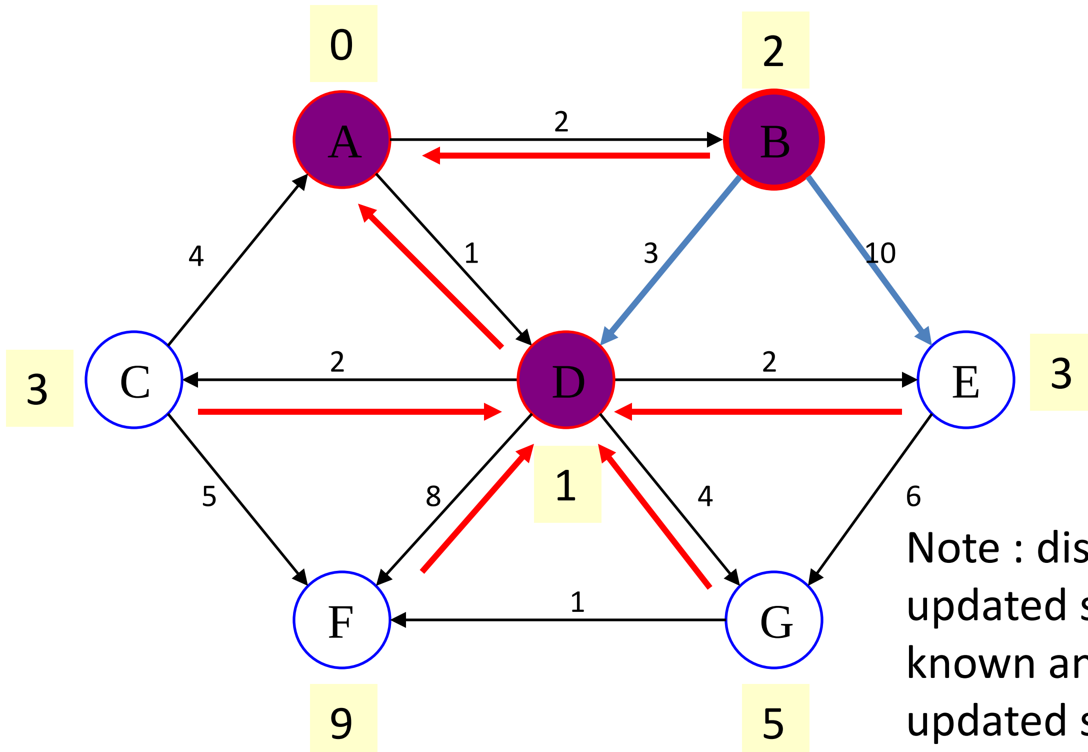
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

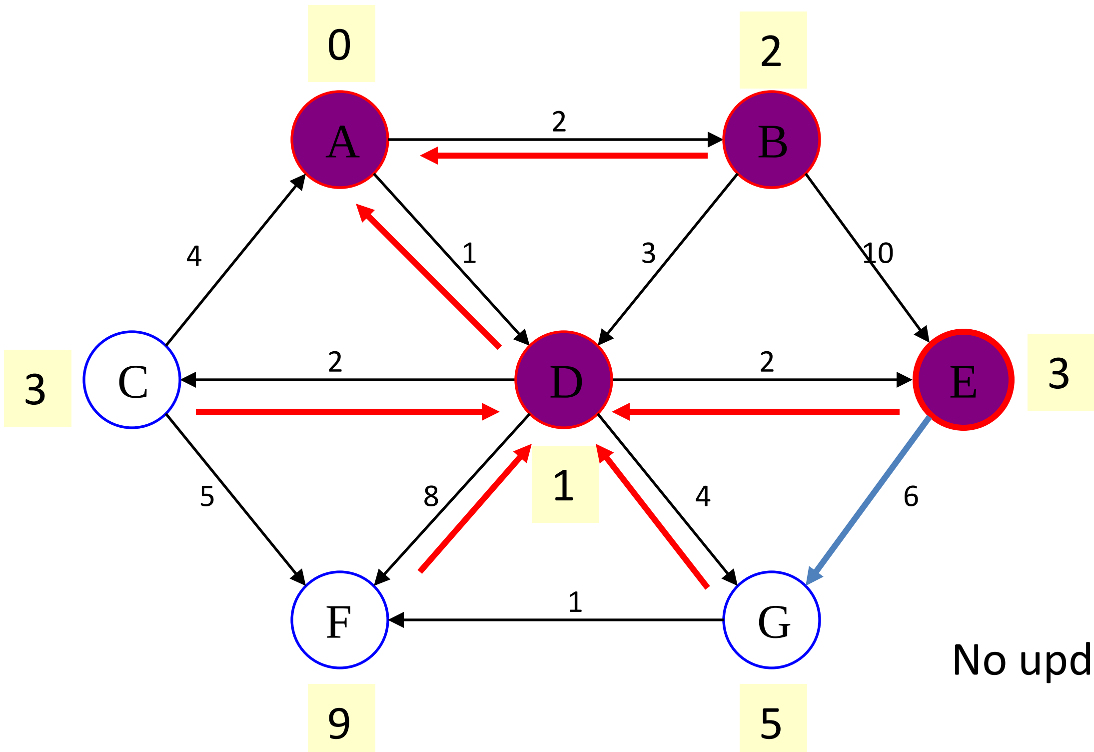
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

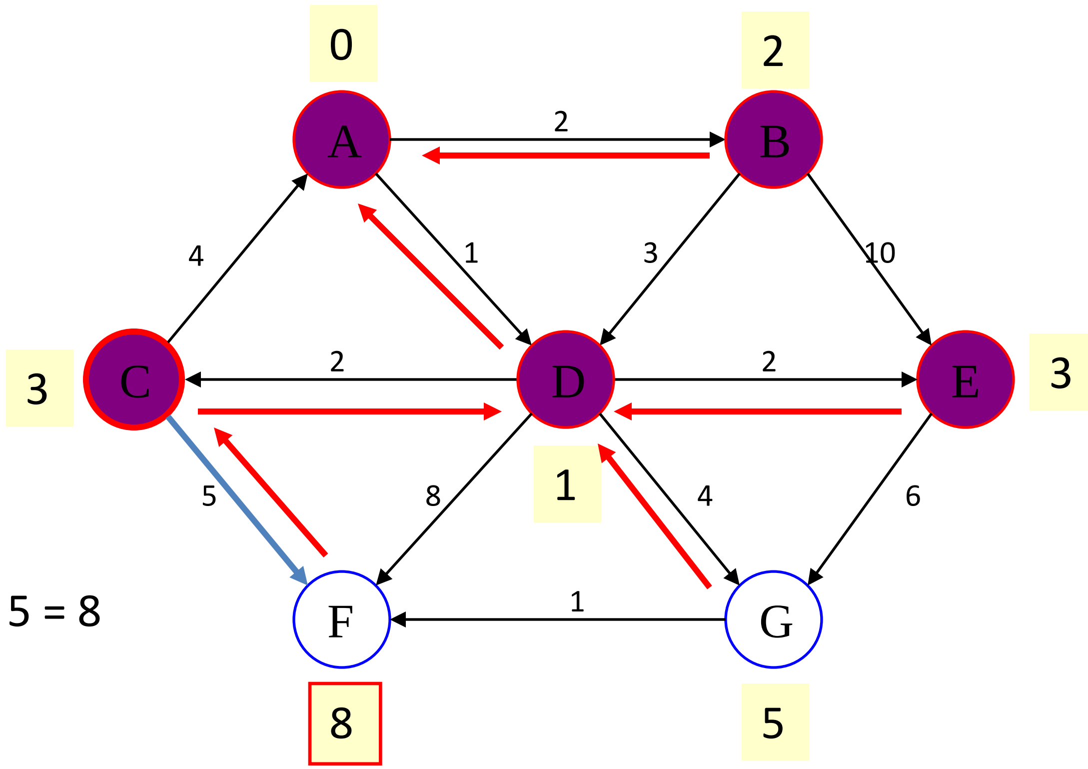
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

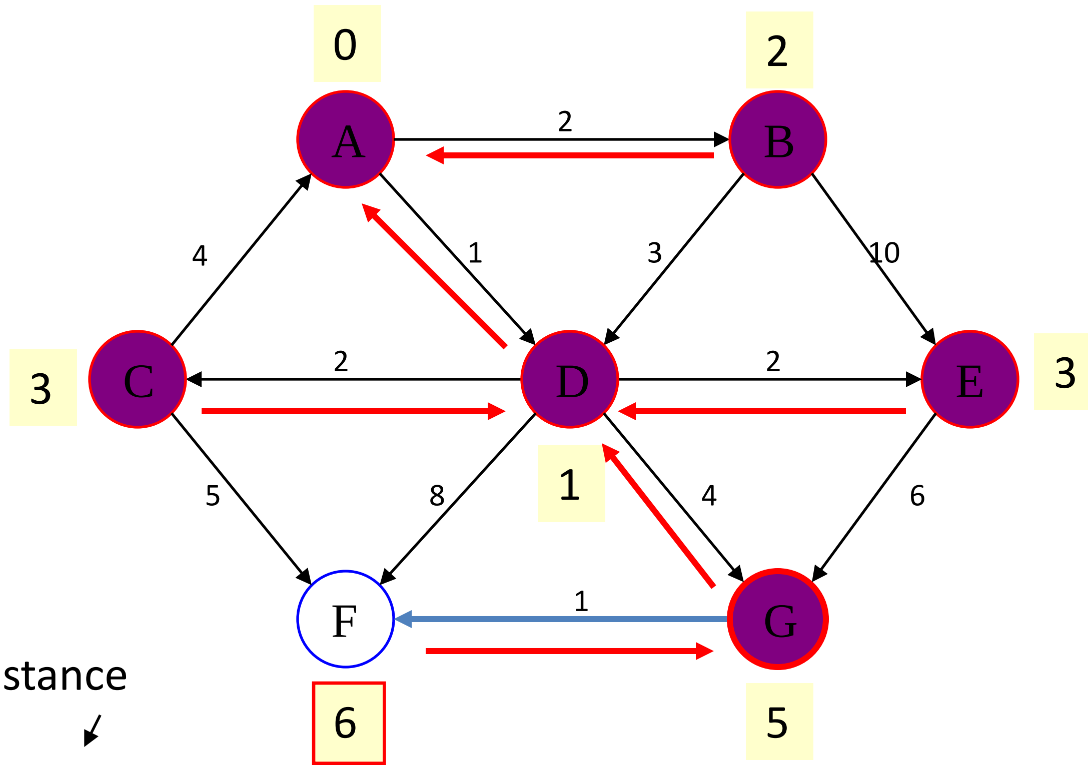
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

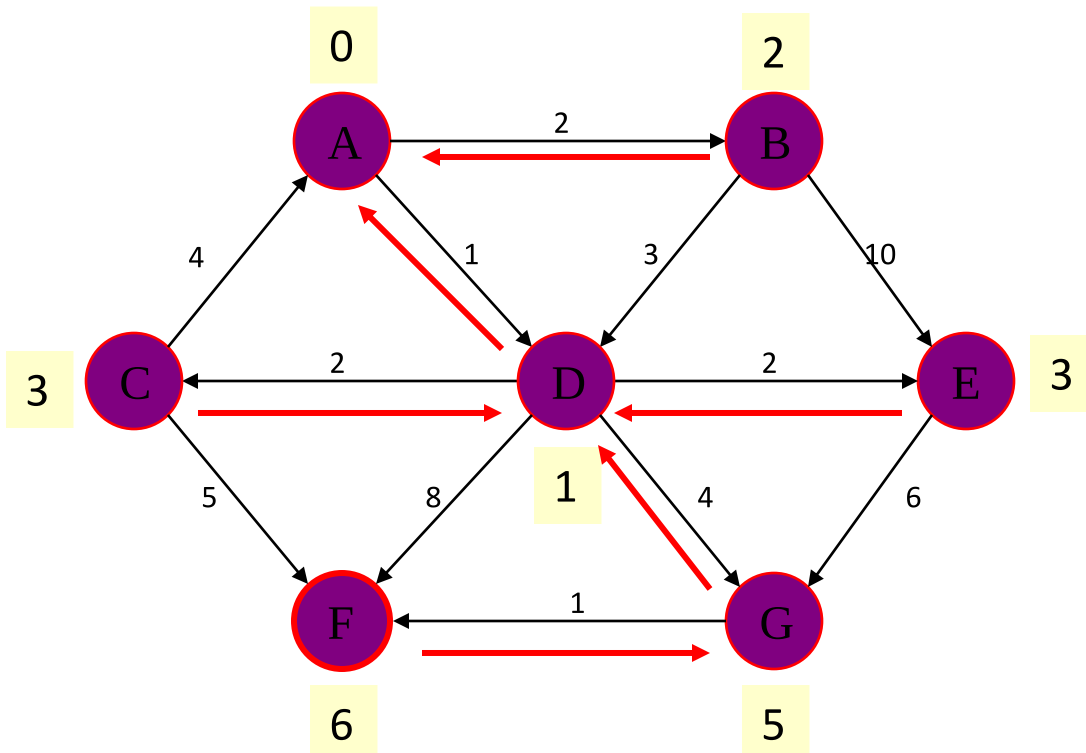
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

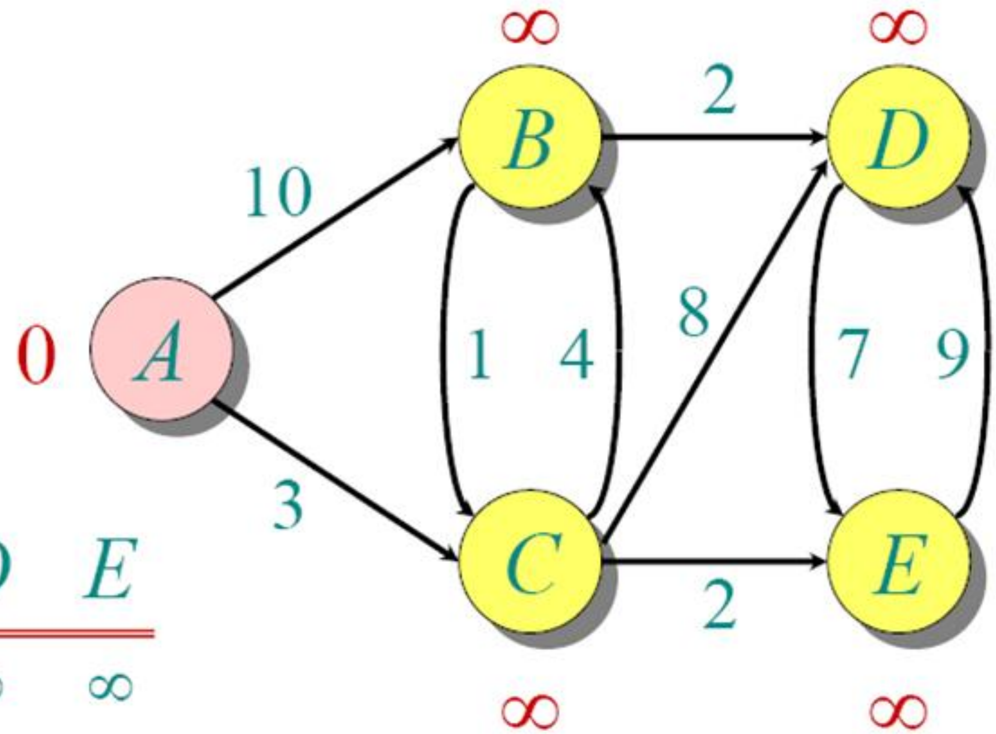
Distance(F) = min (8, 5+1) = 6

Example (end)



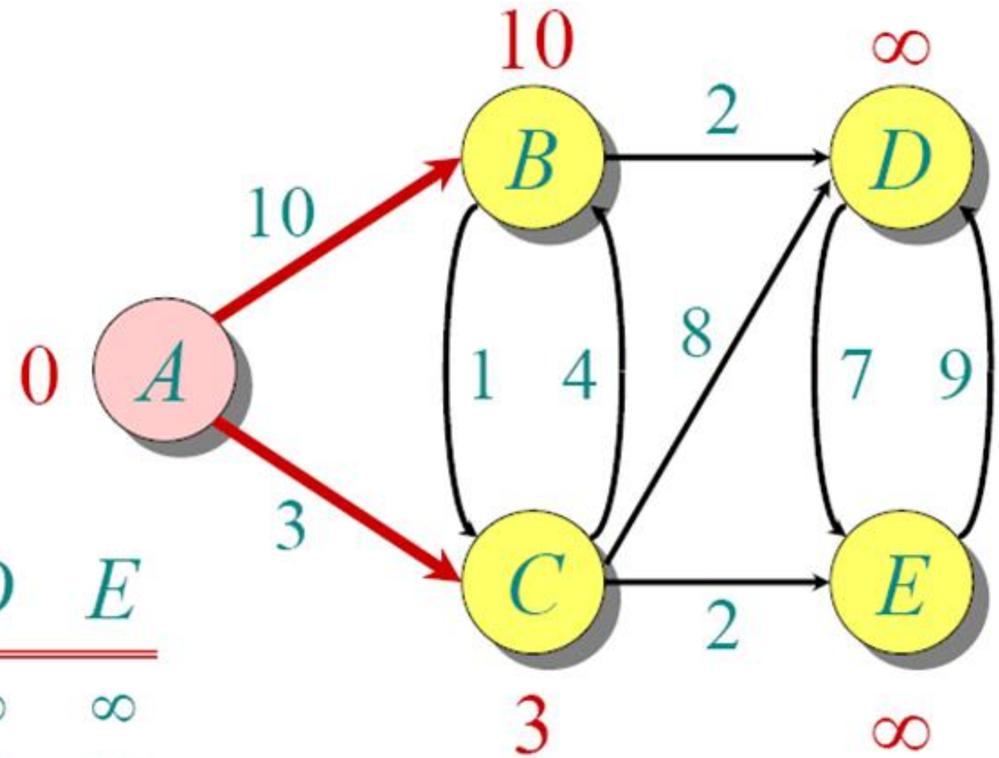
Pick vertex not in S with lowest cost (F) and update neighbors

Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

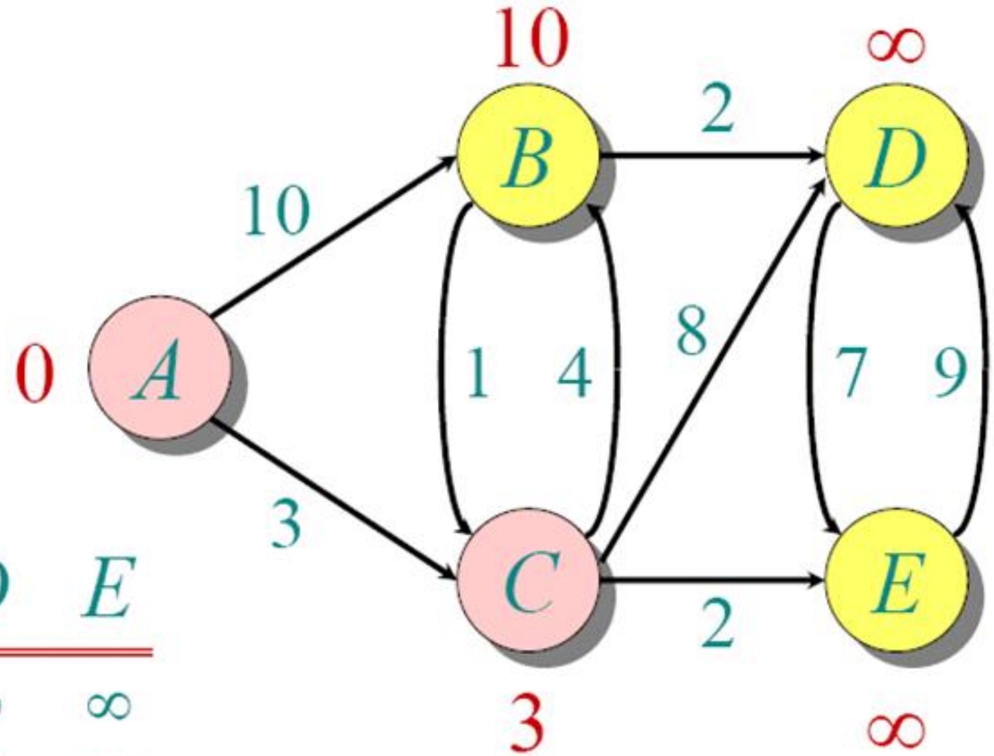
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

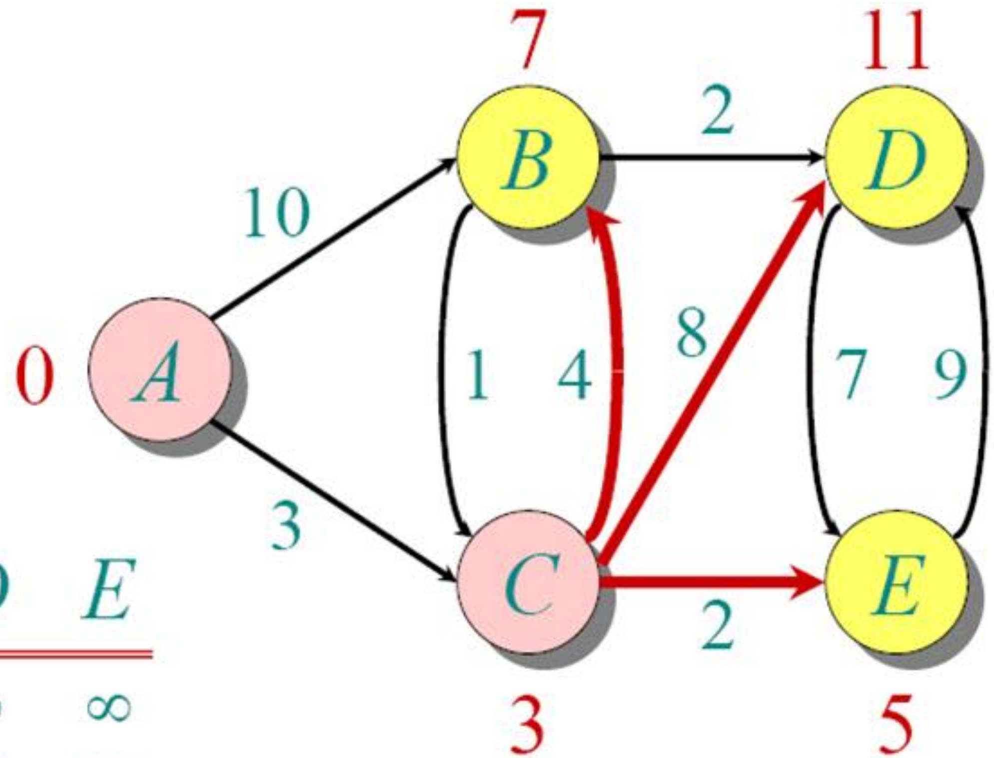
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

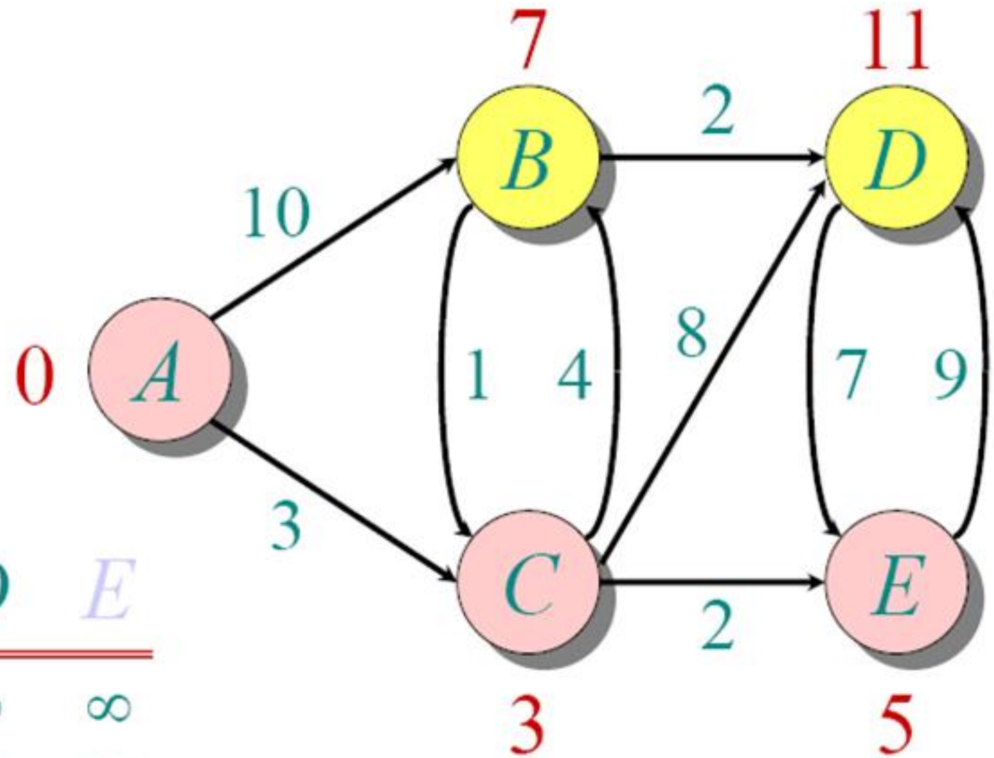
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

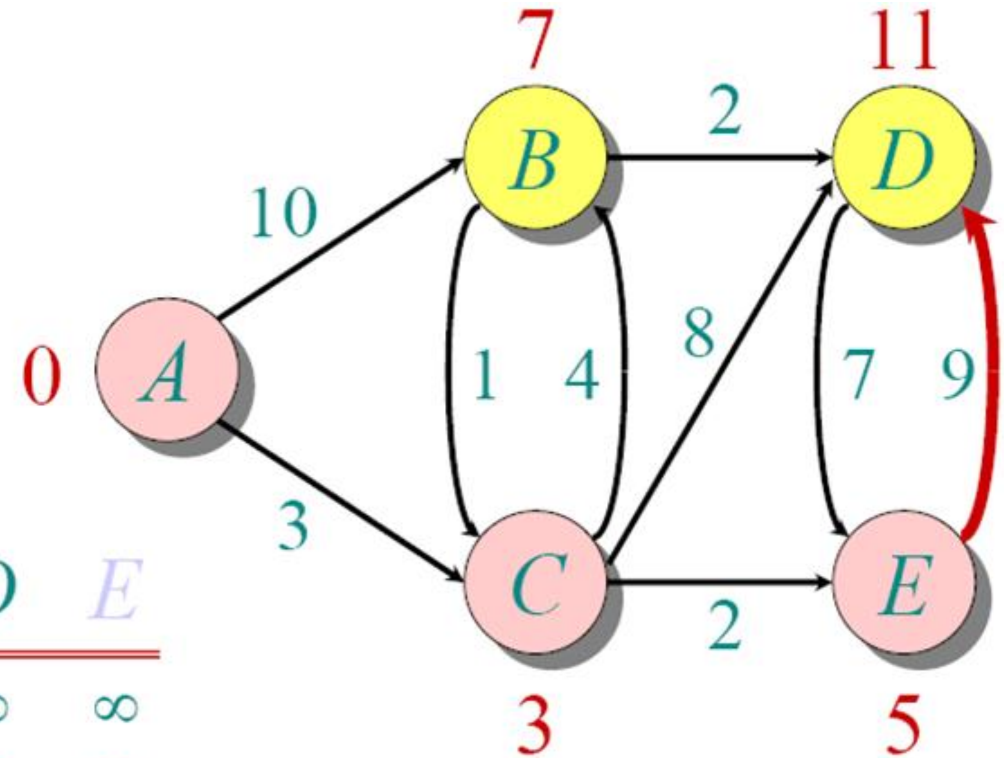
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

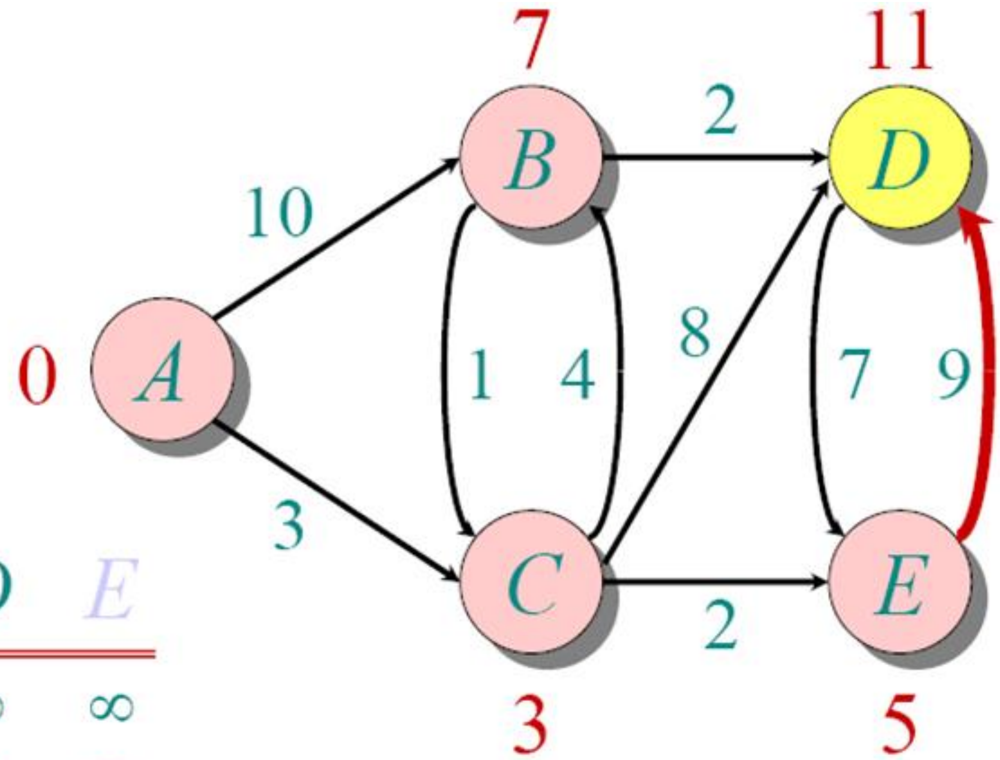
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

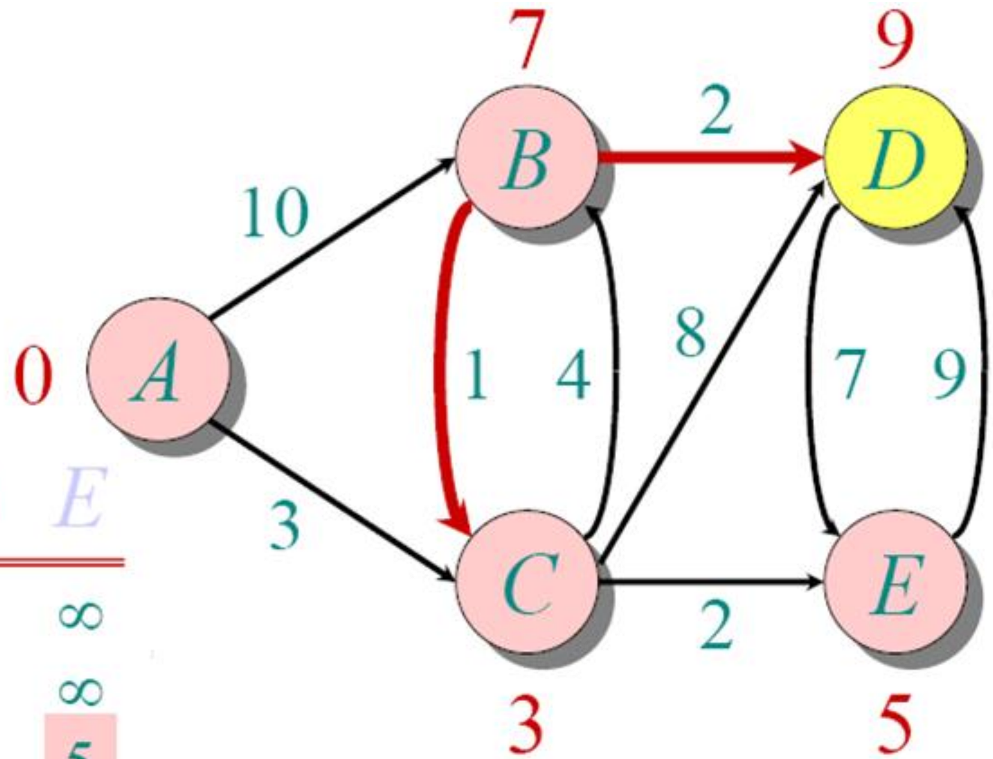
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

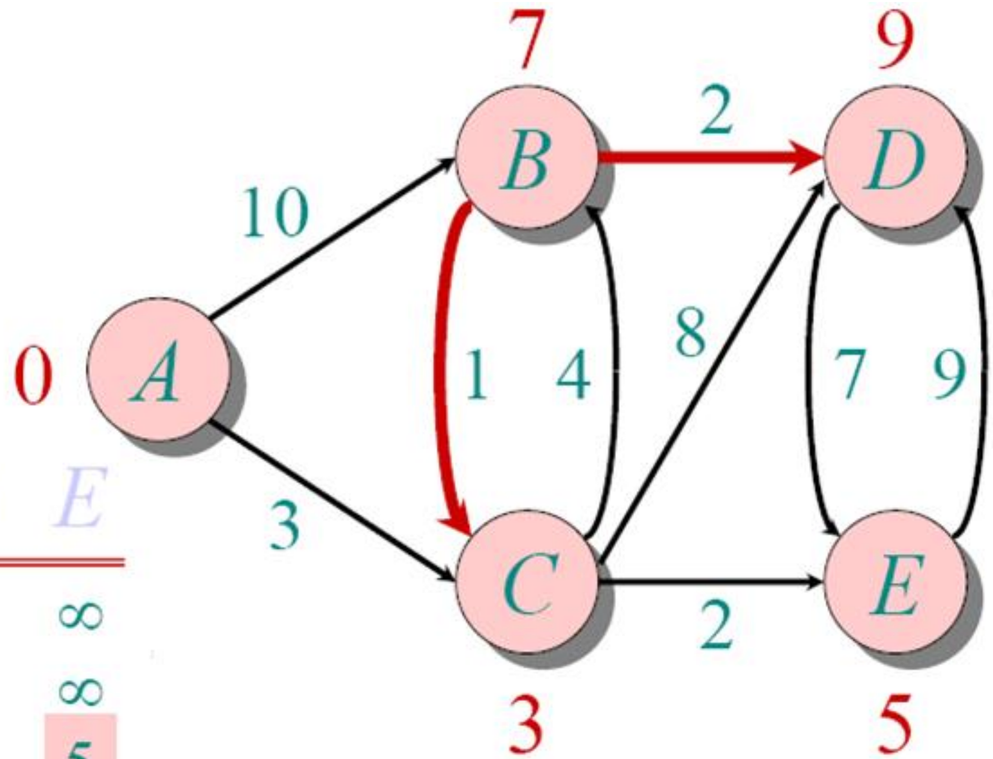
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Another Example



Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

S: { A, C, E, B, D }

- ***Insert lists and their lengths in a minimum heap of lengths.***
- ***Repeat***
  - ***Remove the two lowest length lists (p_i ,p_j) from heap.***
  - ***Merge lists with lengths (p_i,p_j) to form a new list with length $p_{ij} = p_i + p_j$***
  - ***Insert p_{ij} and its symbols into the heap***

until all symbols are merged into one final list

C	10				
B	25	A	30		
A	30	BC	35	BCA	65

- Notice that both Lists (B : 25 elements) and (C : 10 elements) have been merged (moved) twice
- List (A : 30 elements) has been merged (moved) only once.
- Hence the total number of element moves is 100.
- This is optimal among the other merge patterns.

Optimal Merge Pattern

- Merge a set of sorted files of different length into a single sorted file.
- To find an optimal solution, where the resultant file will be generated in minimum time.
- If the number of sorted files are given, there are many ways to merge them into a single sorted file.
- This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.
- To merge a p-record file and a q-record file requires possibly p + q record moves.

Optimal Merge Pattern

- For example
- x1=20, x2=40 and x3=15
- Merging can be done in 3 ways
  - (X1,x2),x3)
  - (x1,(x2,x3))
  - ((x1,x3),x2)
- Problem: find optimal merge cost

(x1,x2),x3)
A= (40+20)= 60
B= (A+x3)=
60+15=75
OM= 60+75 = 135

(x1,(x2,x3)
A= (40+15)= 55
B= (A+x1)=
55+20=75
OM= 55+75 = 130

(x1,x3),x2)
A= (20+15)= 35
B= (A+x2)=
35+40=75
OM= 35+75 = 110

Optimal Merge Pattern

- For i = 1 to n-1

  call Getnode(T) //allocate a new root

  LChild(T) ← least (L) // find a tree with least root weight as left child of the new root, then delete the tree from list

  RChild(T) ← least(L) // find a tree with least root weight as right child of the new root, then delete the tree from list

  Weight (T) ← Weight (LChild(T) + Weight(RChild(T))

  call Insert (L, T) // insert the tree with new root to the list

  repeat // n-1 time until there is only one tree

  return (Least(L))

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

Algorithm Tree(n)
// list is a global list of n single node
// binary trees as described above.
{
    for i := 1 to n − 1 do
    {
        pt := new treenode; // Get a new tree node.
        (pt → lchild) := Least(list); // Merge two trees with
        (pt → rchild) := Least(list); // smallest lengths.
        (pt → weight) := ((pt → lchild) → weight)
                +((pt → rchild) → weight);
        Insert(list, pt);
    }
    return Least(list); // Tree left in list is the merge tree.
}
```

Optimal Merge Pattern

- the data structure has to support delete-min and insert. Clearly, a min-heap is ideal.

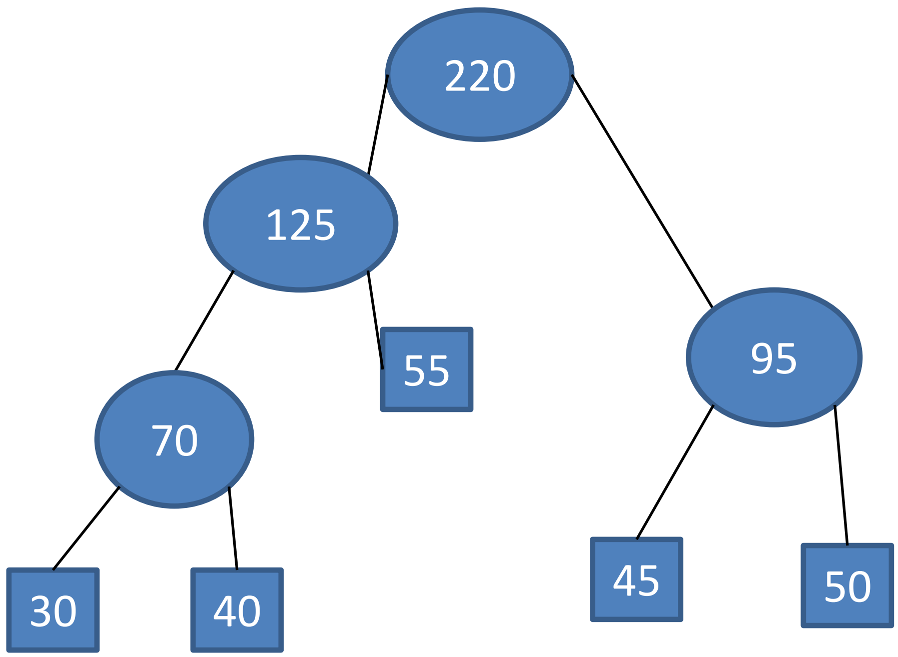
- Time complexity of the algorithm: The algorithm iterates (n-1) times. At every iteration two delete-mins and one insert is performed. The 3 operations take O(log n) in each iteration.

- Thus the total time is O(nlog n) for the while loop + O(n) for initial heap construction.

- That is, the total time is O(nlog n).

Optimal Merge Pattern

- Example [40, 30, 50, 45, 55]
- First step → Sort on ascending order of records



- Total Merge = 70+125+220+95 = 510
- Optimal Merge pattern = (((x1,x2)x5)(x3,x4))

Optimal Merge Pattern

- Example [20, 30, 10, 5, 40]

Huffman coding

- ***Fixed-length encoding***
  - ***ASCII, Unicode***

- ***Variable-length encoding : assign longer codewords to less frequent characters, shorter codewords to more frequent characters.***

Fixed-Length encoding

- **Fixed-Length encoding** - Every character is assigned a binary code using same number of bits. Thus, a string like “aabacdad” can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

Variable- Length encoding

- **Variable- Length encoding** - this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text.
- Thus, for a given string like “aabacdad”, frequency of characters ‘a’, ‘b’, ‘c’ and ‘d’ is 4,1,1 and 2 respectively.
- Since ‘a’ occurs more frequently than ‘b’, ‘c’ and ‘d’, it uses least number of bits, followed by ‘d’, ‘b’ and ‘c’.
- Suppose we randomly assign binary codes to each character as follows-

  a 0 b 011 c 111 d 11

  Thus, the string “aabacdad” gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),**

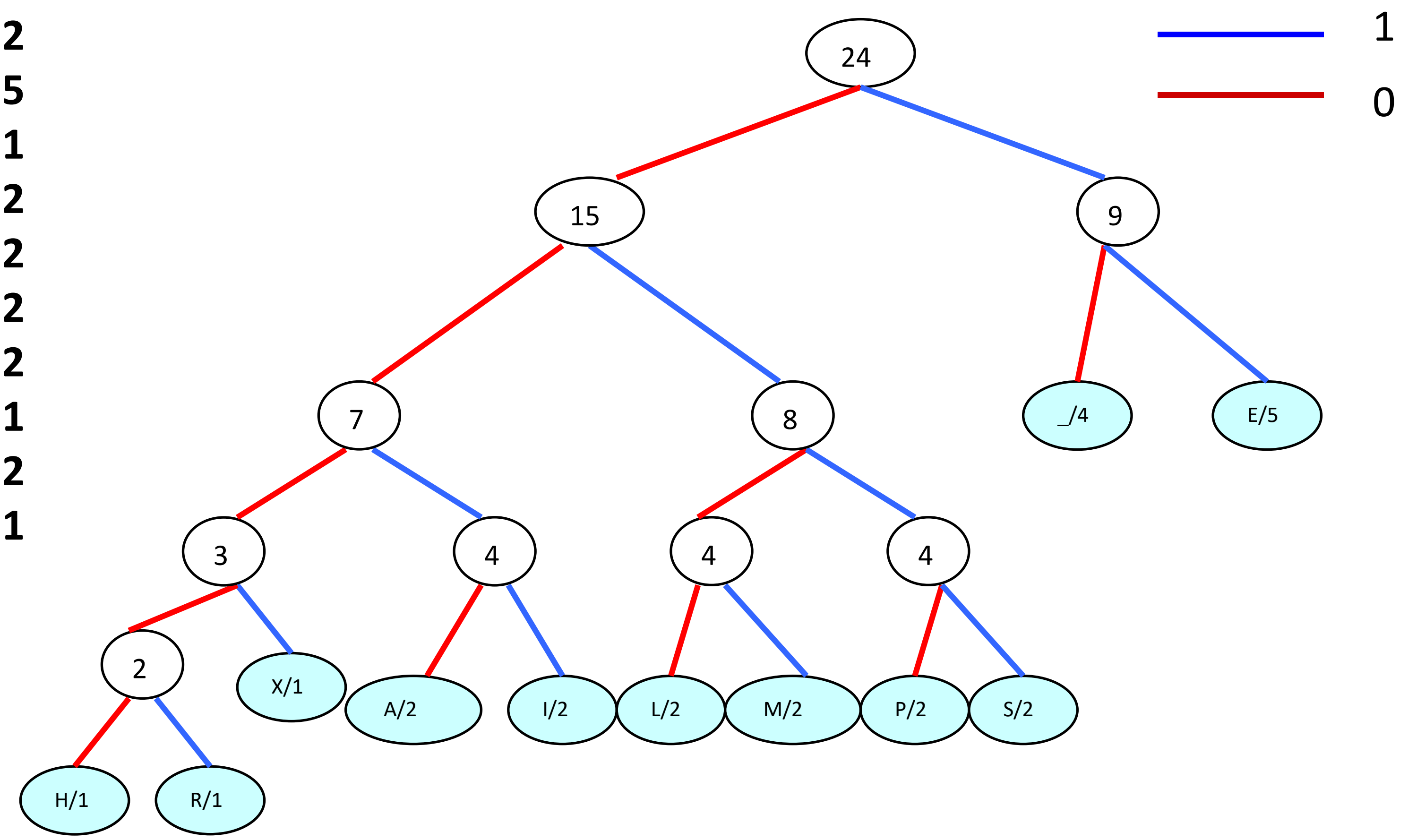
  using fewer number of bits compared to fixed-length encoding scheme.
- But with the decoding phase.
- decode the string 00011011111011,
- it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-
- **aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11) aaadbcad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11)**

1. **Compute the frequencies** of each character in the alphabet
2. **Build a tree forest with one-node trees**, where each node corresponds to a character and contains the frequency of the character in the text to be encoded
3. **Select two** parentless nodes with the **lowest frequency**
4. **Create a new node** which is the parent of the two lowest frequency nodes.
5. **Label** the **left link with 0** and the **right link with 1**
6. **Assign** the new node a **frequency equal to the sum** of its children's frequencies.
7. **Repeat Steps 3 through 6** until there is only one parentless node left.

here is a simple example

Example

_ (space) 4
A 2
E 5
H 1
I 2
L 2
M 2
P 2
R 1
S 2
X 1



The code for each symbol may be obtained by tracing a path from the root of the tree to that symbol.

Implementation

- ***Array frequencies[0...2N] : node frequencies.***
  - ***if frequencies[k] > 0, 0 $\leq$ k $\leq$ N-1, then frequencies[k] is a terminal node***
- ***Array parents[0..2N] : represents the parents of each node in array frequencies[].***
  - ***The parent of node k with frequency frequencies[k] is given by abs(parents[k]).***
  - ***If parents[k] > 0, node k is linked to the left of its parent, otherwise – to the right.***
- ***Priority queue with elements (k, frequencies[k]), where the priority is frequencies[k].***

Algorithm

- compute frequencies[k] and insert in a PQueue if frequencies[k] > 0
- m ← N
- while PQueue not empty do
  - deleteMin from PQueue (node1, frequency1)
  - if PQueue empty break
  - else deleteMin from PQueue (node2, frequency2)
  - create new node m and insert in PQueue
  - m ← m + 1
- end // tree is built with root = node1

Algorithm

Create New Node m

- **frequencies[m] ← frequency1 + frequency2 // new node**
- **frequencies[node1] ← m // left link**
- **frequencies[node2] ← -m // right link**
- **insert in PQueue**

  (m, frequency1 + frequency2)

To Encode a Character

Start with the leaf corresponding to that character and follow the path to the root

The labels on the links in the path will give the reversed code.

To Restore the Encoded Text

- Start with the root and follow the path that matches the bits in the encoded text, i.e. go left if ‘0’, go right if ‘1’.
- Output the character found in the leaf at the end of the path.
- Repeat for the remaining bits in the encoded text

File :

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12

Analysis

Time efficiency of building the Huffman tree

The insert and delete operations each take log(N) time, and they are repeated at most 2N times

Therefore the run time is O(2NlogN) = O(NlogN)