OntarioTech Science

Kourosh Davoudi
kourosh@ontariotechu.ca

Lecture 12: Review

CSCI 3070U: Design and Analysis of Algorithms

Learning Outcomes

- Algorithm and Time Complexity
- Divide and Conquer
- Dynamic Programming
- Greedy Algorithm
- Branch and Bound
- Sort Algorithms
- Graph Algorithms

Learning Outcomes

- Algorithm and Time Complexity (Analysis)
- Divide and Conquer (Designing Method)
- Dynamic Programming (Design Method)
- Greedy Algorithm (Design Method)
- Branch and Bound (Design Method)
- Sort Algorithms (Important Class of Algorithm)
- Graph Algorithms (Important Class of Algorithm)

Algorithms and Time Complexity

Case Study: Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

t_j : The number of times the while loop test in line 5 is executed for that value of j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Case Study: Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

- Best Case:

$$t_j = 1$$

- Worst Case:

$$t_j = j$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Case Study: Fibonacci Series

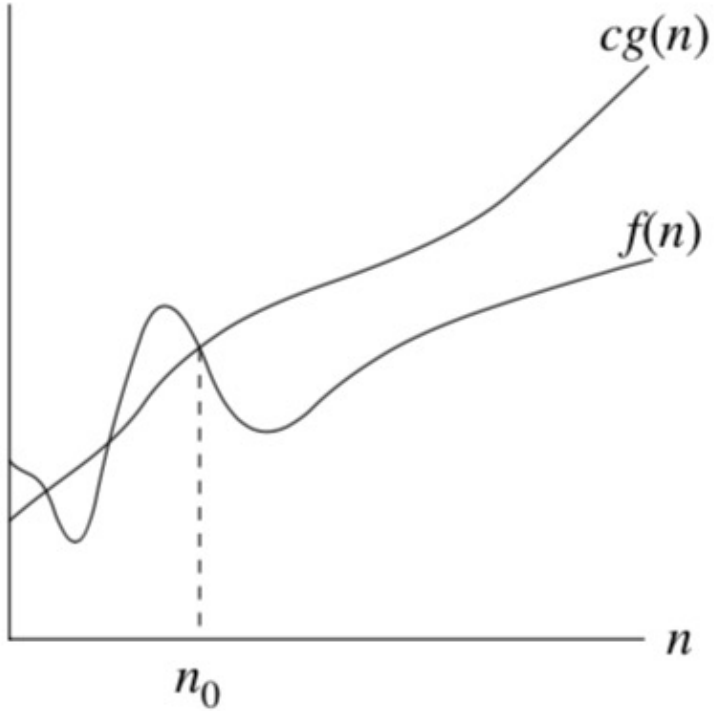
```
FIB(n)
1   if n == 0 or n == 1
2         return 1
3   else
4         return FIB(n − 1) + FIB(n − 2)
```

$$T(0) = c_1$$
$$T(1) = c_2$$
$$T(n) = T(n-1) + T(n-2) + c_3$$

Asymptotic Notations

Asymptotic Notations

- O - notation



$$f(n) \in O(g(n))$$

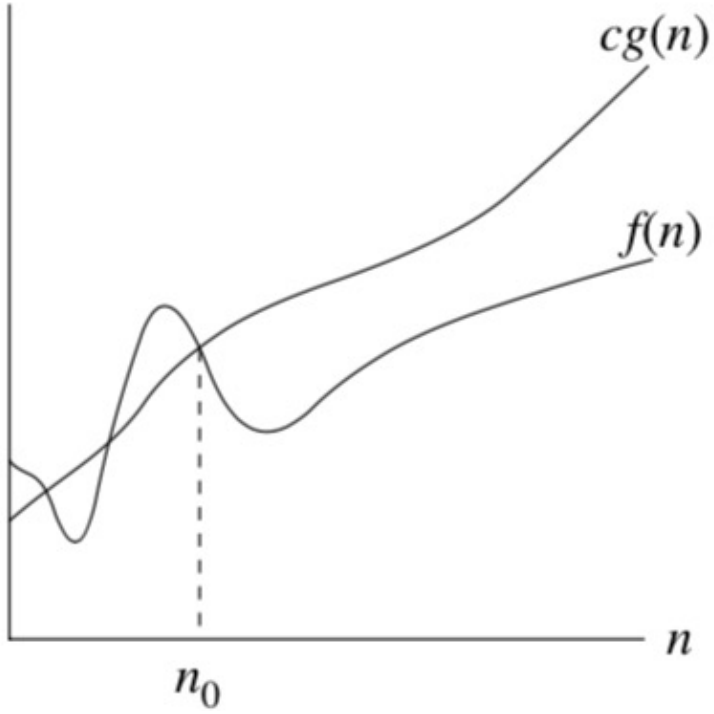
or

$$f(n) = O(g(n))$$

$$O(g(n)) = \{ f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$

Asymptotic Notations

- O - notation



$$f(n) \in O(g(n))$$

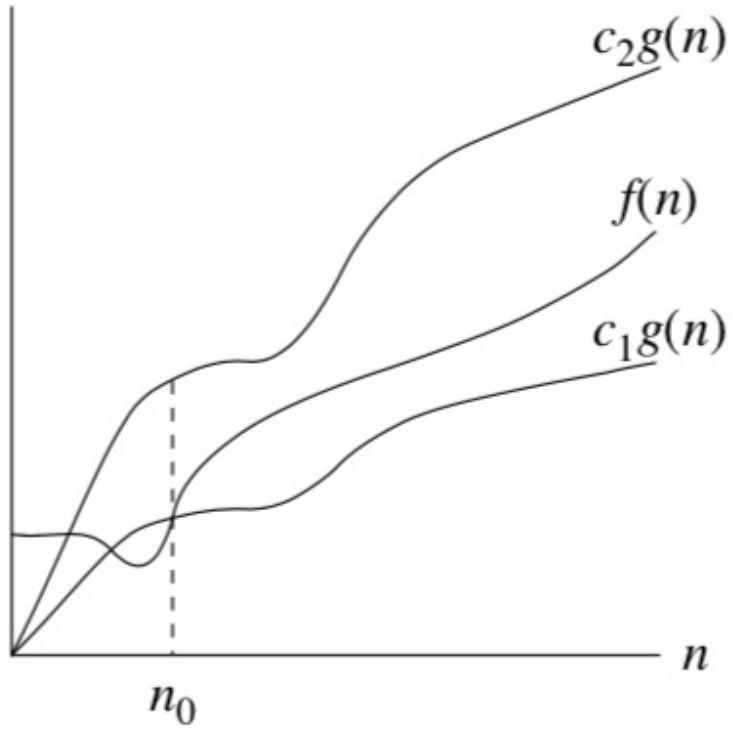
or

$$f(n) = O(g(n))$$

$$O(g(n)) = \{ f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \} .$$

Asymptotic Notations

- Θ - notation



$$f(n) \in \Theta(g(n))$$

or

$$f(n) = \Theta(g(n))$$

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

Asymptotic Notations

- Θ - notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem

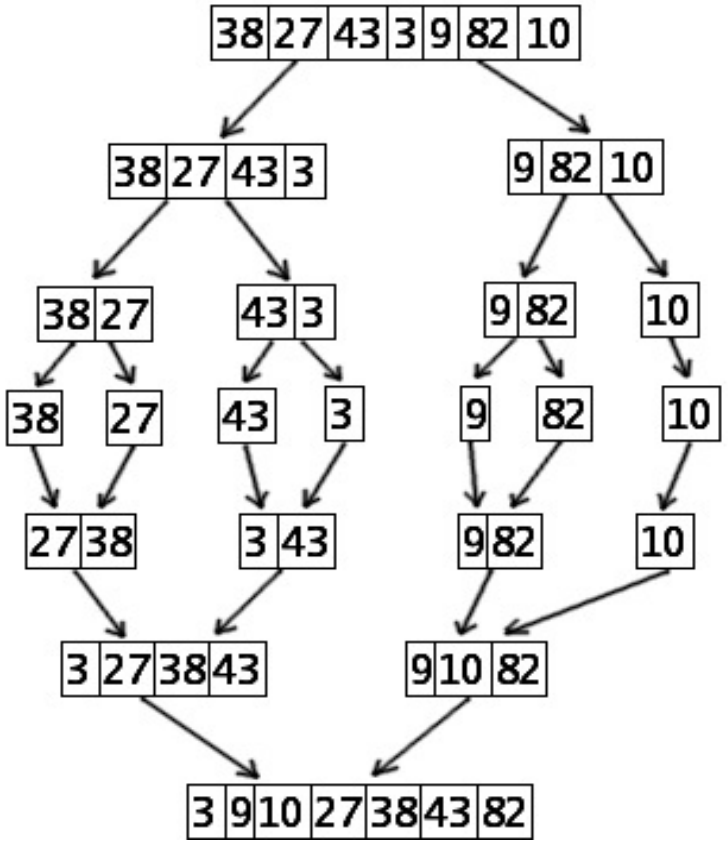
$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$

$$f(n) = \Theta(g(n))$$

Divide and Conquer

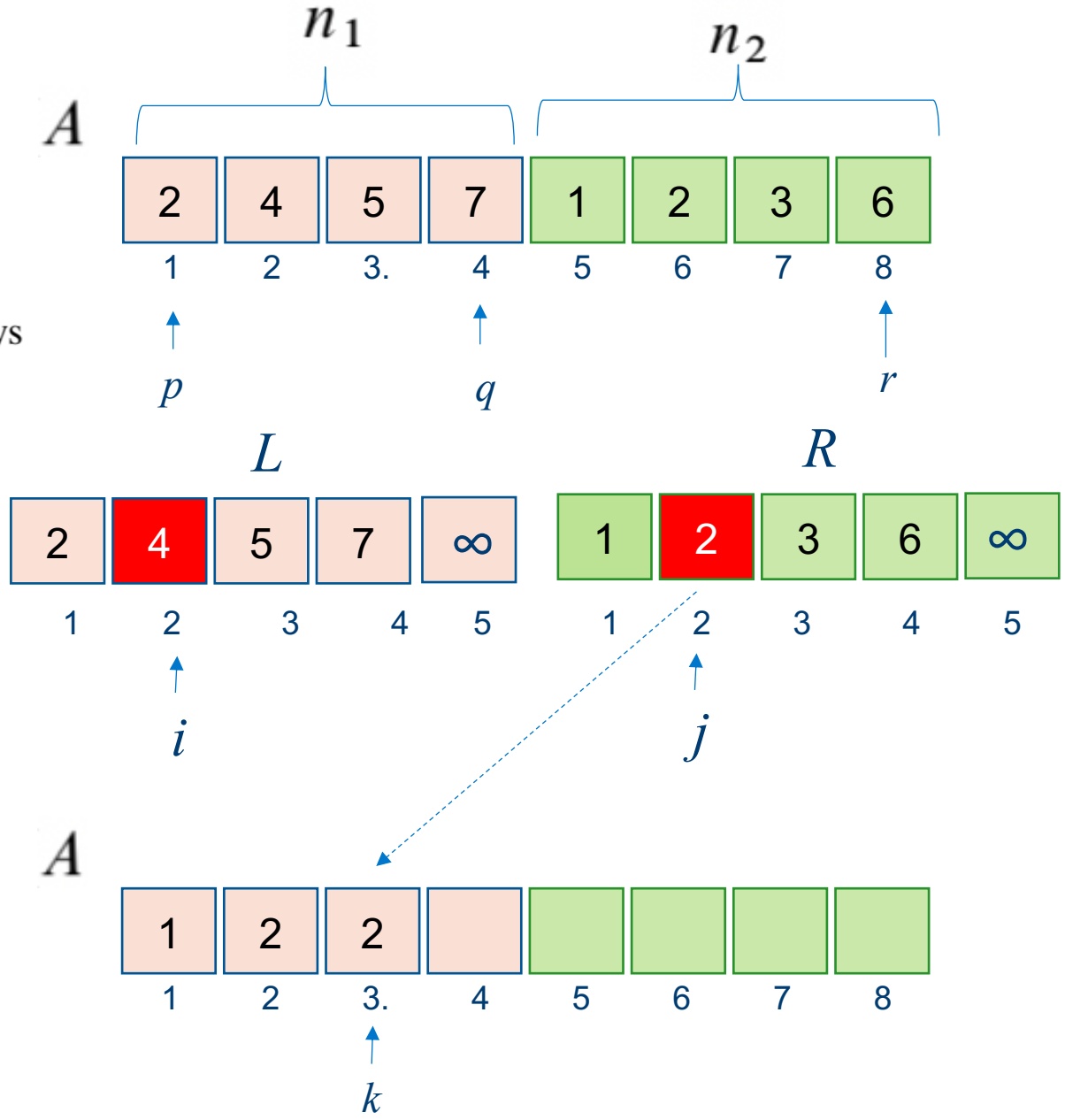
Case Study: Merge Sort

```
MERGE-SORT(A, p, r)
  if p < r                          // check for base case
      q = ⌊(p + r)/2⌋               // divide
      MERGE-SORT(A, p, q)           // conquer
      MERGE-SORT(A, q + 1, r)       // conquer
      MERGE(A, p, q, r)             // combine
```



Case Study: Merge Sort

```
MERGE(A, p, q, r)
  n1 = q − p + 1
  n2 = r − q
  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
  for i = 1 to n1                                   Θ(n1)
      L[i] = A[p + i − 1]
  for j = 1 to n2                                   Θ(n2)
      R[j] = A[q + j]                                          Θ(n)
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r                                    Θ(n)
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```



Merge Sort Time Complexity

```
MERGE-SORT(A, p, r)
 if p < r
     q = ⌊(p + r)/2⌋                    Θ(1)
     MERGE-SORT(A, p, q)                T(n/2)
     MERGE-SORT(A, q + 1, r)            T(n/2)
     MERGE(A, p, q, r)                  Θ(n)
```

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

Recurrence Equations

How to Solve Recurrence Equation?

- Substitution Method
  - Guess, then use induction to prove it
- Recursion Tree
  - Draw the recurrence as a tree and use geometry (i.e. tree height and width) to estimate total cost
- Master Theorem
  - Applies to recurrences of the form: $T(n) = aT(n/b) + f(n)$

Master Theorem

$$T(n) = aT(n/b) + f(n) \qquad \text{where } a \geq 1, b > 1, \text{ and } f(n) > 0$$

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)

Solution: $T(n) = \Theta(n^{\log_b a})$

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$

$af(n/b) \leq cf(n)$ for some constant $c < 1$

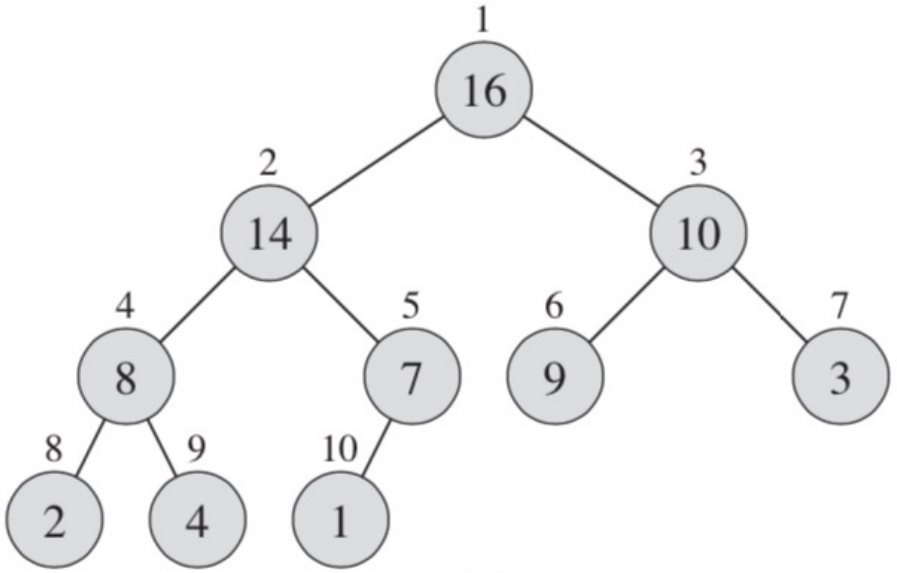
($f(n)$ is polynomially greater than $n^{\log_b a}$.)

Solution: $T(n) = \Theta(f(n))$

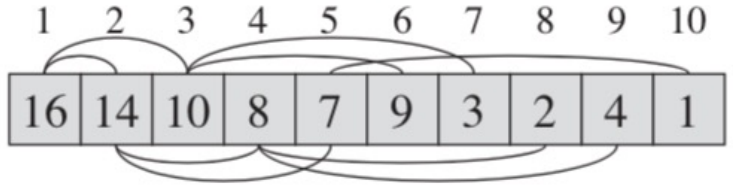
Heap

Data Structure for Heap Tree

- Complete binary tree can be easily stored in an array !



(a)



(b)

Basic Operations

A basic set of heap operations:

- MAX-HEAPIFY
- BUILD-MAX-HEAP
- HEAP-MAXIMUM
- HEAP-EXTRACT-MAX
- MAX-HEAP-INSERT

Heap Applications

- Heapsort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)                    O(n)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
```

$O(\log n)$

Time Complexity

$$O(n \log n)$$

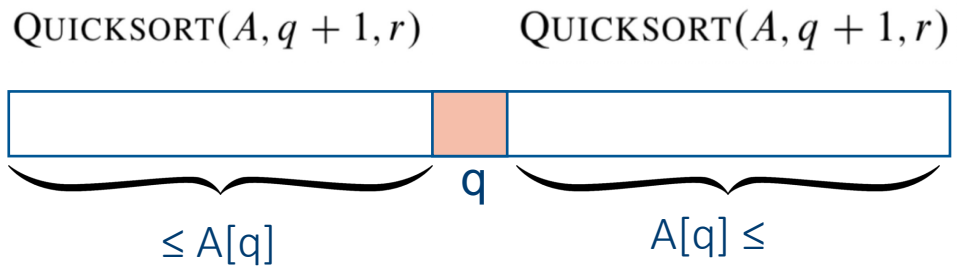
Priority Queue Implementations

	Unsorted Array	Sorted Array	Heap
INSERT(S, x)	$\Theta(1)$ Add into the end	$\Theta(n)$ Shift	$\Theta(\log n)$ MAX-HEAP-INSERT
MAXIMUM(S)	$\Theta(n)$ Linear Search	$\Theta(1)$ Last Element	$\Theta(1)$ HEAP-MAXIMUM
EXTRACT-MAX(S)	$\Theta(n)$ Search + shift	$\Theta(1)$ Shorten	$\Theta(\log n)$ HEAP-EXTRACT-MAX

Sort Algorithms

Quick Sort

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```



Partitioning in Quicksort

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

$\Theta(n)$

Lower Bounds for Sorting

- How fast can we sort?
  - All sorts seen so far are

$$\Omega(n \log n)$$

  - We'll show that $\Omega(n \log n)$ is a lower bound for comparison sorts.
  - We use decision trees for this purpose:
    - The important factor is number of comparison and decision tree help us track it !

Linear Time Sorting

- Comparison sorting lover bound: $n \log n$
- So, how is linear time sorting possible?
  - We don't use comparison between elements to sort
  - Linear sorting algorithms only work with numeric keys !

Counting Sort
Radix Sort
Bucket Sort

Counting Sort

- *Assumption*: numbers to be sorted are integers in

$$\{0, 1, \ldots, k\}$$

- Input: $A[1\,..\,n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$

- Output: $B[1\,..\,n]$, sorted. B is assumed to be already allocated and is given as a parameter

- Auxiliary storage: $C[0\,..\,k]$

Radix Sort Idea

- Key Ideas:
  - View each number as a multi-digit word.
  - Each digit can be arbitrary bits long.
  - Sort from the least significant digit to the most significant digit using any **stable** sorting algorithm.

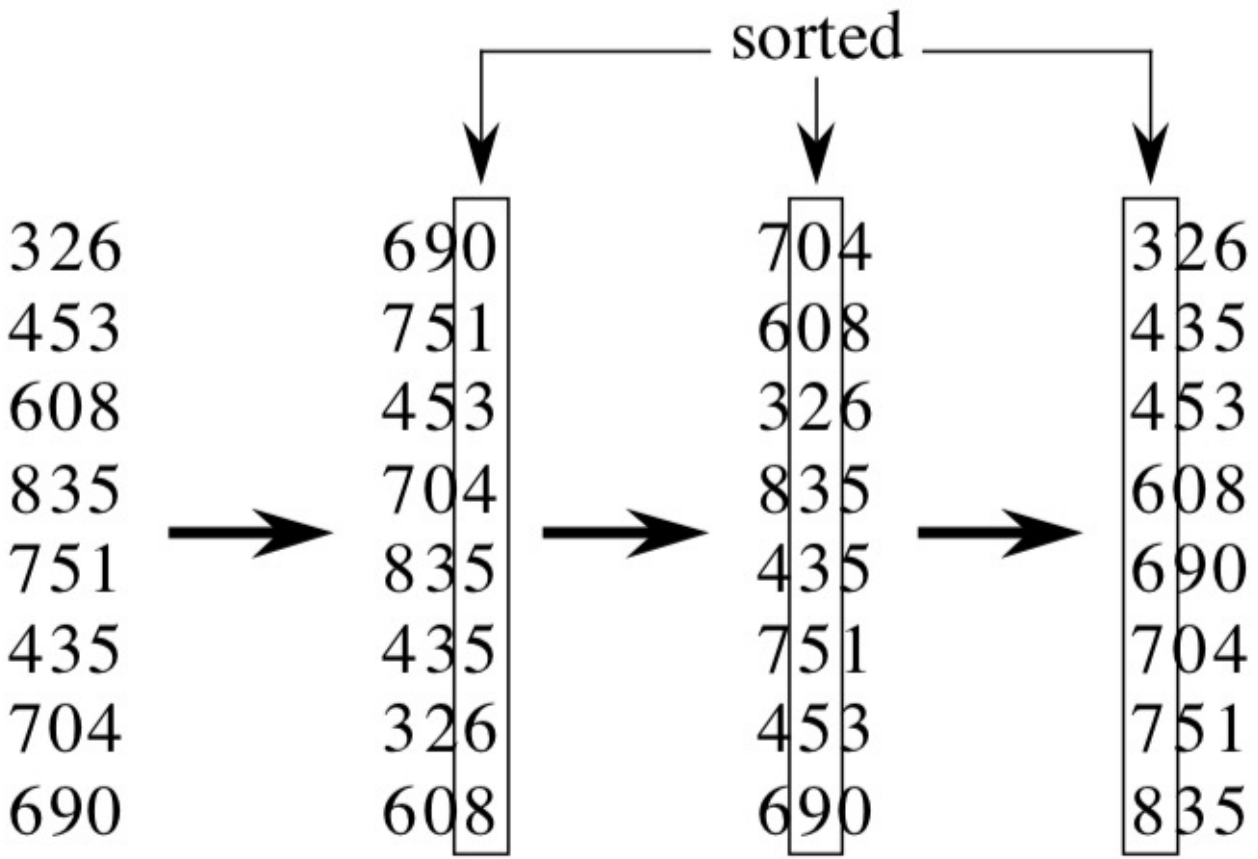
RADIX-SORT(A, d)

for $i = 1$ **to** d

use a stable sort to sort array A on digit i

Radix Sort

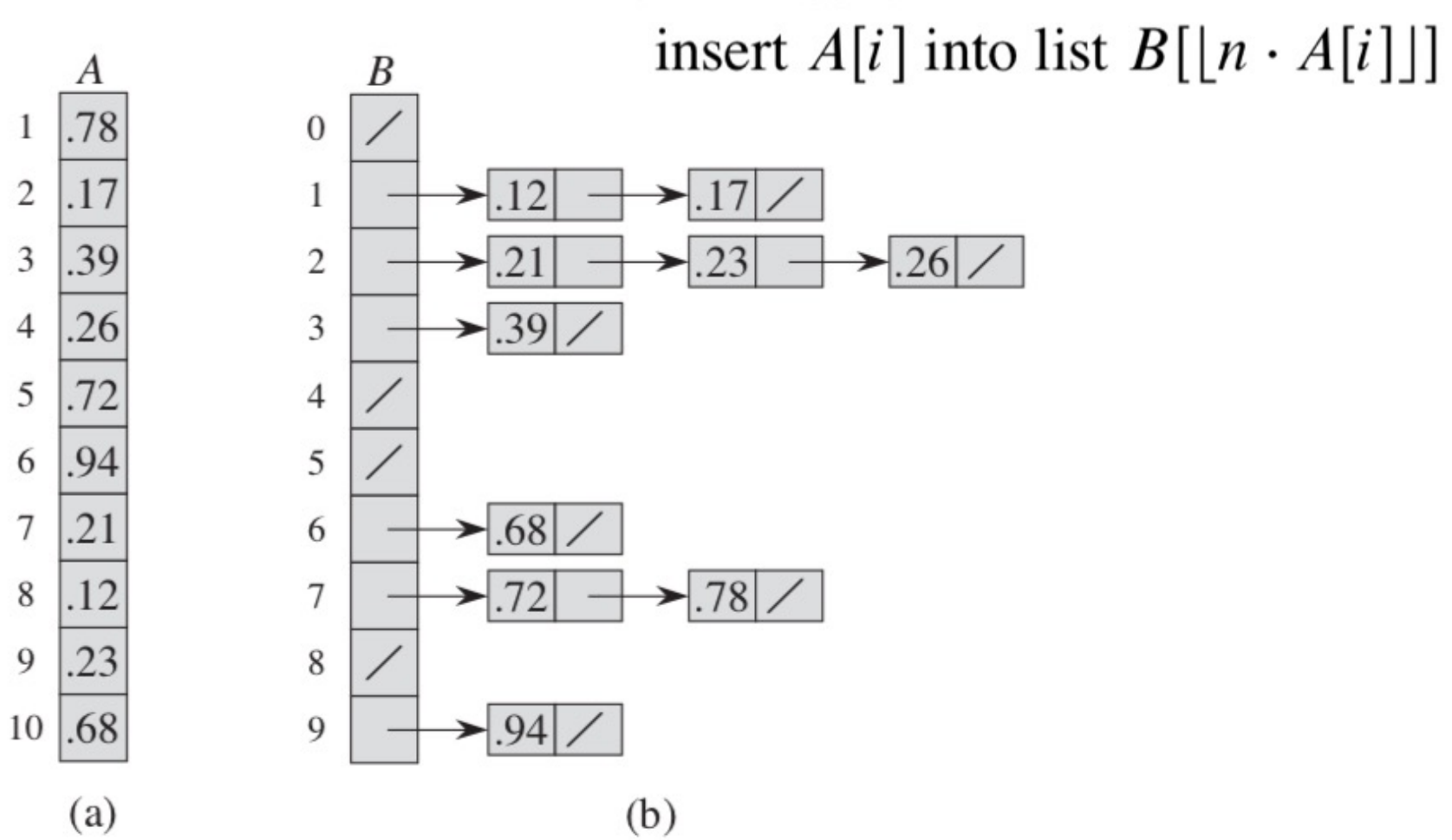
- Example:



Bucket Sort

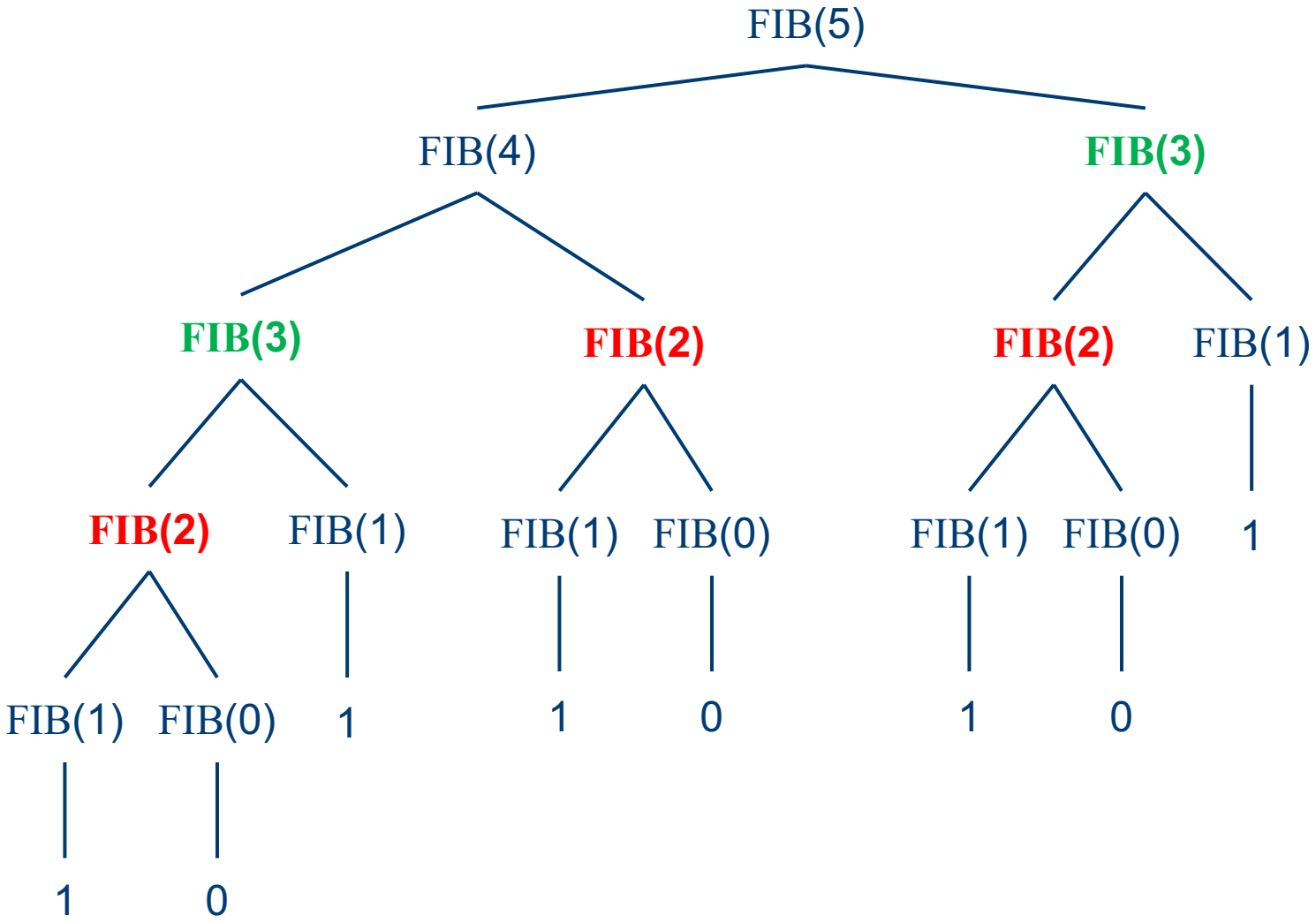
- *Assumption*: the input is generated by a random process that distributes elements uniformly over [0, 1)
- General Idea
  - Divide [0,1) into n equal-sized *buckets*
  - Distribute the n input values into the buckets
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one

Bucket Sort


insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94
(a)
(b)

Dynamic Programming

Motivation: Fibonacci Series



Case Study: Fibonacci Series

```
FIBONACCI(n)
  let fib[0..n] be a new array
  fib[0] = fib[1] = 1
  for i = 2 to n
      fib[i] = fib[i − 1] + fib[i − 2]
  return fib[n]
```

$$T(n) = \Theta(n)$$

Dynamic Programming Foundation

- Idea: Previously computed subproblem solutions are stored
  - Dynamic programming often involves a space-time trade-off
- Storing computed solutions is called memoization
- Each time a computation needs to occur, check
  - if it exists in our table, Yes: Use it
  - No: Compute it, and store it in the table

Problem Formulation

- We pick as our subproblems the problems of determining the minimum cost of parenthesizing

$$A_i A_{i+1} .. A_j \qquad 1 \le i \le j \le n$$

  - $A_i : p_{i\text{-}1} \times p_i$
- Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
- Our goal is to find

$$m[1,n]$$

Problem Formulation

- We pick as our subproblems the problems of determining the minimum cost of parenthesizing

$$A_i A_{i+1} .. A_j \qquad 1 \le i \le j \le n$$

  - $A_i : p_{i\text{-}1} \times p_i$
- Recursive Formulation:

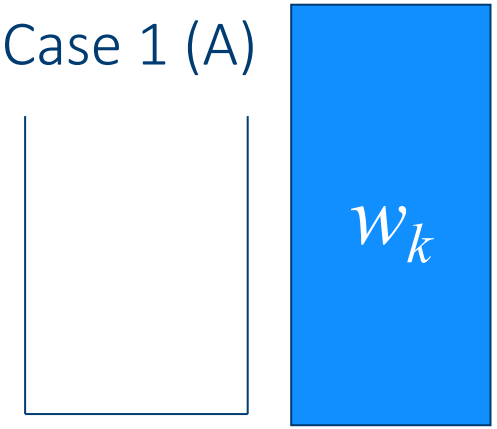
$$(A_i A_k)\,(A_{k+1} .. A_j)$$

$$p_{i\text{-}1} \times p_k \qquad p_k \times p_j$$

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

Knapsack Optimum Subproblem

Case 1 (A)

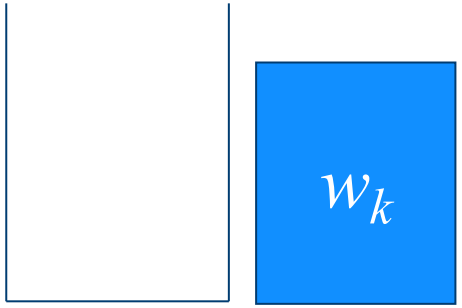


w

$w_k > w$

$B[k, w] = B[k-1, w]$

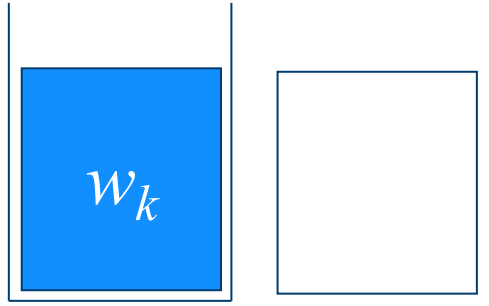
Case 1 (B)



w

$w_k \leq w$

Case 2



$w - w_k$

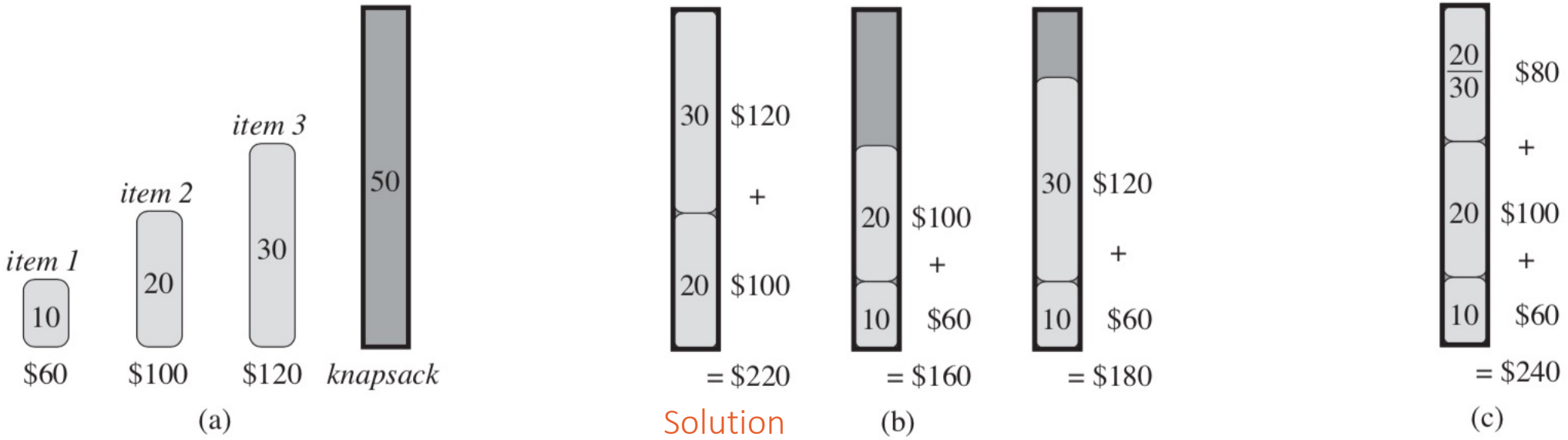
$w_k \leq w$

$B[k, w] = \max\{B[k-1, w], b_k + B[k-1, w - w_k]$

Assume $B[k, w]$ is a best value for the set S_k

Greedy Algorithms

Fractional vs. 0/1 Knapsack



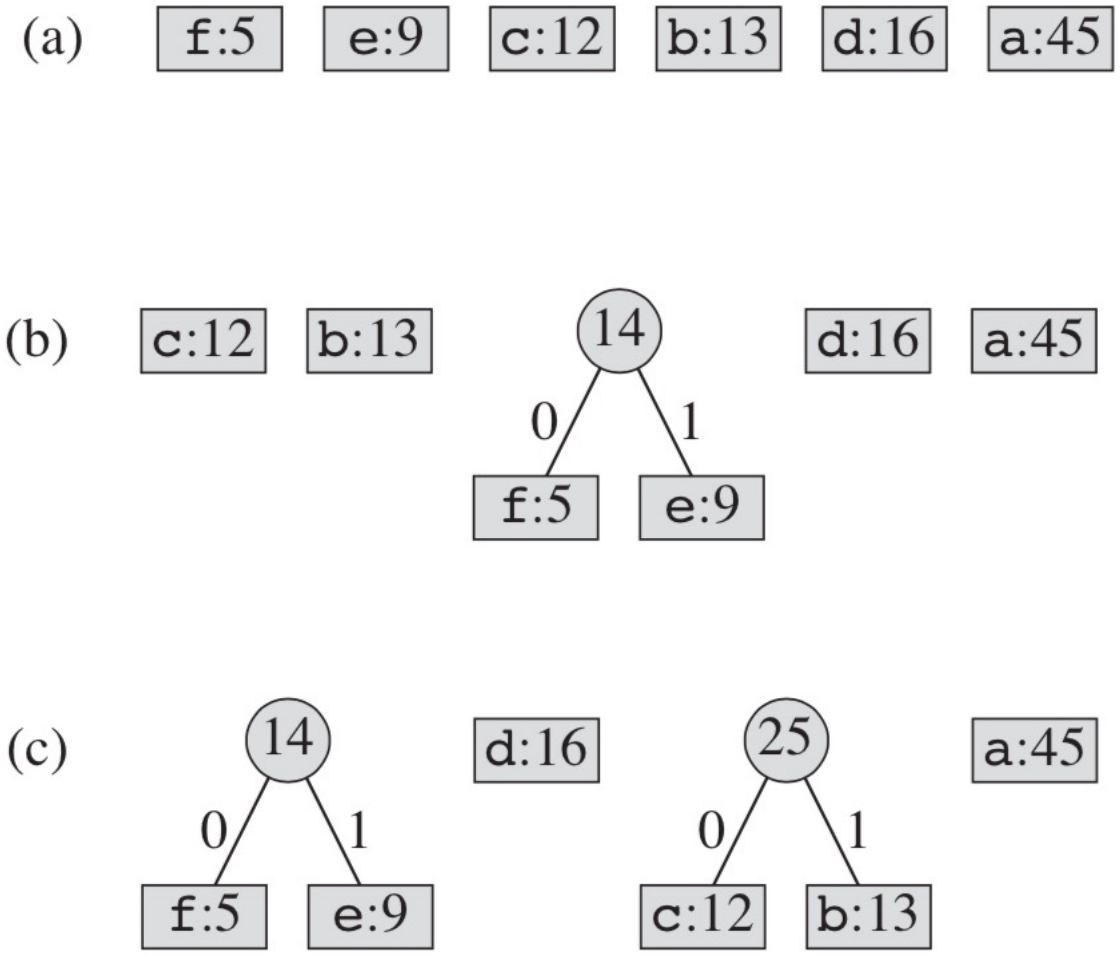
item 1: *60/10 =6*

item 2: *100/20 =5*

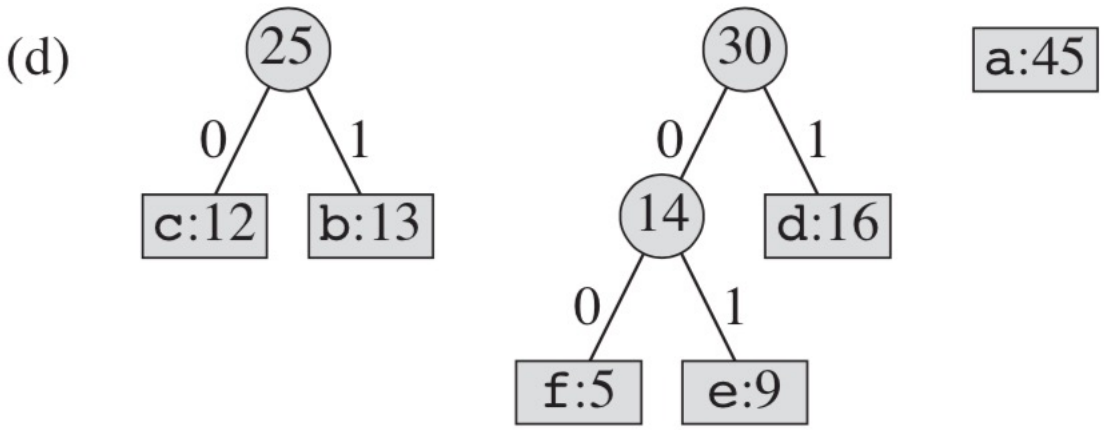
item 3: *120/30 =4*

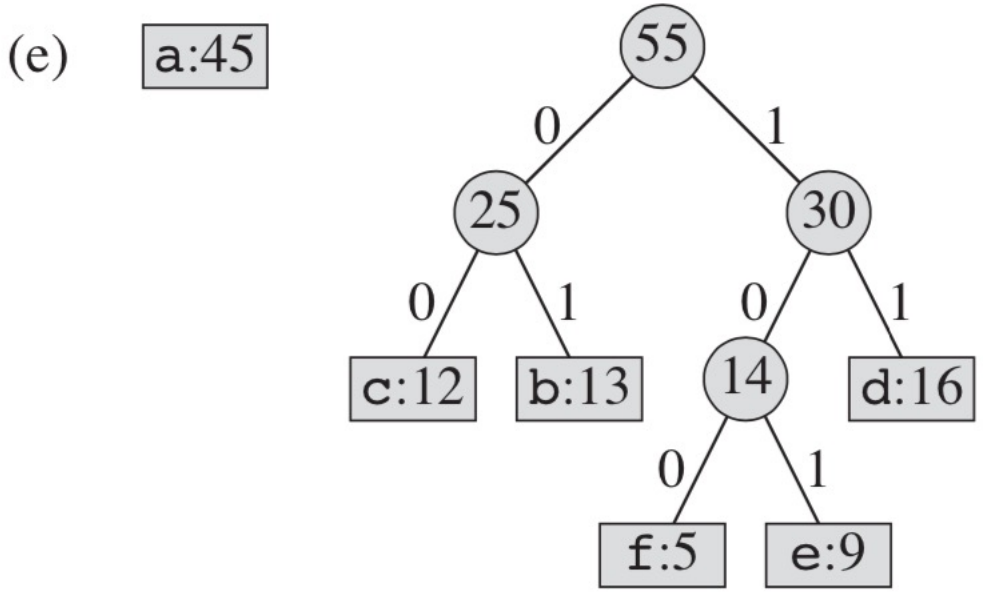
For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Huffman Code: Example



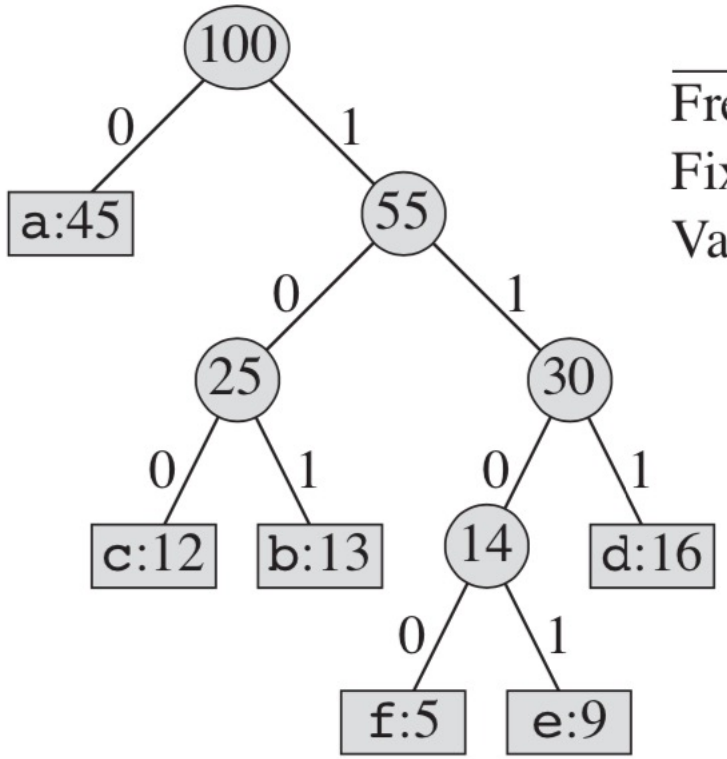
Huffman Code: Example





Huffman Code: Example

(f)



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Huffman's Algorithm

Time Complexity:

$O(n \log n)$

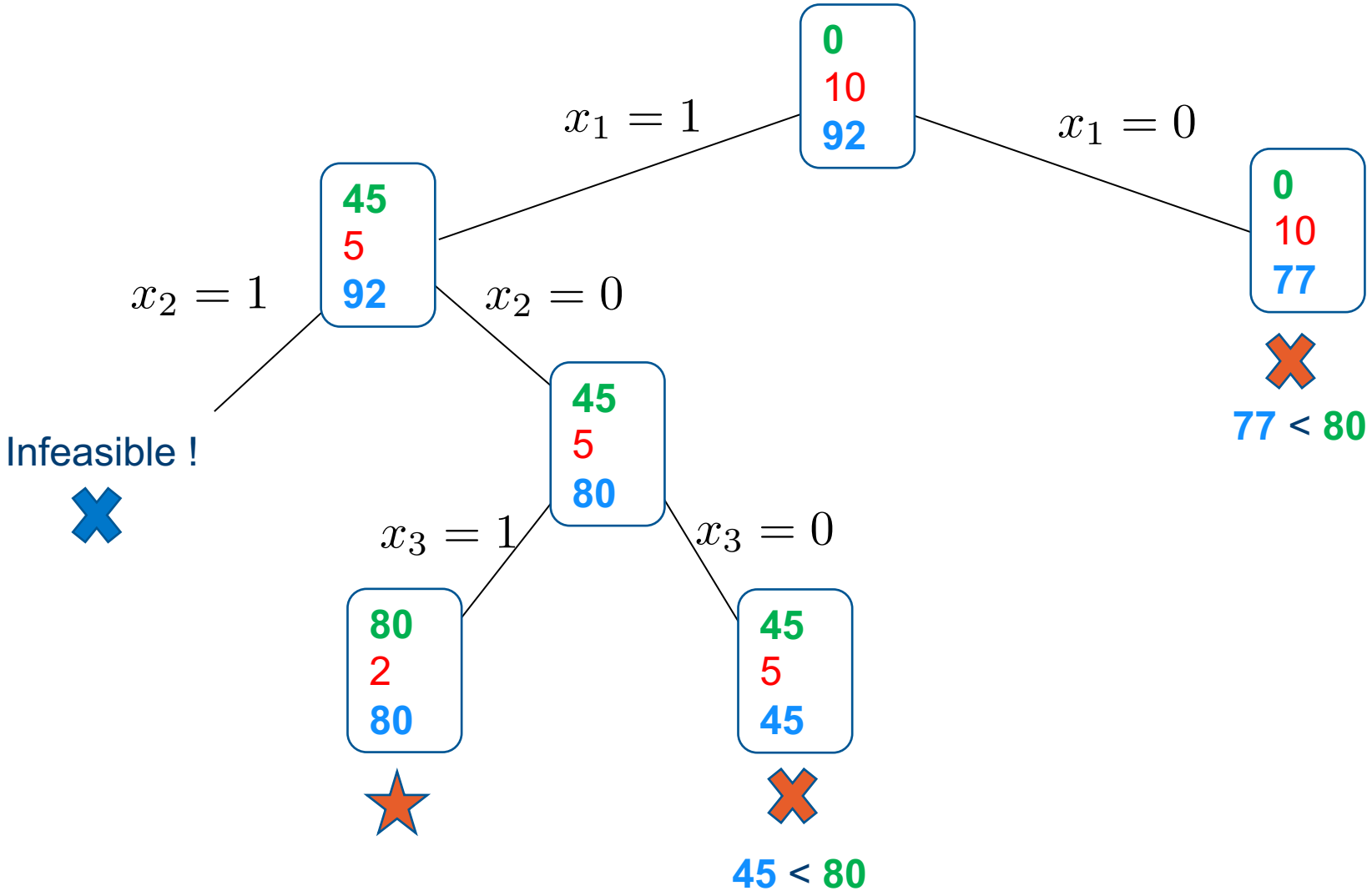
```
HUFFMAN(C)
1  n = |C|
2  Q = C                                              O(n)
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)                    O(log n)
6      z.right = y = EXTRACT-MIN(Q)                   O(log n)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)                                   O(log n)
9  return EXTRACT-MIN(Q)      // return the root of the tree   O(log n)
```

Branch and Bound

Example 2: Tighter upper bound

item	value	Weight
1	45	5
2	48	8
3	35	3

W = 10



Heuristic upper bound : sum of value of remining items based on fractional knapsack

Graph Search

BFS Algorithm

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

$v.d$: distance from vertex v to the source s

$v.\pi$: predecessor of vertex v along the shortest path from source s.

$Adj[u]$: list of neighbours of vertex u.

$$\Theta(V + E)$$

DFS Algorithm

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

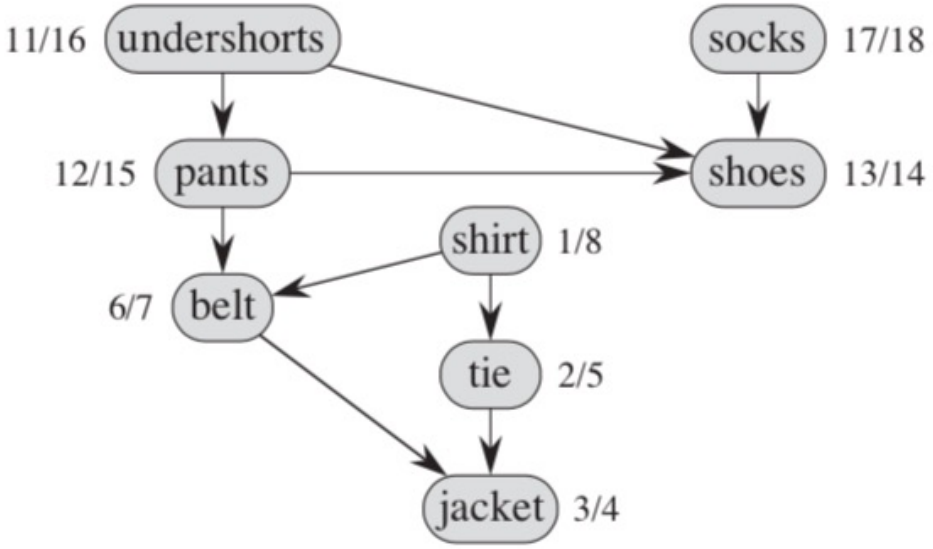
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

$$\Theta(V + E)$$

Graph Search Applications

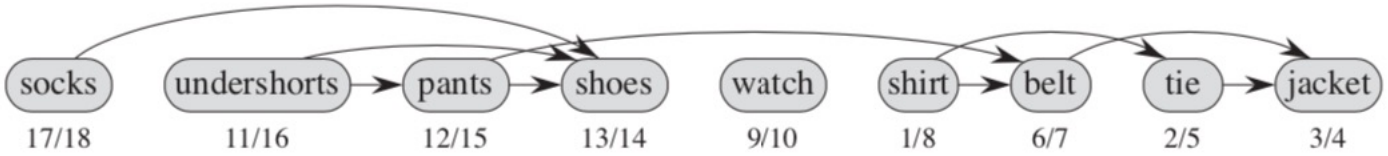
Topological Sort

- Topological sort:
  - Given a DAG, a linear ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v.



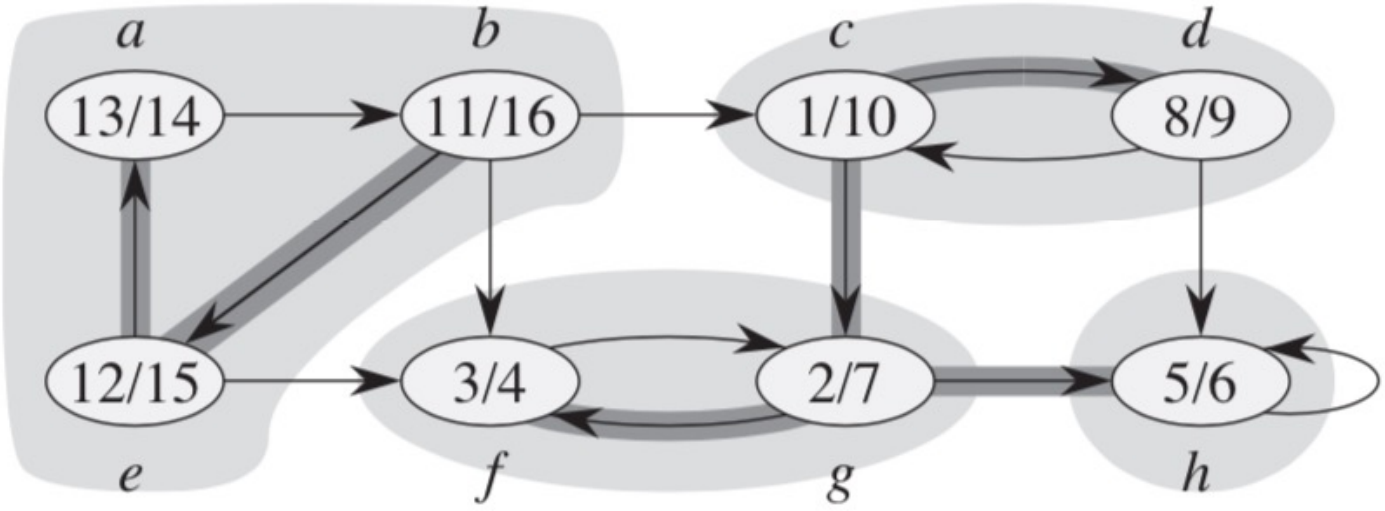
$\Theta(V + E)$

Vertices are in order of decreasing finishing times !



Strongly Connected Components

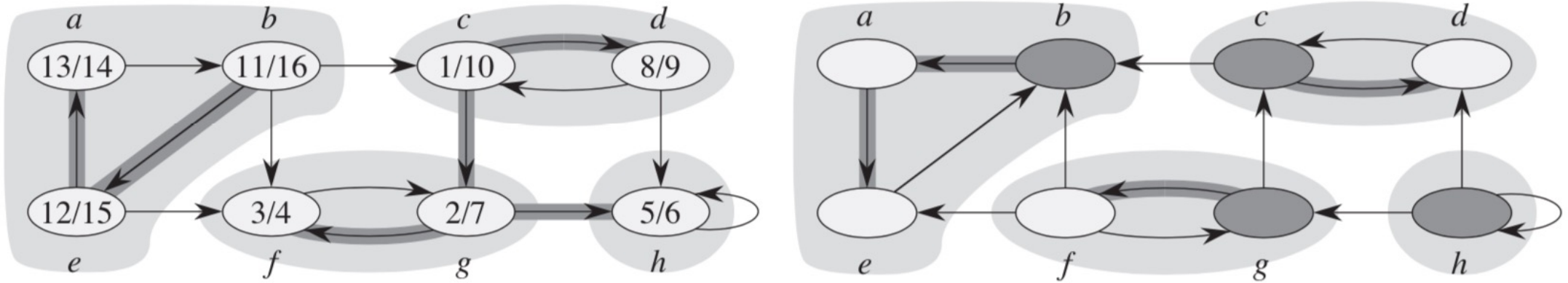
- Problem: How can we identify the Strongly Connected Components (SCC)?



We can use DFS to identify SCC

Strongly Connected Components

- The algorithm uses transpose graph $G^T = (V, E^T)$



$G = (V, E)$ $G^T = (V, E^T)$

Graph G and G^T have exactly the same strongly connected components

Given an adjacency-list representation of G, the time to create G^T is $O(V + E)$

Strongly Connected Components

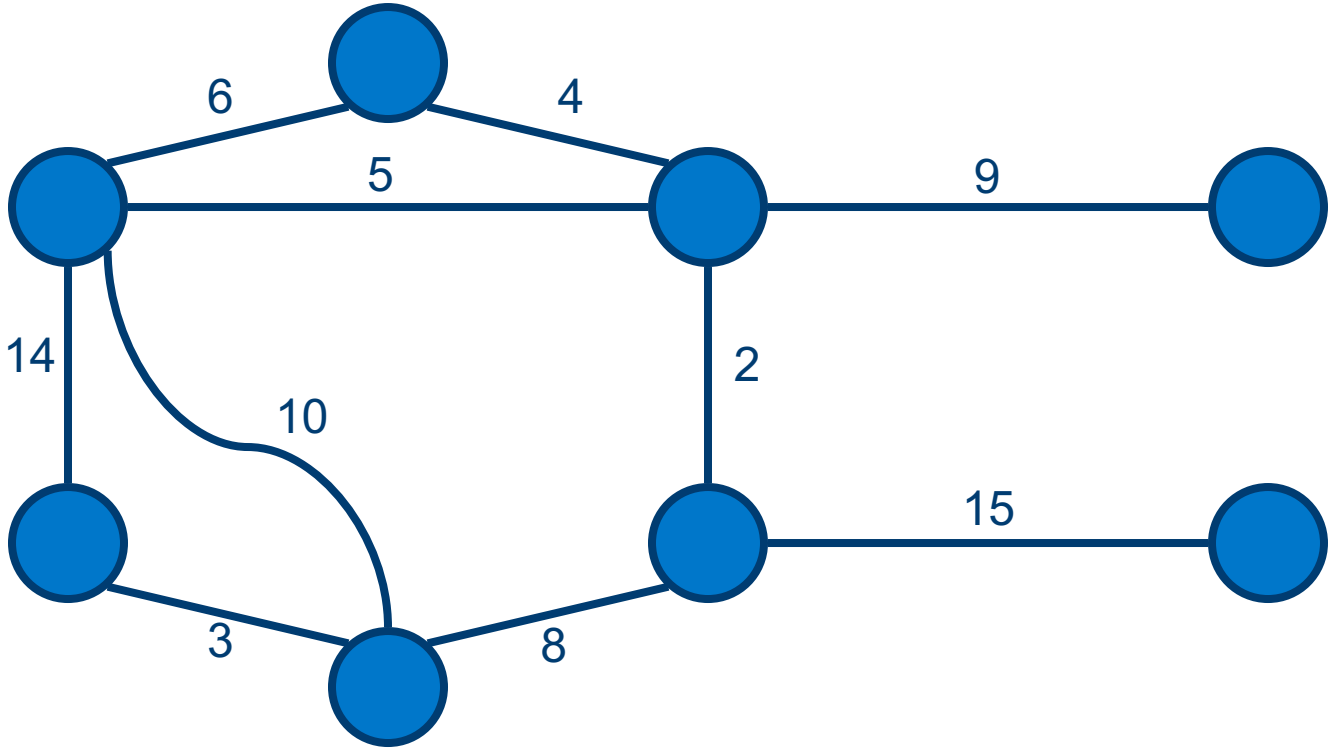
- Algorithm:

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

Minimum Spanning Tree

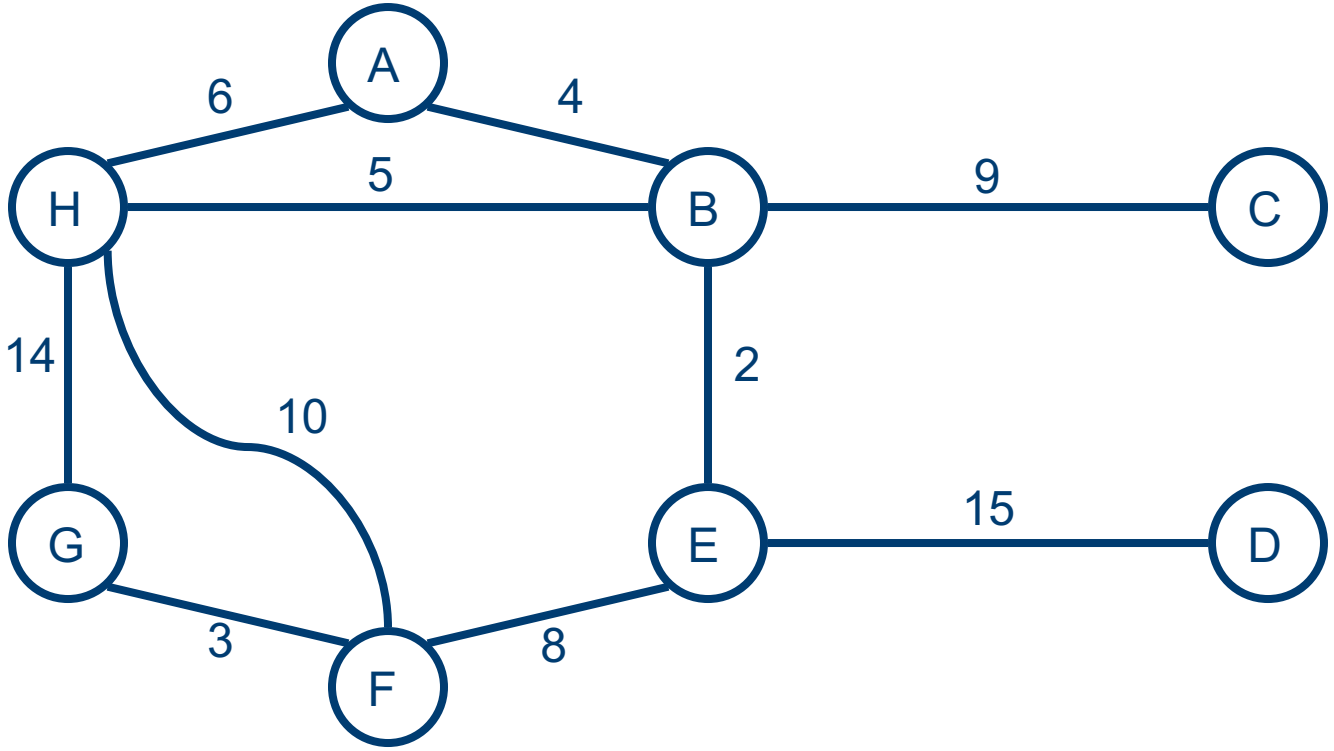
- Problem: given a connected, undirected, weighted graph, find a spanning tree using edges that minimize the total weight



Minimum Spanning Tree

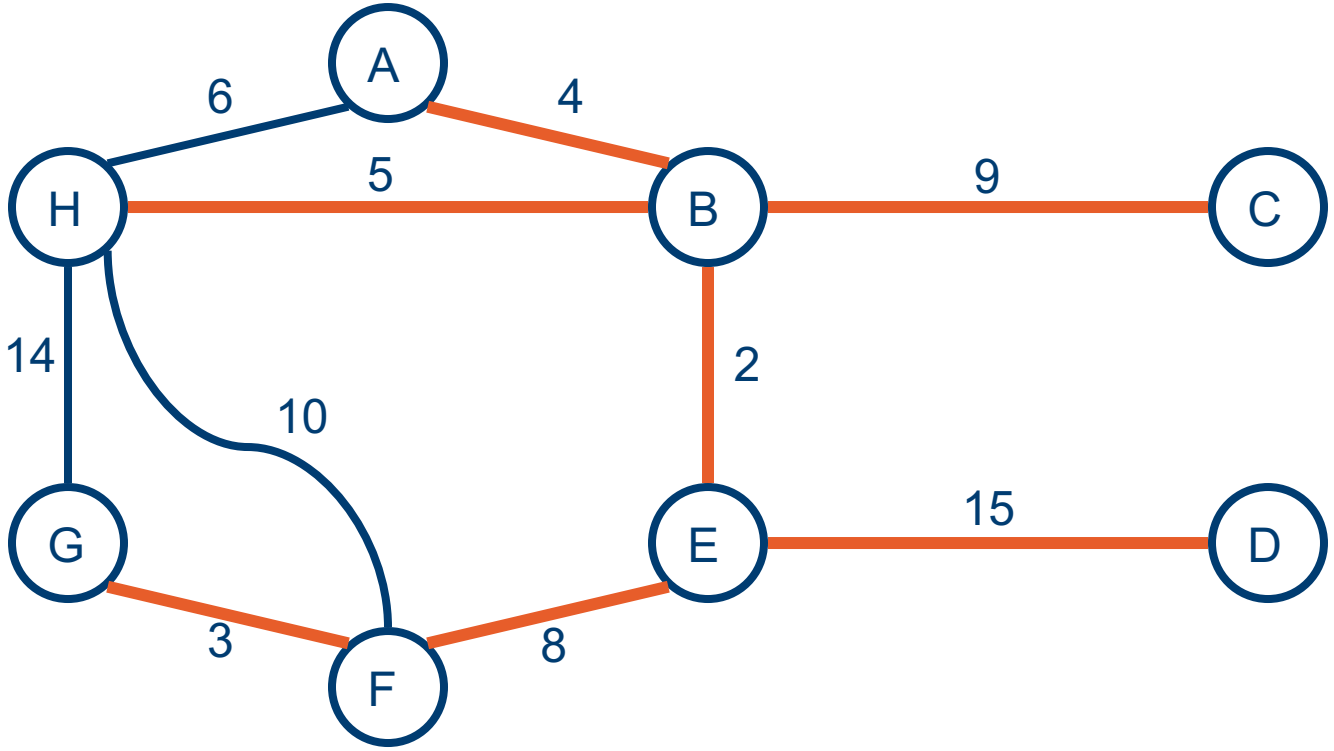
Minimum Spanning Tree

- Which edges form the minimum spanning tree (MST) of the below graph?



Minimum Spanning Tree

- Answer:



Kruskal's algorithm

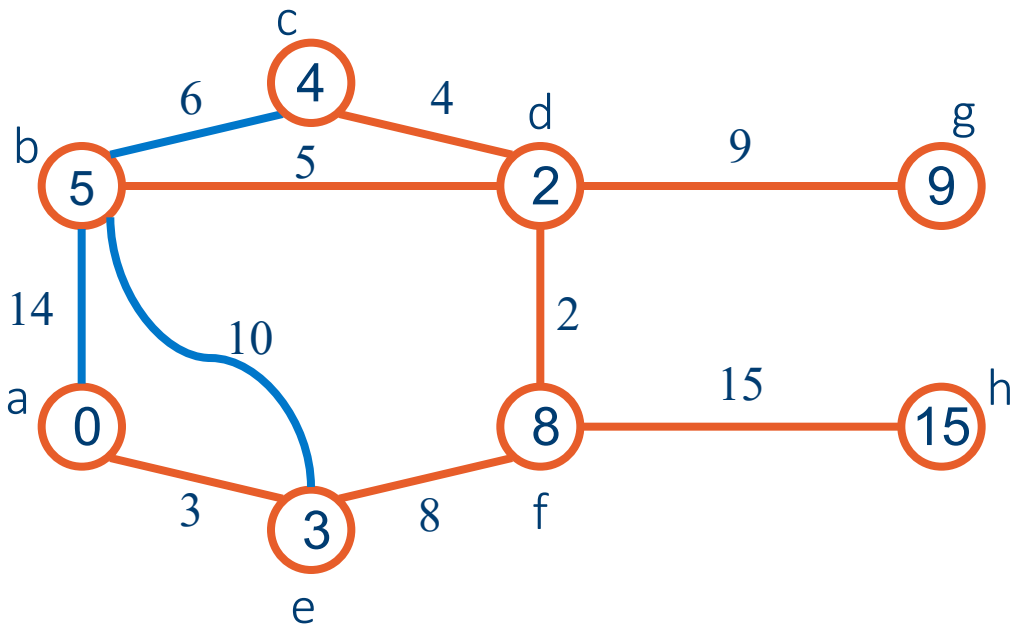
```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Time Complexity

$$O(E \log V)$$

Prim's Algorithm

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```



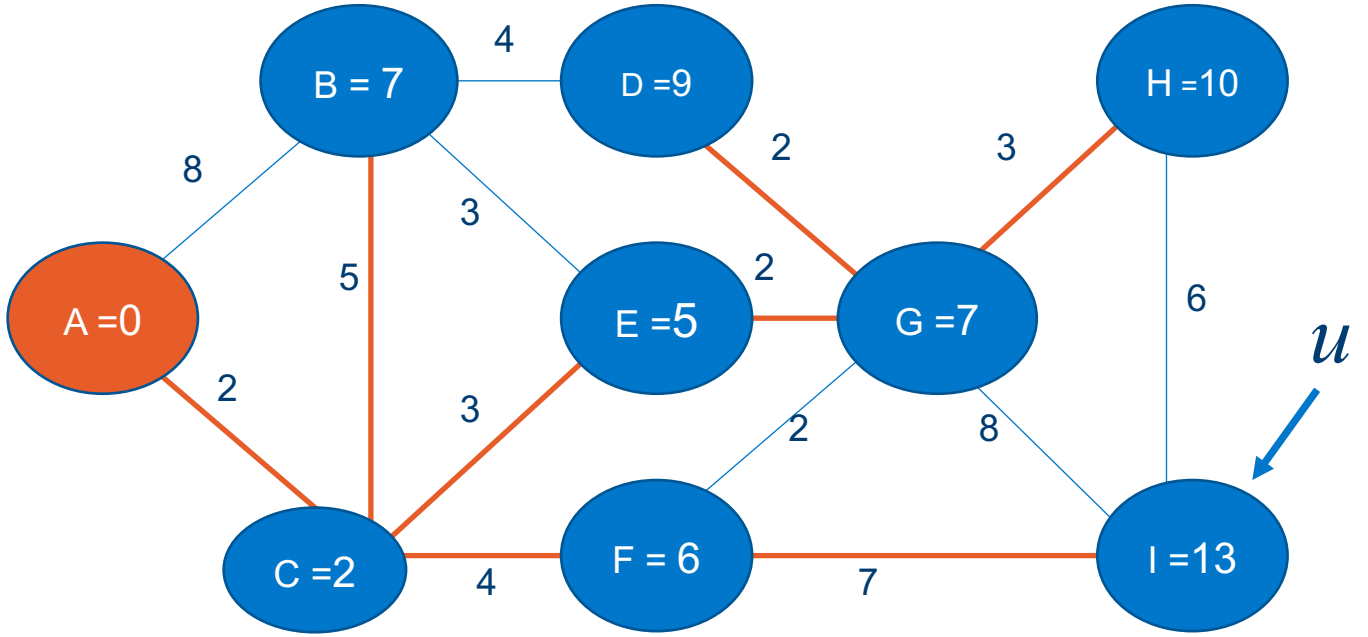
Single Source Shortest Paths

Dijkstra vs. Bellman-Ford

- Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative.
- Bellman-Ford algorithm, allow negative-weight edges in the in-put graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.
  - Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Dijkstra's algorithm

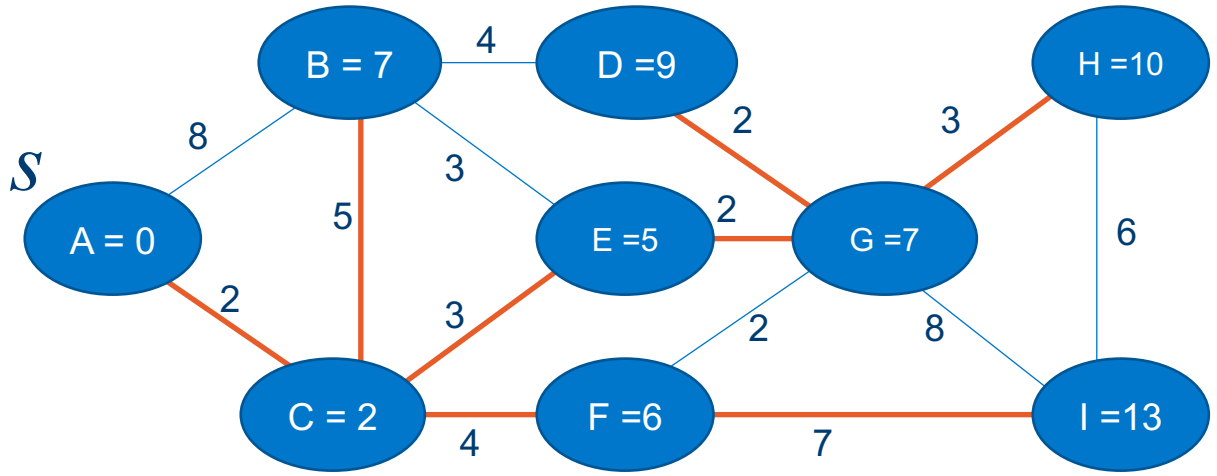
```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          if v.d > u.d + w(u, v)
              v.d = u.d + w(u, v)
              v.π = u
          if v.d changed
              DECREASE-KEY(Q, v, v.d)
```



Q: A(0), B(7), C(2), D(9), E(5), F(6), G(7), H(10), I(13)

Bellman-Ford

```
BELLMAN-FORD(G, w, s)
    for each v ∈ G.V
        v.d = ∞
        v.π = NIL
    s.d = 0
    for i = 1 to |G.V| − 1
        for each edge (u, v) ∈ G.E
            if v.d > u.d + w(u, v)
                v.d = u.d + w(u, v)
                v.π = u
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            return FALSE
    return TRUE
```



$$\Theta(VE)$$

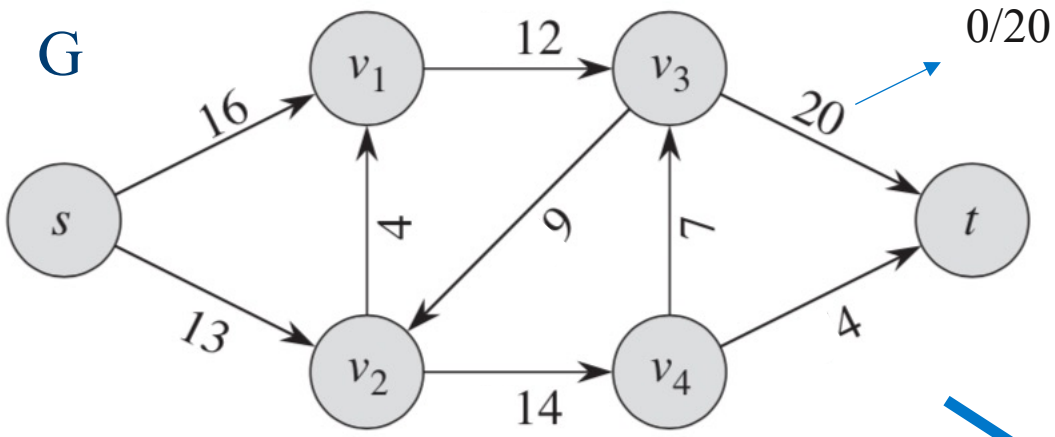
Flow Network

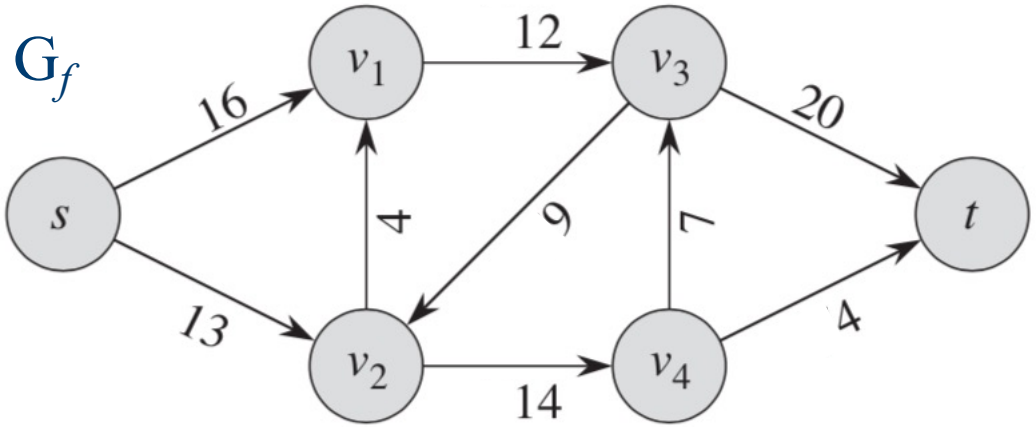
Ford-Fulkerson Algorithm

- Given G, s, t, and c, it finds a flow whose value is maximum

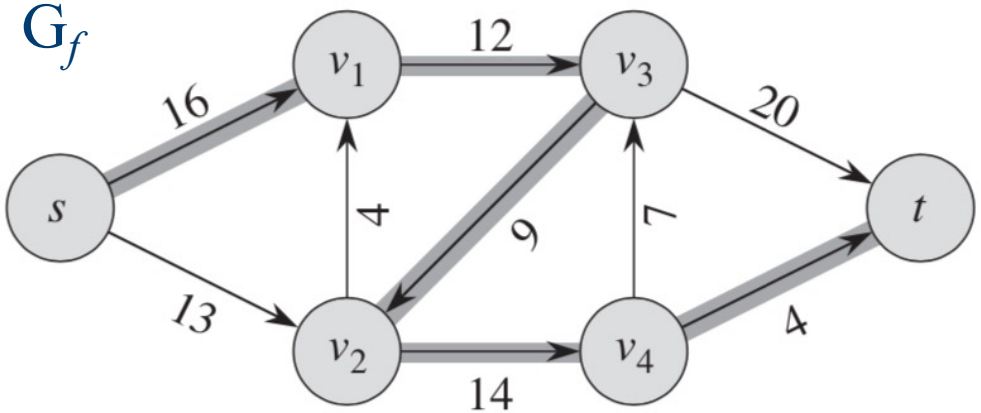
- General ideas:
  - In each iteration of the Ford-Fulkerson method, we find some augmenting path p and use p to modify the flow f
  - That is, adding flow when the residual edge in p is an original edge and subtracting it otherwise
  - When no augmenting paths exist, the flow f is a maximum flow.

Ford-Fulkerson Algorithm Example

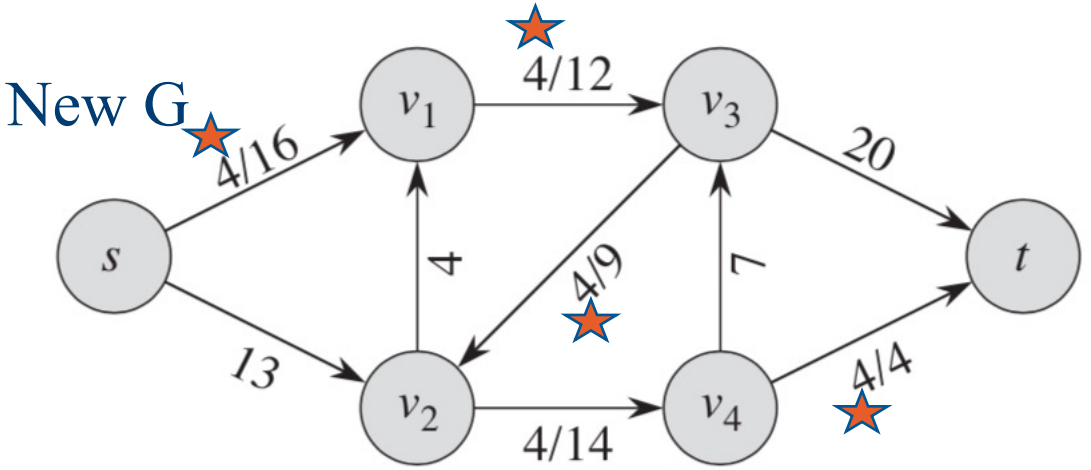




Are there any augmenting path?



$c_f(p) = 4$



Thanks