0-1 Knapsack Problem

- A thief robbing a store finds n items.
- The i-th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers.
- The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W.
- Which items should he take?
- We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind.

Fractional Knapsack Problem

- The thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.
- You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Knapsack Problems

- Both 0-1 and fractional knapsack problems exhibit the optimal-substructure property.
- The fractional knapsack problem is solvable by a greedy strategy.
- The 0-1 knapsack problem is not solvable by a greedy strategy.
- The dynamic-programming is needed to find optimal solution for the 0-1 knapsack problem.

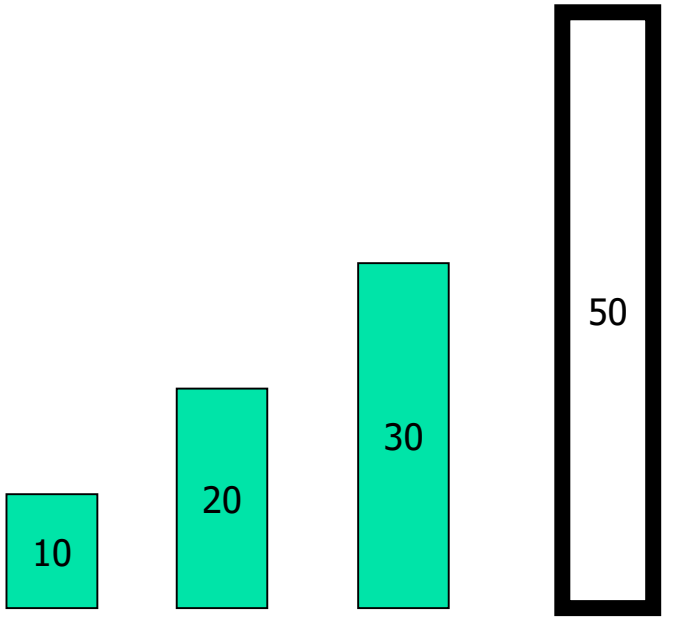
Optimal Substructure Property

- 0-1 knapsack problem
  - Consider the most valuable load that weighs at most W pounds.
  - If we remove item j from this load, the remaining load must be the most valuable load weighing at most W-w_j that the thief can take from the n-1 original items excluding j.
- Fractional knapsack problem
  - If we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most W - w that the thief can take from the n-1 original items plus w_j-w pounds of item j.

Greedy Choice Property

- To solve the fractional problem, we first compute the value per pound v_i/w_i for each item.
- Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W .
- Thus, by sorting the items by value per pound, the greedy algorithm runs in O(n lg n) time.

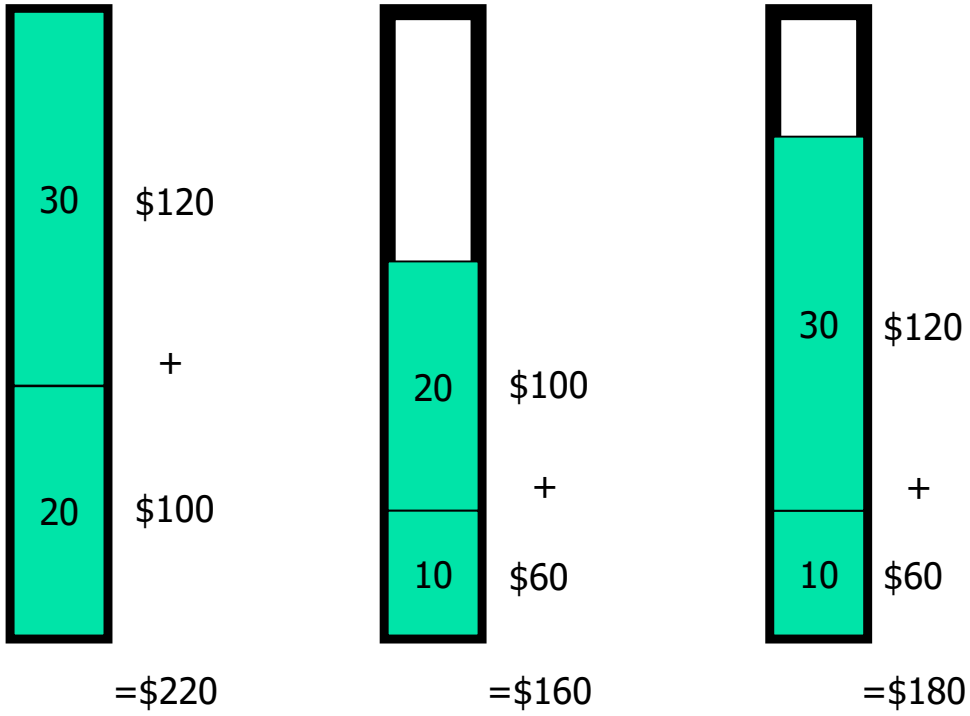
0-1 Knapsack Problem



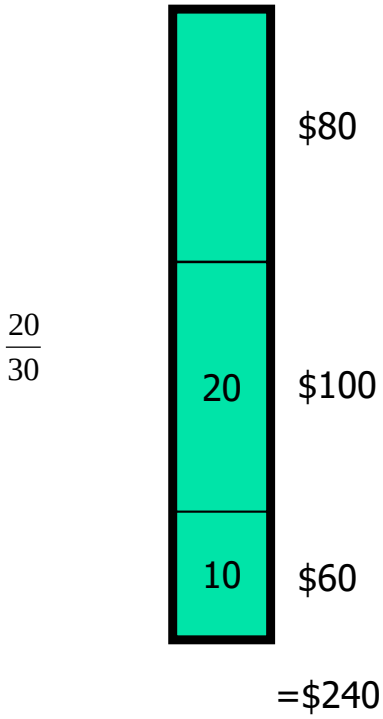
The value per pound

6 5 4

0-1 Knapsack Problem



Fractional Knapsack Problem



Dynamic Programming for 0-1 Knapsack Problem

- Let c[i,w]=value of solution for items 1…i and maximum weight w

$$c[i,w] = \begin{cases} 0 & if\ i = 0\ or\ w = 0 \\ c[i-1,w] & if\ w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & if\ i > 0\ and\ w \geq w_i \end{cases}$$

Dynamic Programming for 0-1 Knapsack Problem

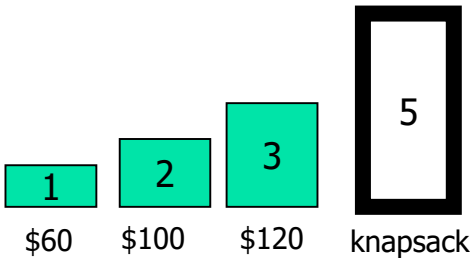
```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

$$c[i,w] = \begin{cases} 0 & if\ i = 0\ or\ w = 0 \\ c[i-1,w] & if\ w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & if\ i > 0\ and\ w \geq w_i \end{cases}$$

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

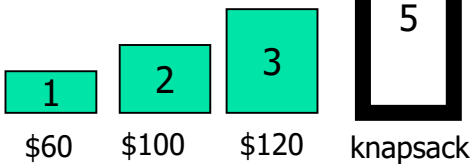


Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i \ W	0	1	2	3	4	5
0						
1						
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120

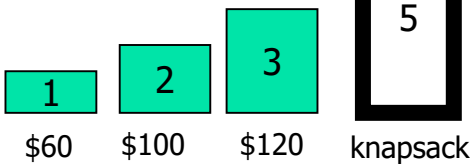


Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



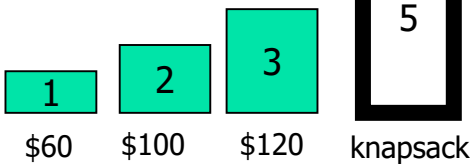
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 0

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



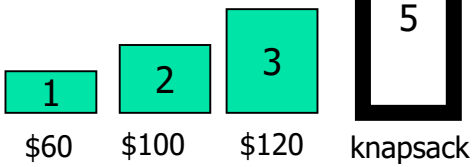
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 0

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



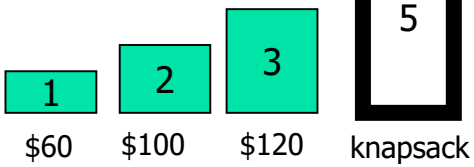
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 1

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60				
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



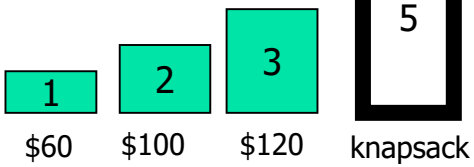
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 2

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60			
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



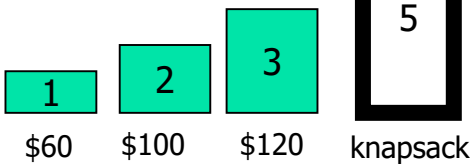
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 3

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60		
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



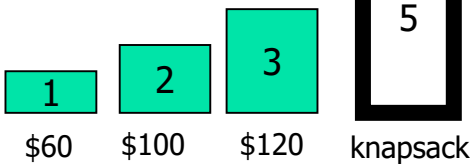
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 4

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



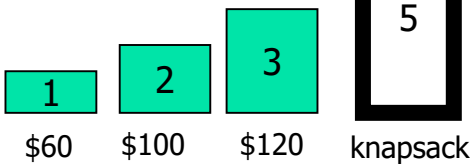
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 1
w = 5

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



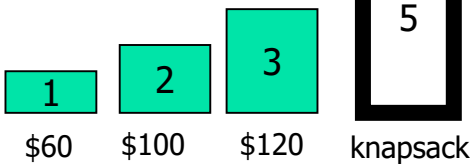
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 0

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2						
3						

index	1	2	3
w	1	2	3
v	60	100	120



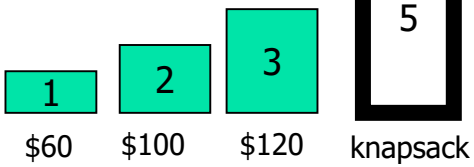
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 0

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0					
3						

index	1	2	3
w	1	2	3
v	60	100	120



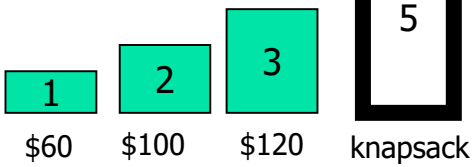
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 1

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0					
3						

index	1	2	3
w	1	2	3
v	60	100	120



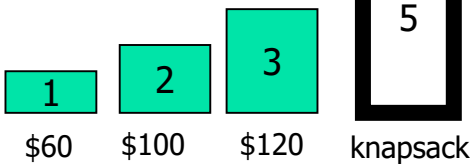
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 1

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60				
3						

index	1	2	3
w	1	2	3
v	60	100	120



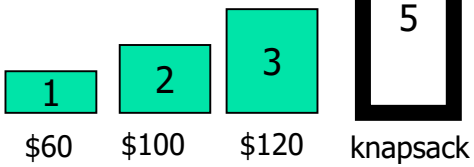
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 2

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100			
3						

index	1	2	3
w	1	2	3
v	60	100	120



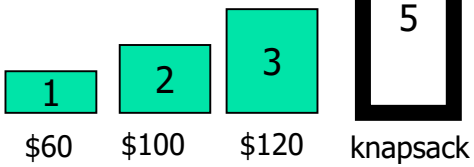
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 3

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160		
3						

index	1	2	3
w	1	2	3
v	60	100	120



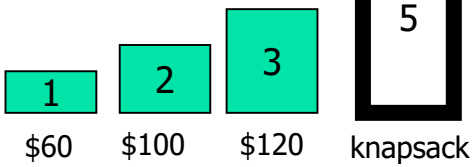
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 4

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	
3						

index	1	2	3
w	1	2	3
v	60	100	120



Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

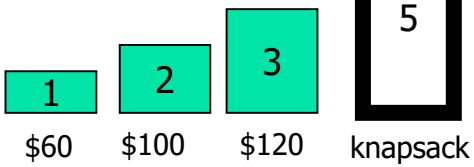
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 2
w = 5

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3						

index	1	2	3
w	1	2	3
v	60	100	120



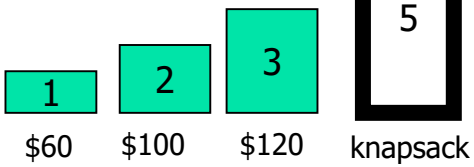
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 0

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0					

index	1	2	3
w	1	2	3
v	60	100	120



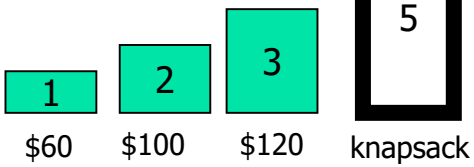
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 0

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0					

index	1	2	3
w	1	2	3
v	60	100	120



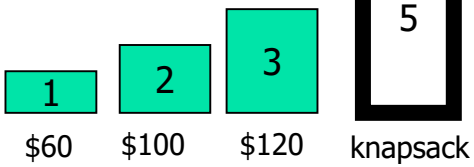
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 1

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0					

index	1	2	3
w	1	2	3
v	60	100	120



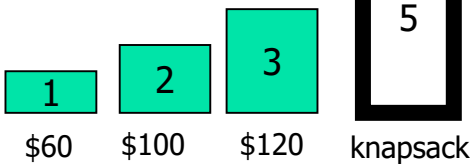
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 1

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60				

index	1	2	3
w	1	2	3
v	60	100	120



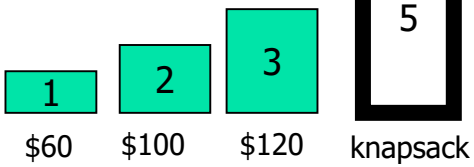
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 2

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60				

index	1	2	3
w	1	2	3
v	60	100	120



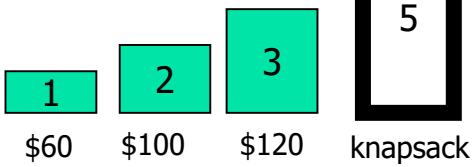
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 2

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100			

index	1	2	3
w	1	2	3
v	60	100	120



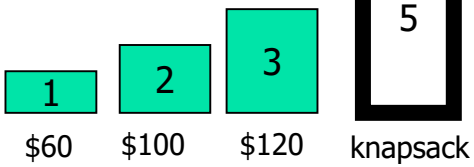
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 3

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160		

index	1	2	3
w	1	2	3
v	60	100	120



Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

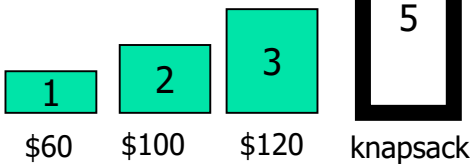
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 4

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	

index	1	2	3
w	1	2	3
v	60	100	120



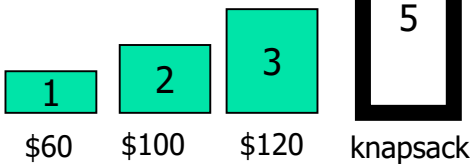
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i = 3
w = 5

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

index	1	2	3
w	1	2	3
v	60	100	120

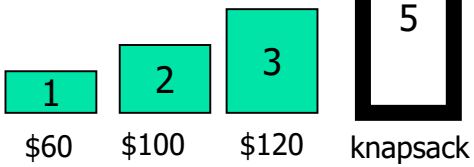


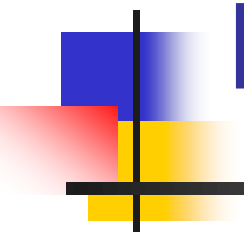
Dynamic Programming for 0-1 Knapsack Problem

```
DP-KNAPSACK(n, W)
1.    for w = 0 to W
2.        C[0,w] = 0
3.    for i = 1 to n
4.        for w = 0 to W
5.            if (w[i] ≤ w)
6.                C[i,w] = max{C[i-1,w], v[i]+C[i-1,w-w[i]]}
7.            else
8.                C[i,w] = C[i-1,w]
9.    return  C[n,W]
```

i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

index	1	2	3
w	1	2	3
v	60	100	120





Huffman Codes

Huffman Codes

- Huffman codes compress data very effectively – typically savings of 20% to 90%.
- We consider the data to be a sequence of characters.
- Huffman's greedy algorithm uses a table giving how often each character occurs to build up an optimal way of representing each character as a binary string.
- Suppose we have a 100 character data file that we wish to store compactly.
- We observe that the characters in the file occur with the frequencies below. That is, only 6 different characters appear, and the character a occurs 45 times.

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

Huffman Codes

- We have many options for how to represent such a file of information.
- Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a codeword.
- If we use a fixed-length code, we need 3 bits to represent 6 characters: a = 000, b = 001, . . . , f = 101.
- This method requires 300 bits to code the entire file with 100 characters.
- Can we do better?

Huffman Codes

- A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.
- Figure below shows such a code.

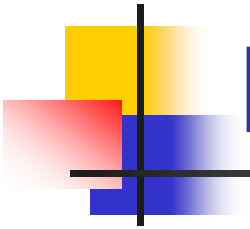
	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- This code requires (45*1+13*3+12*3+16*3+9*4+5*4) = 224 bits (savings of approximately 25%).
- In fact, this is an optimal character code for this file, as we shall see.

Prefix Codes

- We consider here only codes in which no codeword is also a prefix of some other codeword.
- Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.
- We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file.
- In our example, the string 001011101 parses uniquely as 0 0 101 1101, which decodes to aabe.
- Although we won't prove it here, a prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.

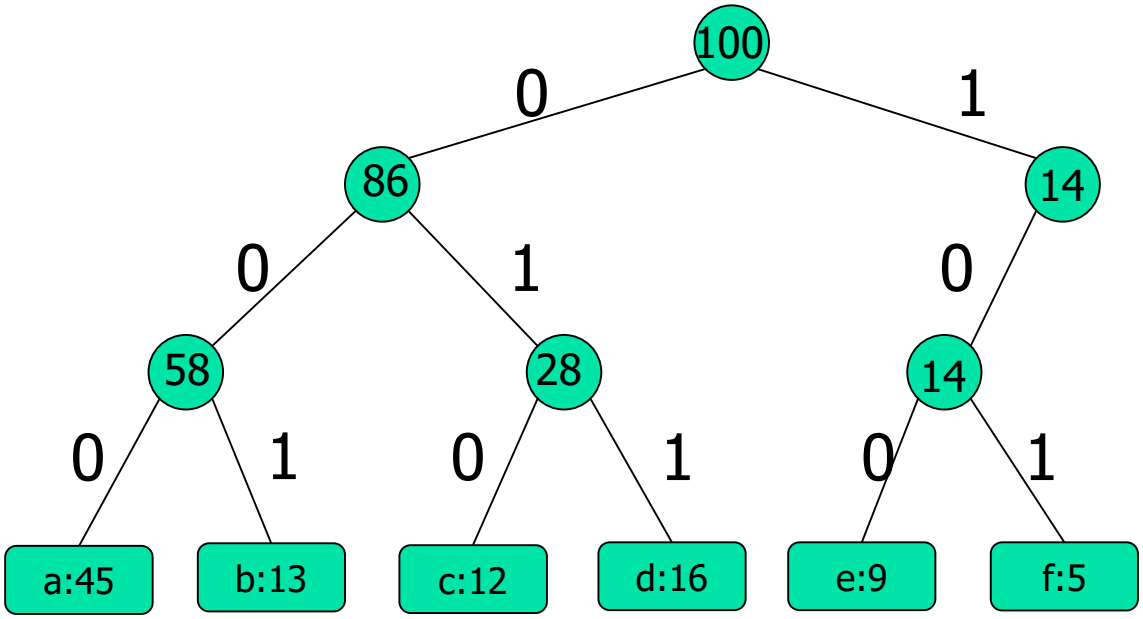
	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100



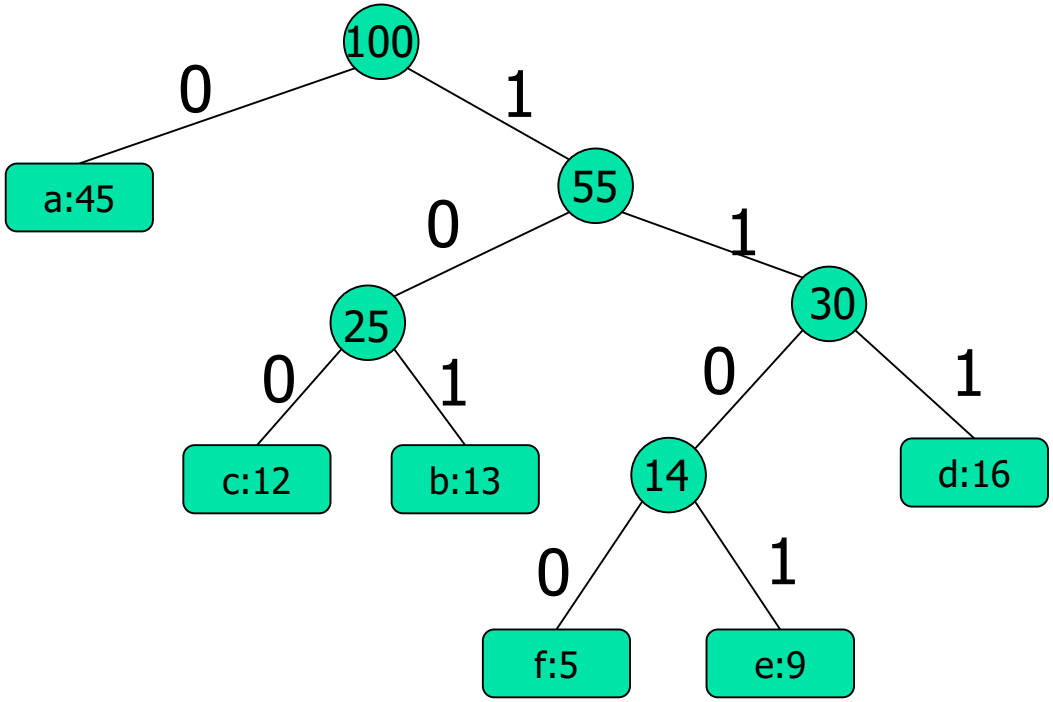
Prefix Codes

- The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword.
- A binary tree whose leaves are the given characters provides one such representation.
- We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child."

A Tree Corresponding to the Fixed-length Code



A Tree Corresponding to the Variable-length Code



Prefix Codes

- An optimal code for a file is always represented by a full binary tree, in which every non-leaf node has two children (see Exercise 16.3-2).
- The fixed-length code in our example is not optimal since its tree, shown previously, is not a full binary tree.
- Since we can now restrict to full binary trees, if C is the alphabet from which the characters are drawn and all character frequencies are positive, the tree for an optimal prefix code has exactly |C| leaves, one for each letter of the alphabet, and exactly |C|-1 internal nodes.

Prefix Codes

- Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file.
- For each character c in the alphabet C, let the attribute c.freq denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree (i.e. length of the codeword for character c).
- The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) * d_T(c)$$

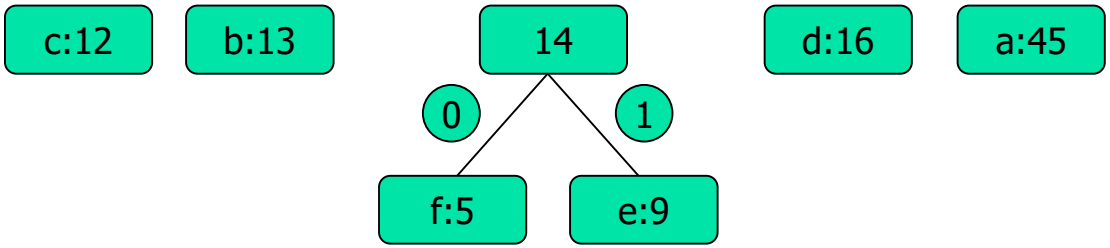
Greedily Constructing a Huffman Code

Step1: Make frequency table and sort it.

f:5	e:9	c:12	b:13	d:16	a:45

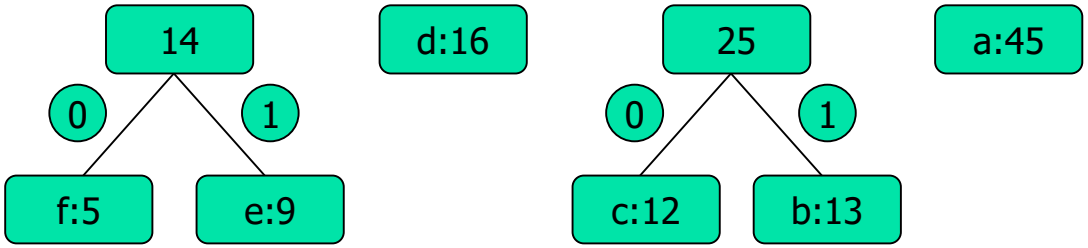
Greedily Constructing a Huffman Code

Step2: Extract top-two element and merge into one node.

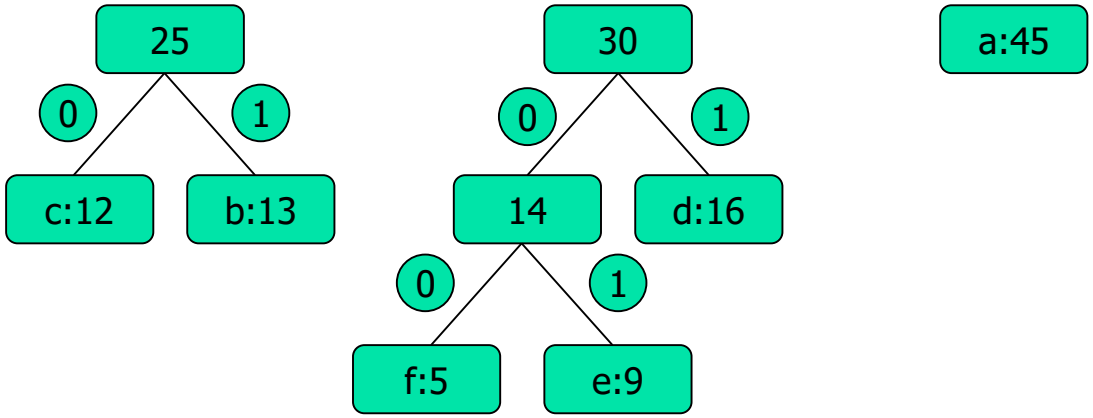


Greedily Constructing a Huffman Code

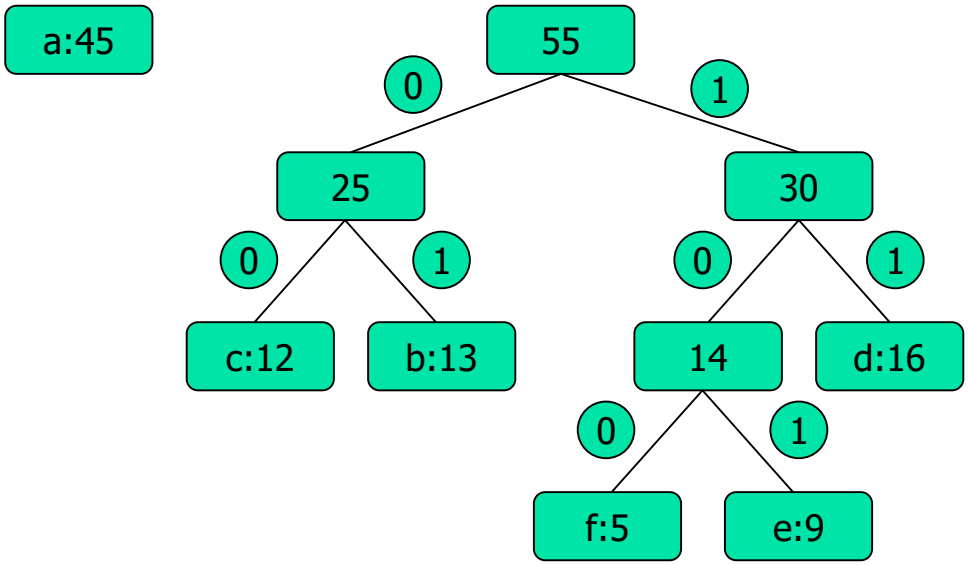
Step3: Back to Step 1 until there's only one element in the Queue.



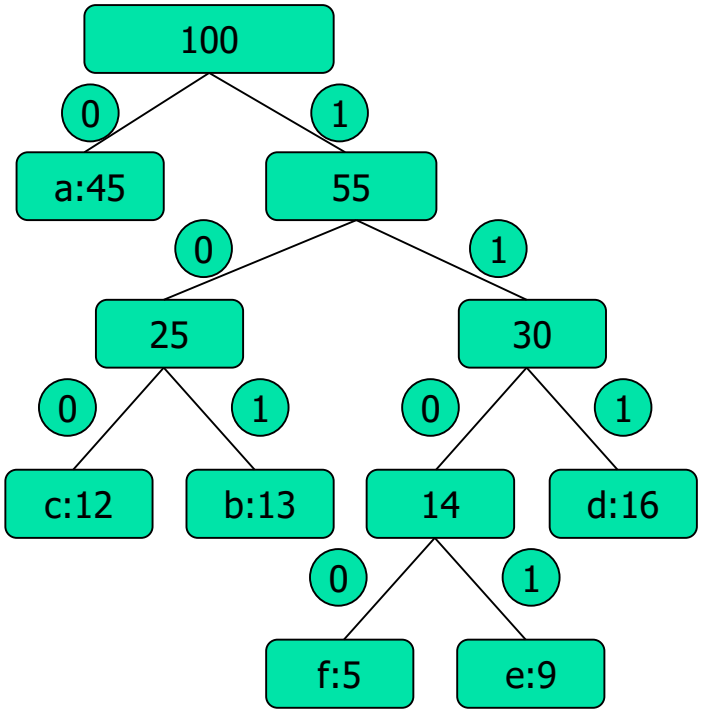
Greedily Constructing a Huffman Code



Greedily Constructing a Huffman Code



Greedily Constructing a Huffman Code



Constructing a Huffman Code

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

- For a set C of n characters, we can initialize Q in line 2 in O(n) time using the BUILD-MIN-HEAP procedure.
- The **for** loop in lines 3–8 executes exactly n-1 times, and since each heap operation requires time O(lg n), the loop contributes O(n lg n) to the running time.
- Thus, total running time on a file with n characters is O(n lg n).

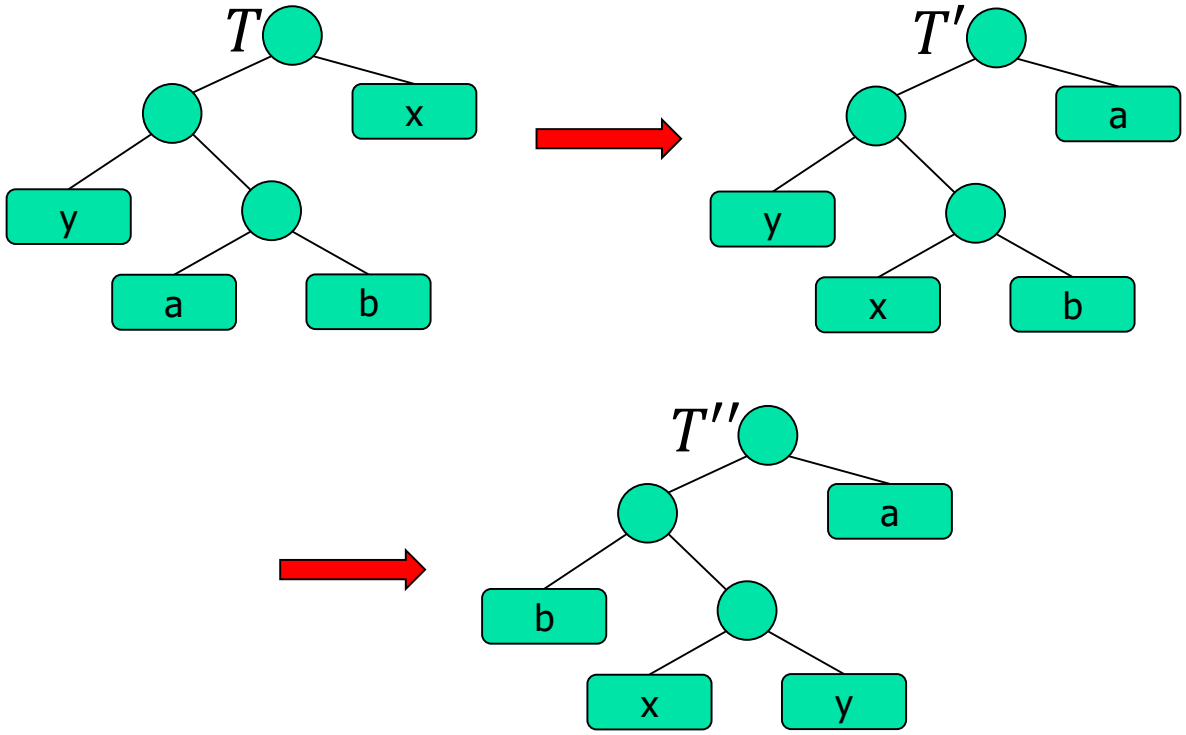
Correctness of Huffman's Algorithm

- Greedy Choice
  - In each step, we select and extract two minimum elements in the queue, merge them into one node and insert the node in the queue again.
  - We can prove that this greedy choice yields globally optimal solution.
- Optimal Substructure
  - After greedy choice, the sub-solution must be optimal solution.
  - In this case, the sub-tree T' of T, $T' = T - \{x,y\}$, represents an optimal prefix code for the alphabet $C' = C - \{x,y\} \cup \{z\}$.
  - Let z be the parent of x and y, and $f(z) = f(x) + f(y)$.
- Huffman's Algorithm produces an optimal prefix code because it satisfies above two properties.

Greedy Choice Property

- Lemma
  - Let C be an alphabet in which each character $c \in C$ has frequency c.freq.
  - Let x and y be two characters in C having the lowest frequencies.
  - Then, there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Proof
  - The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree.
  - If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

An Illustration of the Key Step in the Proof



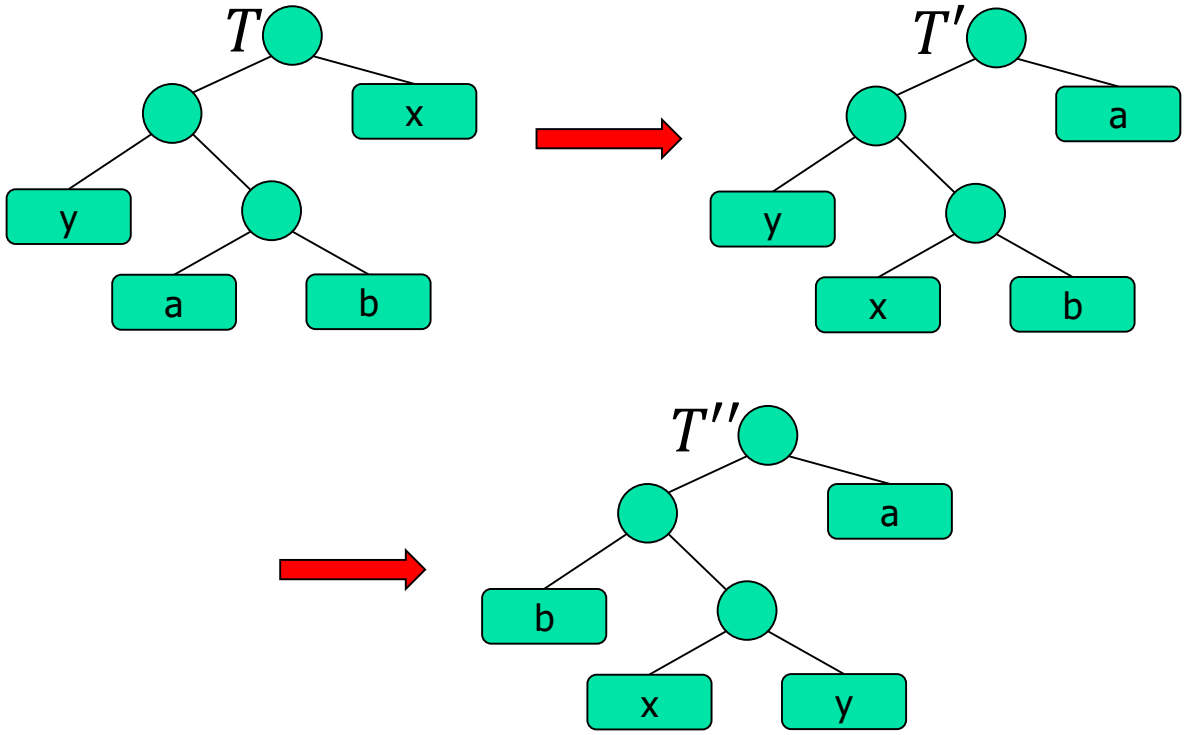
Greedy Choice Property

- Lemma
  - Let C be an alphabet in which each character $c \in C$ has frequency c.freq.
  - Let x and y be two characters in C having the lowest frequencies.
  - Then, there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Proof
  - Let a and b be two characters that are sibling leaves of maximum depth in T
  - Without loss of generality, we assume that a.freq $\leq$b.freq and x.freq $\leq$ y.freq.
  - Since x.freq and y,freq are the two lowest leaf frequencies, in order, and a.freq and b.freq are two arbitrary frequencies, in order, we have x.freq$\leq$ a.freq and y.freq $\leq$ b.freq.

Greedy Choice Property

- Lemma
  - Let C be an alphabet in which each character $c \in C$ has frequency c.freq.
  - Let x and y be two characters in C having the lowest frequencies.
  - Then, there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Proof
  - It is possible that we could have x.freq = a.freq or y.freq = b.freq.
  - However, if we had x.freq = b.freq, then we would also have a.freq = b.freq = x.freq = y.freq and and the lemma would be trivially true.
  - Thus, we will assume that x.freq $\neq$ b.freq (i.e., $x \neq b$).

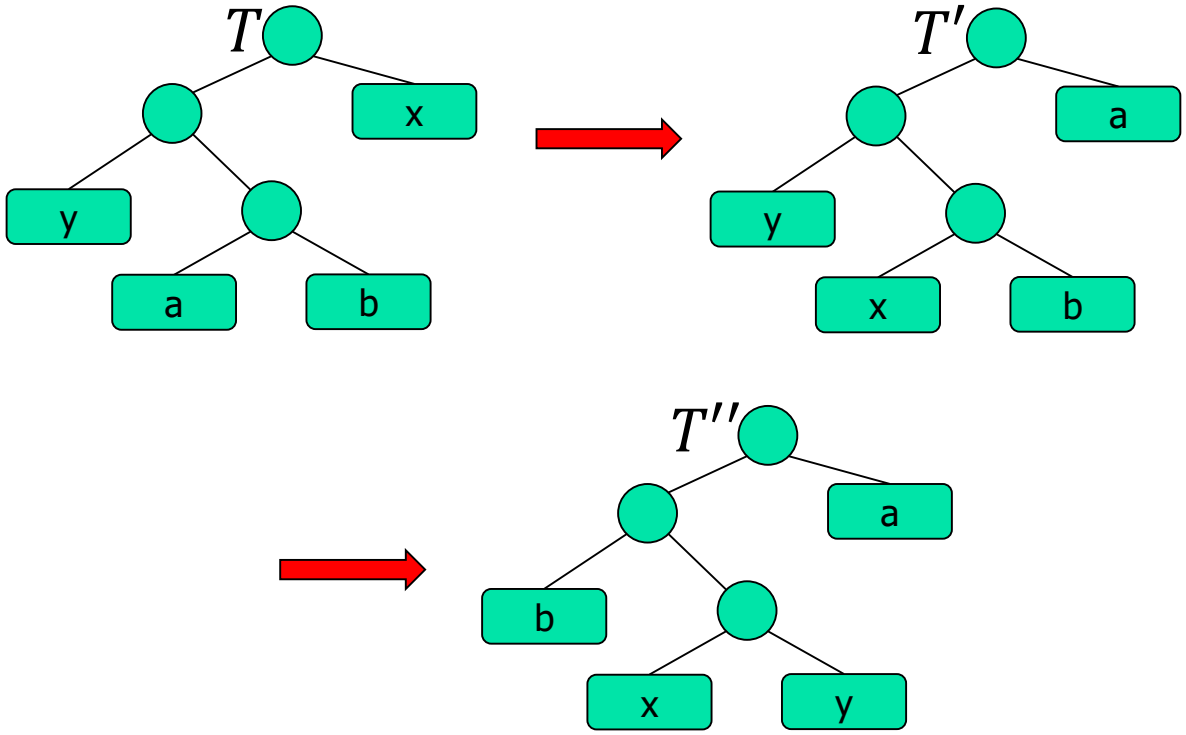
An Illustration of the Key Step in the Proof



Greedy Choice Property

- Lemma 16.2
  - Let C be an alphabet in which each character $c \in C$ has frequency c.freq.
  - Let x and y be two characters in C having the lowest frequencies.
  - Then, there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Proof
  - We exchange the positions in T of a and x to produce a tree T′, and then we exchange the positions in T′ of b and y to produce a tree T″ in which x and y are sibling leaves of maximum depth.
  - The difference in cost between T and T′ is
    - $B(T) - B(T') = \sum_{c \in C} c.freq * dT(c) - \sum_{c \in C} c.freq * dT'(c)$
      $= x.freq * d_T(x) + a.freq * d_T(a) - x.freq * d_{T'}(x) - a.freq * d_{T'}(a)$
      $= x.freq * d_T(x) + a.freq * d_T(a) - x.freq * d_T(a) - a.freq * d_T(x)$
      $= (a.freq - x.freq)(d_T(a) - d_T(x)) \geq 0.$

An Illustration of the Key Step in the Proof



Greedy Choice Property

- Lemma 16.2
  - Let C be an alphabet in which each character $c \in C$ has frequency c:freq.
  - Let x and y be two characters in C having the lowest frequencies.
  - Then, there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Proof
  - We exchange the positions in T of a and x to produce a tree T', and then we exchange the positions in T' of b and y to produce a tree T" in which x and y are sibling leaves of maximum depth.
  - Similarly, the difference in cost between T' and T" is $B(T') - B(T'') \geq 0$. (i.e., $B(T'') \leq B(T)$).
  - Since T is optimal, we also have $B(T) \leq B(T'')$.
  - Both conditions imply $B(T) = B(T'')$.
  - Thus, T" is an optimal tree in which x and y appear as sibling leaves of maximum depth.

Optimal Substructure

- Lemma 16.3
  - Let C be a given alphabet with frequency c.freq defined for each character $c \in C$.
  - Let x and y be two characters in C with minimum frequency.
  - Let C′ be the alphabet C with characters x,y removed and (new) character z added, so that $C' = C - \{x,y\} \cup \{z\}$.
  - Define f for C′ as for C, except that z.freq=x.freq+y.freq.
  - Let T′ be any tree representing an optimal prefix code for the alphabet C′.
  - Then the tree T, obtained from T′ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

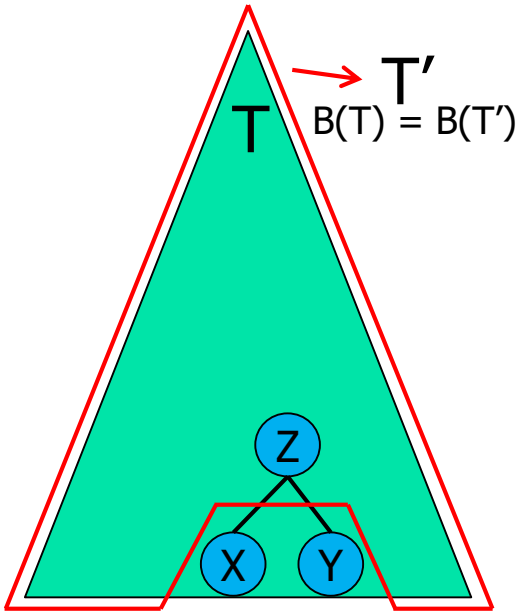
Optimal Substructure

- We first show how to express the cost B(T) of tree T in terms of the cost B(T′) of tree T′.
- For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq * d_T(c) = c.freq * d_{T'}(c)$.
- Since $d_T(x) = d_T(y) = d_{T'}(z)+1$, we have

$$x.freq * d_T(x) + y.freq * d_T(y)$$
$$= (x.freq + y.freq)(d_{T'}(z)+1)$$
$$= z.freq * d_{T'}(z) + (x.freq + y.freq).$$

- Thus, we have $B(T) = B(T') + x.freq + y.freq$.

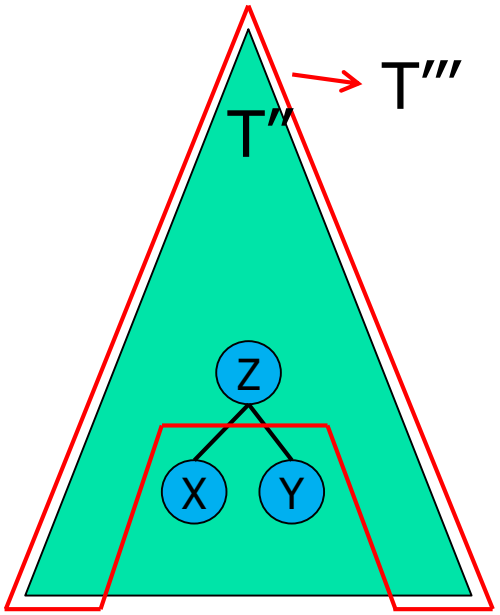
Optimal Substructure



B(T) = B(T′) + x.freq + y.freq.

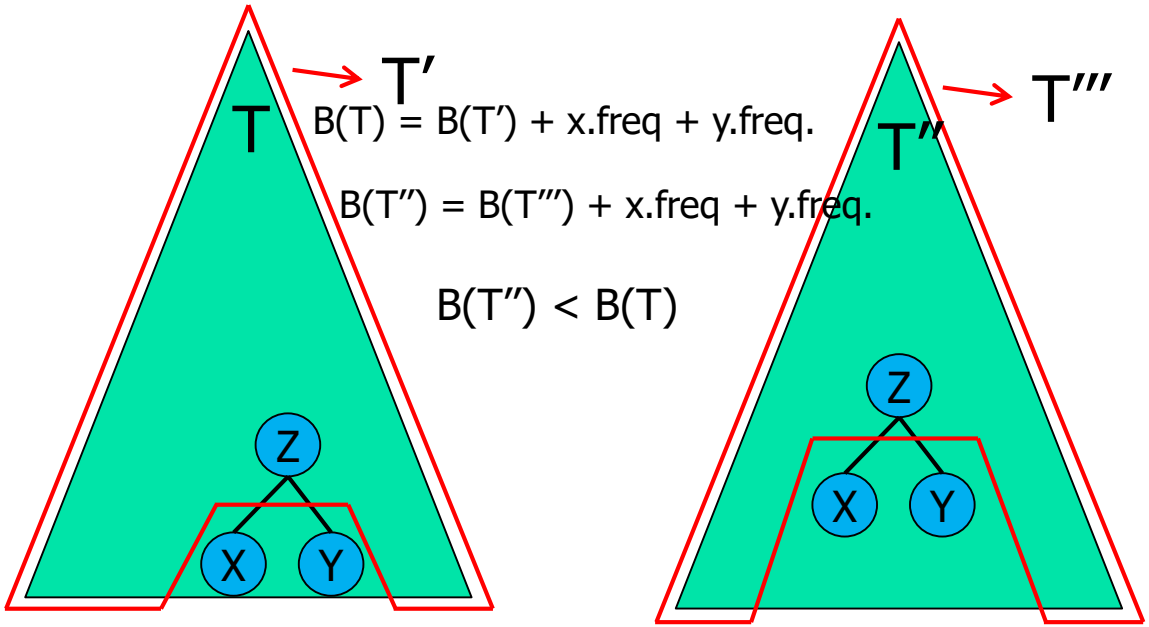
Optimal Substructure

- Suppose that T does not represent an optimal prefix code for C.
- Then, there exists an optimal tree T″ such that B(T″) < B(T).
- Without loss of generality (by Lemma 16.2), T″ has x and y as siblings.
- Let T‴ be the tree T″ with the common parent of x and y replaced by a leaf z with frequency z.freq = x.freq + y.freq.



B(T″) = B(T‴) + x.freq + y.freq

Optimal Substructure



Optimal Substructure

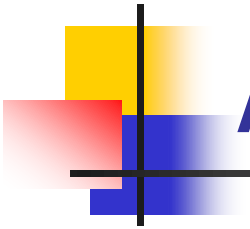
- We have
  - $B(T) = B(T') + x.freq + y.freq$
  - $B(T'') = B(T''') + x.freq + y.freq$
  - $B(T'') < B(T)$
- We now prove by contradiction.
  - Suppose that T does not represent an optimal prefix code for C.
  - Then, there exists an optimal tree T″ such that $B(T'') < B(T)$.
  - Without loss of generality (by Lemma 16.2), T″ has x and y as siblings.
  - Let T‴ be the tree T″ with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$.
  - $B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T')$ yielding a contradiction to the assumption that T′ represents an optimal prefix code for C′.
  - Thus, T must be optimal for the alphabet C.

Optimality of the Huffman

- Then we can get the optimal prefix code for C using Huffman.
  - The greedy choice property
    - There must be an optimal code for two least frequent elements that have the same length and differ only in the last 1 bit.
    - So we build one node using the two least frequent elements, and instead of the two elements, insert the new node with the frequency that's the sum of the two elements.
  - The optimal substructure property
    - A tree that's constructed using the remaining elements, must be optimal, too.
  - So if we do this step repeatedly, we can get the optimal prefix code.

Optimality of the Huffman Code

- Theorem 16.4
  - Procedure HUFFAN produces an optimal prefix code.
- Proof
  - Immediate from Lemmas 16.2 and 16.3.

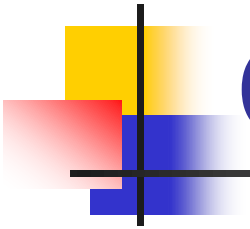


Any Question?

Introduction to Algorithms (Chapter 21: Data Structures for Disjoint Sets)

Kyuseok Shim

Electrical and Computer Engineering
Seoul National University



Outline

- In this chapter, we shall describes the operations supported by a disjoint-set data structure and we present efficient implementations for disjoint sets.

Disjoint Sets Data Structure

- Some applications involve grouping n distinct elements into a collection of disjoint sets.
- These applications often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets.
- A disjoint-set is a collection $S=\{S_1, S_2,\ldots, S_k\}$ of distinct dynamic sets.
- Each set is identified by a member of the set, called representative which is some member of the set.

Disjoint Sets Data Structure

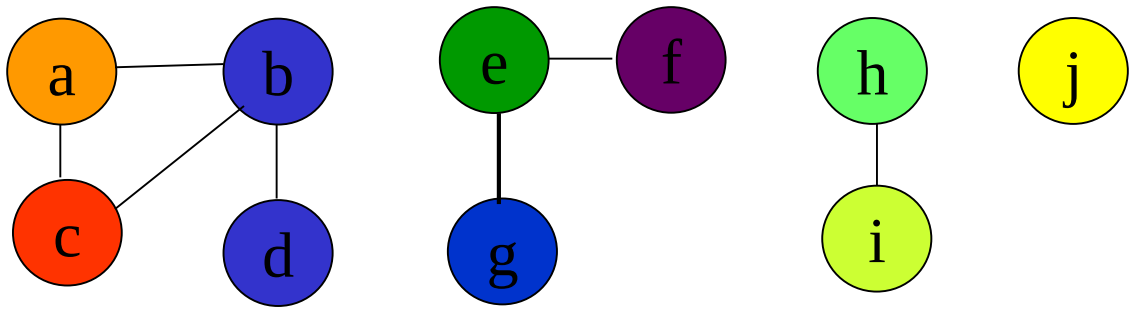
- We represent each element of a set by an object.
- Letting x denote an object, we wish to support the following operations:
  - MAKE-SET(x): Creates a new set with only x.
    - Since the sets are disjoint, we require that x not already be in some other set.
  - UNION(x, y): Combines the two sets S_x and S_y, containing x and y respectively, into a new set that is the union of these two sets.
    - Assume that the two sets are disjoint prior to the operation
    - The representative of the resulting set is any member of $S_x \cup S_y$.
    - Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y.
    - In practice, we often absorb the elements of one of the sets into the other set.
  - FIND-SET(x): Returns the representative of the set containing x.

Disjoint-set Operations

- We shall analyze the running times of disjoint-set data structures in terms of two parameters.
  - n: the number of MAKE-SET operations
  - m: the total number of MAKE-SET, UNION, and FIND-SET operations
- We assume that the n MAKE-SET operations are the first n operations performed.
- Since the sets are disjoint, each UNION operation reduces the number of sets by one.
  - The number of UNION operations is thus at most n-1.
- Note also that since the MAKE-SET operations are included in the total number of operations m, we have $m \geq n$.

An Application of Disjoint-set Data Structures

- One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.
  - An undirected graph is connected if there is a path from every vertex to every other vertex.



An Example of a Graph with 4 Connected Components

Finding Connected Component of an Undirected Graph

- The procedure CONNECTED-COMPONENTS uses the disjoint-set operations to compute the connected components of a graph.
- In pseudocode, we denote the set of vertices of a graph G by G.V and the set of edges by G.E.

```
CONNECTED-COMPONENTS(G)
1.    for each vertex v ∈ G.V
2.        MAKE-SET(v)
3.    for each edge (u, v) in G.E
4.      if FIND-SET(u) ≠ FIND-SET(v)
5.            UNION(u,v)
```

Finding Connected Component of an Undirected Graph

- The procedure CONNECTED-COMPONENTS uses the disjoint-set operations to compute the connected components of a graph.
- In pseudocode, we denote the set of vertices of a graph G by G.V and the set of edges by G.E.
- Once CONNECTED COMPONENTS has preprocessed the graph, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component.

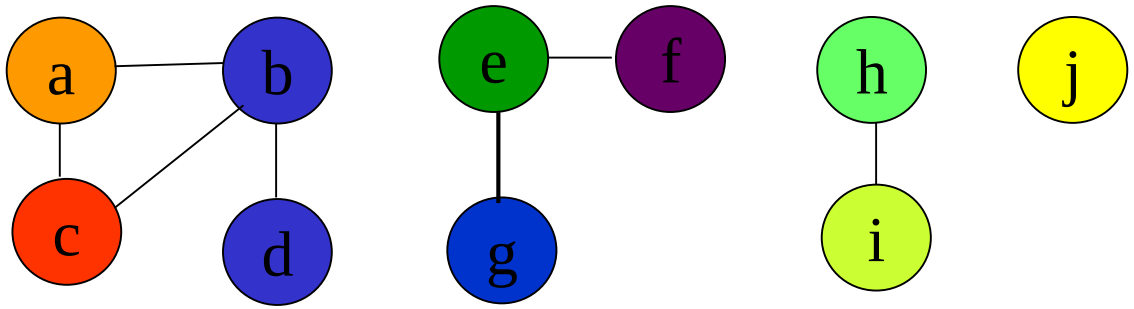
```
SAME-COMPONENT(u, v)
1.    if FIND-SET(u) == FIND-SET(v)
2.        return TRUE
3.    else return FALSE
```

Finding Connected Component of an Undirected Graph

- The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set.
- Then, for each edge (u, v), it unites the sets containing u and v.
- After processing all the edges, two vertices are in the same connected component if and only if the corresponding objects are in the same set.
- Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component.

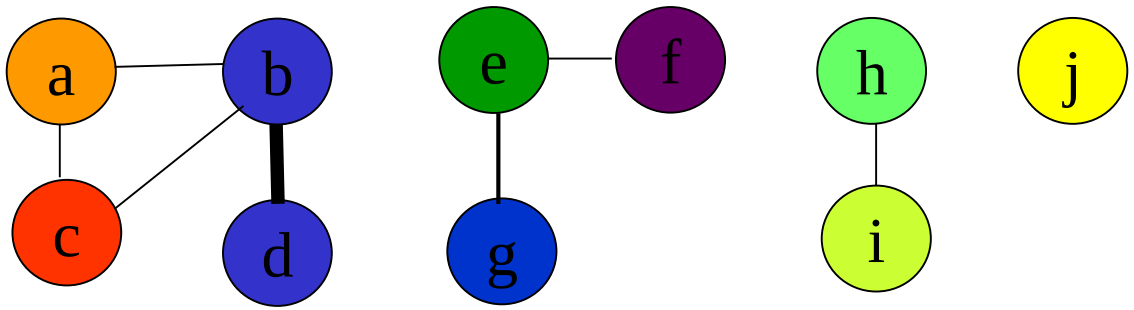
An Example of a Graph with 4 Connected Components

- Initially, each element is a set in itself:
  - {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}



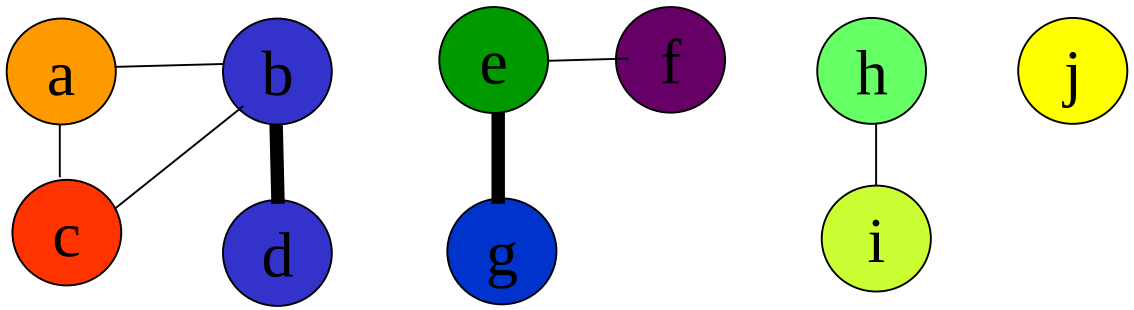
Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (b,d)
  - {a}, {b, d}, {c}, {e}, {f}, {g}, {h}, {i}, {j}



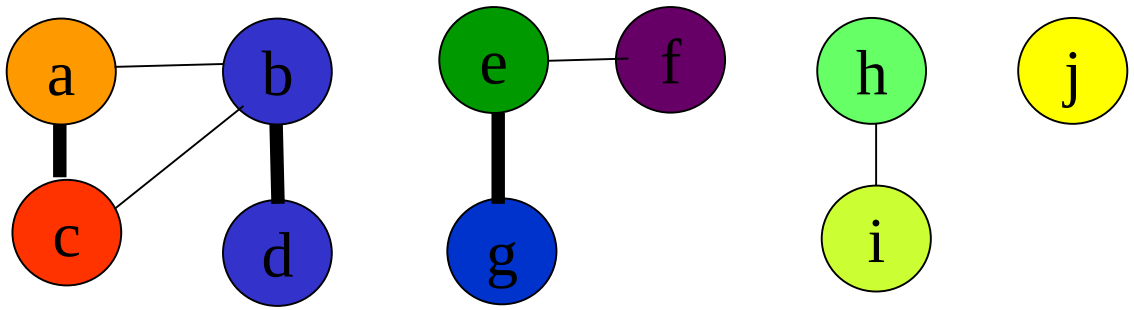
Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (e,g)
  - {a}, {b, d}, {c}, {e, g}, {f}, {h}, {i}, {j}



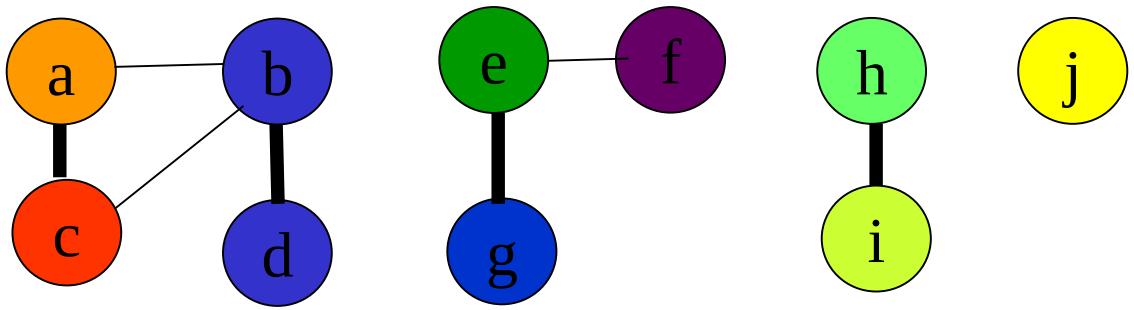
Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (a,c)
  - {a, c}, {b, d}, {e, g}, {f}, {h}, {i}, {j}



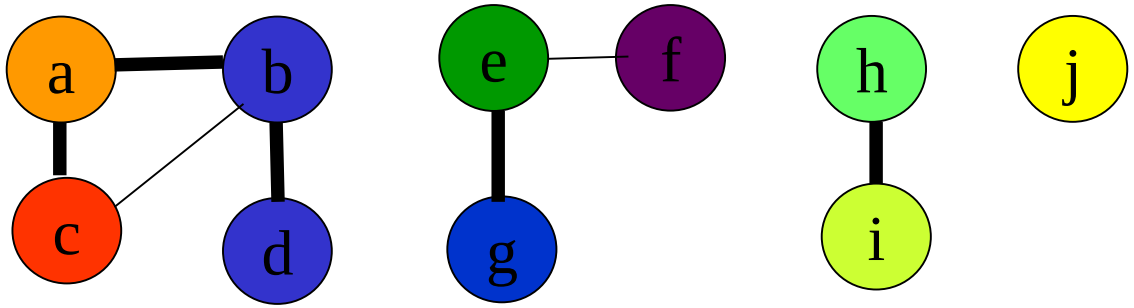
Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (h,i)
  - {a, c}, {b, d}, {e, g}, {f}, {h, i}, {j}



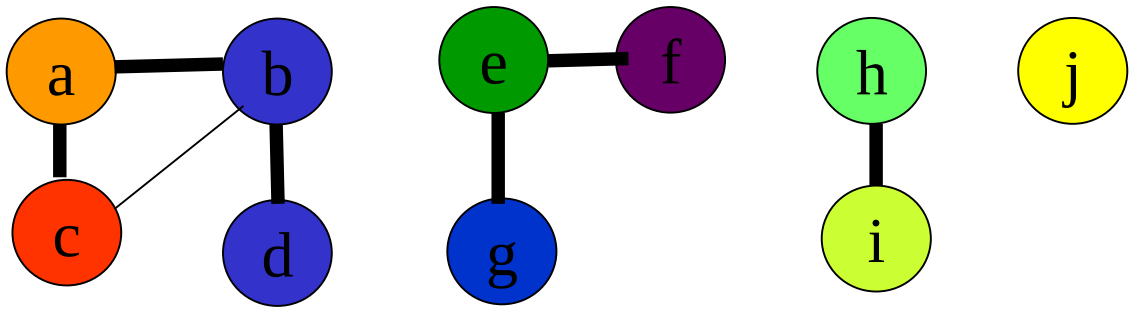
Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (a,b)
  - {a, b, c, d}, {e, g}, {f}, {h, i}, {j}



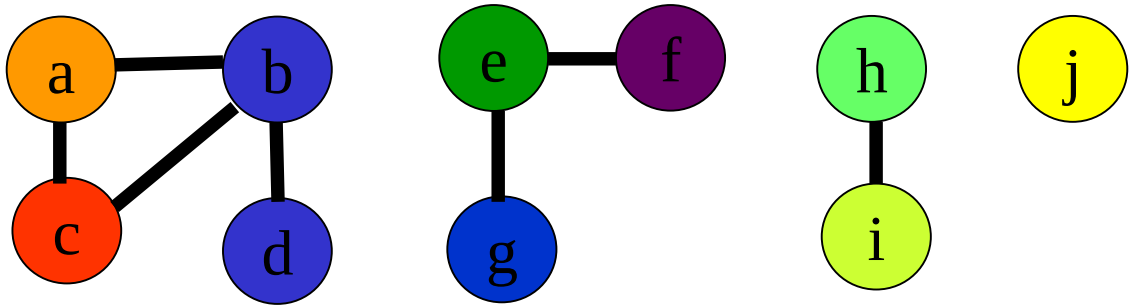
Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (e,f)
  - {a, b, c, d}, {e, f, g}, {h, i}, {j}

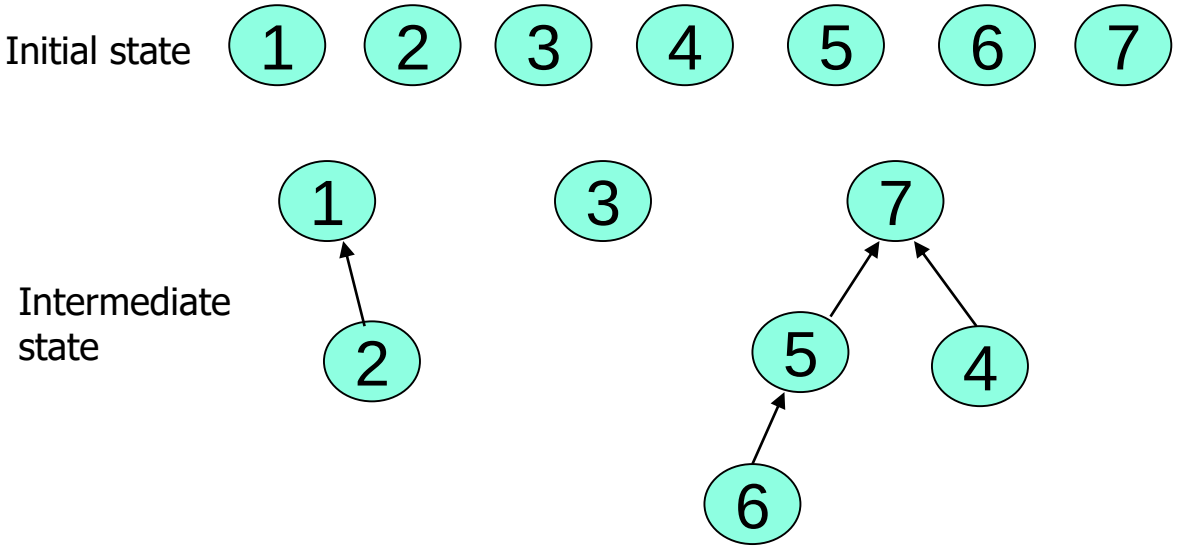


Collection of Disjoint Sets After Processing Each Edge

- Edge processed: (b,c)
  - {a, b, c, d}, {e, f, g}, {h, i}, {j}



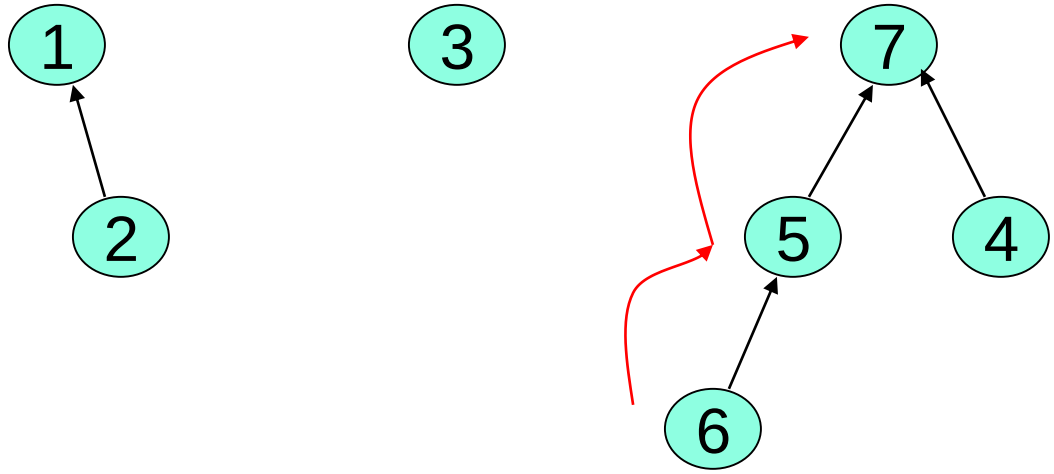
Forest for Disjoint Sets



Roots are the representatives of each set.

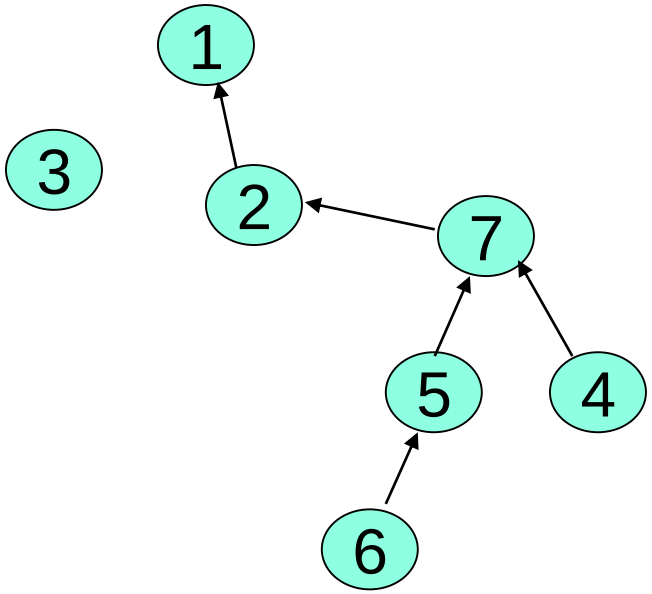
Find Operation

- FIND-SET(6) = 7

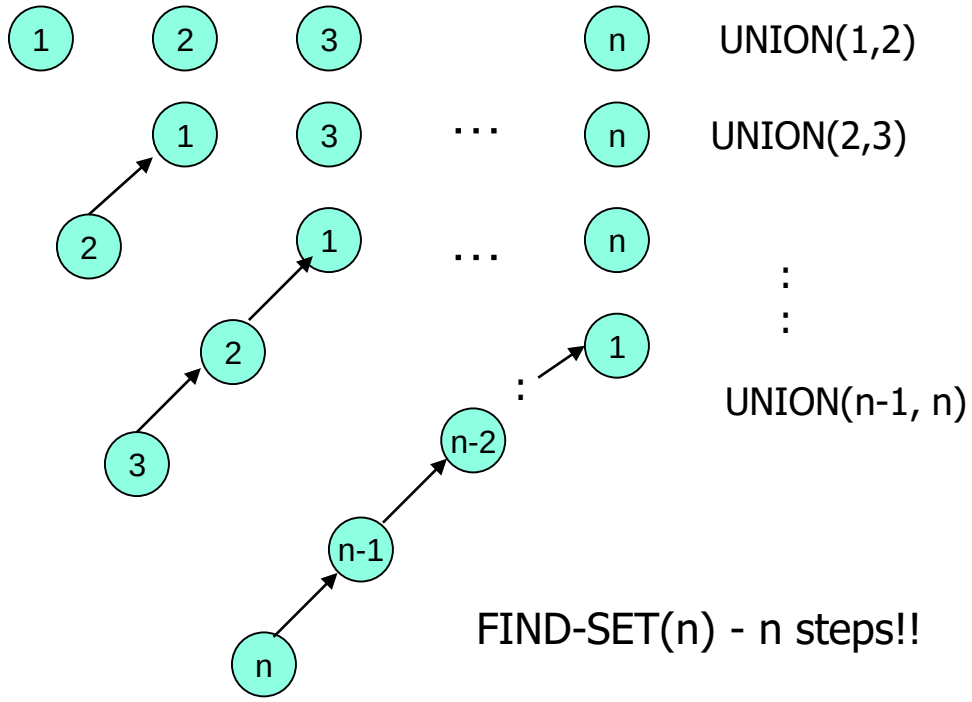


Union Operation

- UNION(1, 7)



A Worst Case


1
2
3
n
UNION(1,2)
1
3
n
UNION(2,3)
2
1
n
2
1
3
n-2
n-1
n
UNION(n-1, n)
FIND-SET(n) - n steps!!

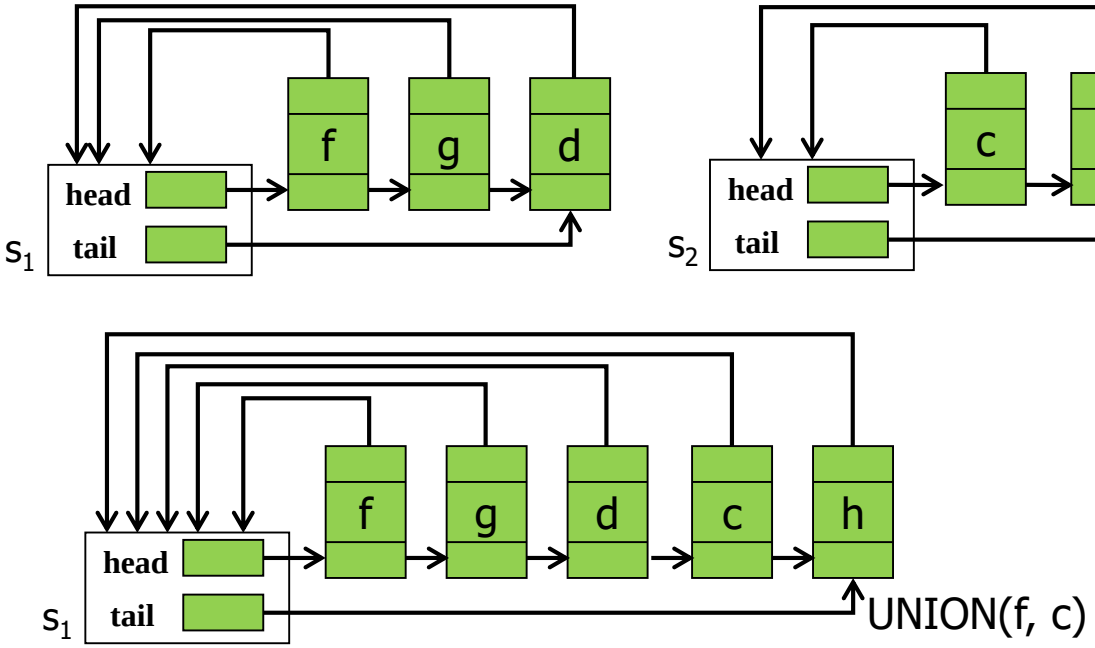
Linked-List Implementation

- Each set is represented by its own linked list.
- The object for each set has the following attributes
  - Head: pointing to the first object in the list
  - Tail: pointing to the last object in the list
- Each object in the list contains
  - A set member
  - A pointer to the next object in the list
  - A pointer back to the set object
- Within each linked list, the objects may appear in any order.
- The representative is the set member in the first object in the list.

Linked-List Implementation

- MAKE-SET(x) creates a new linked list whose only object is x. (O(1) time)
- FIND-SET(x) just follows the pointer from x back to its set object and then return the member in the object that head points to. (O(1) time)
- UNION(x, y) appends y's list onto the end of x's list.
  - The representative of x's list becomes the representative of the resulting set.
  - We use the tail pointer for x's list to quickly find where to append y's list.
  - Unfortunately, we must update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list.

Linked-lists for Two Sets



UNION Implementation

- Suppose that we have objects $x_1, x_2, \ldots, x_n$.
- Execute the sequence of n MAKE-SET operations followed by n-1 UNION operation.
- The total number of operations is m=2n-1.
- So 2n-1operations takes $\Theta(n^2)$ time. (i.e., $\Theta(n)$ time per operation on average.)
  - $\Theta(n)$ for n MAKE-SET operation
  - $\Theta(n^2)$ for n-1 UNION operations since the i-th UNION operation updates i objects. ($\sum_{i=1}^{n-1} i = \Theta(n^2)$)

Operation	# of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
…	…
MAKE-SET(x_n)	1
UNION(x_2,x_1)	1
UNION(x_3,x_2)	2
…	…
UNION(x_n,x_{n-1})	n-1

UNION Implementation

- In the worst case, the previous implementation of the UNION procedure requires an average of $\Theta(n)$ time per call.
  - When we append a longer list onto a shorter list, we must update the pointer to the set object for each member of the longer list.
- Weighted-Union Heuristic
  - Instead that each list also includes the length of the list (which we can easily maintain) and that we always append the shorter list onto the longer one.
  - With this simple weighted-union heuristic, a single UNION operation can still take $\Omega(n)$ time if both sets have $\Omega(n)$ members.

Weighted-Union Heuristic

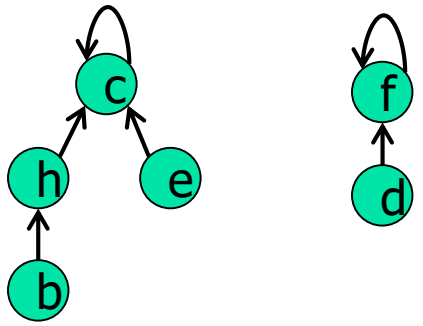
- Instead appending x to y, UNION(x, y) appends the shorter list to the longer list.
- Associated a length with each list, which indicates how many elements in the list.
- Theorem 21.1
  - Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes O(m+n lg n) time.

A Weighted-Union Heuristic

- Proof of Theorem 21.1
  - For an object x, each time x's pointer was updated, x must have started in the smaller set.
  - The first time x's pointer was updated, therefore, the resulting set must have had at least 2 members.
  - Similarly, the next time x's pointer was updated, the resulting set must have had at least 4 members.
  - Since the largest set has at most n members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations.
  - Thus, the total time spent updating object pointers over all UNION operations is O(n lg n).
  - We must also account for updating the tail pointers and the list lengths, which take only $\Theta(1)$ time per UNION operation.
  - The total time spent in all UNION operations is thus O(n lg n).
  - Each MAKE-SET and FIND-SET operation takes O(1) time, and there are O(m) of them.
  - The total time for the entire sequence is thus O(m+n lg n).

A Faster Implementation of Disjoint-set Forests

- We represent sets by rooted trees.
- Each member points only to its parent.
- The root of each tree contains the representative and is its own parent.
- The straightforward algorithms that use this representation are no faster than ones that use the linked-list representation.

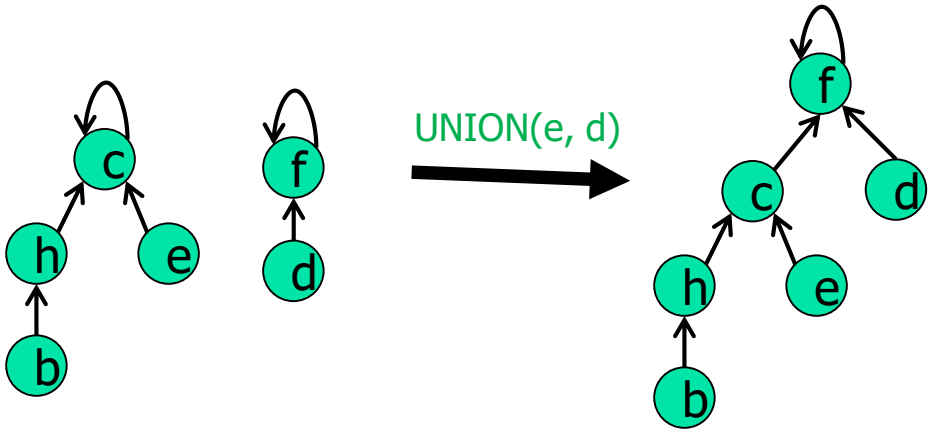


Straightforward Solution

- Three operations
  - MAKE-SET(x): Create a tree containing only x.
  - FIND-SET(x): Follow the chain of parent pointers until to the root. O(height of x's tree)
  - UNION(x, y): Let the root of one tree point to the root of the other.
- It is possible that (n-1) UNION operations results in a tree of height n-1. (just a linear chain of n nodes).
- So n FIND-SET operations cost $O(n^2)$.

Straightforward Solution

- A disjoint-set forest

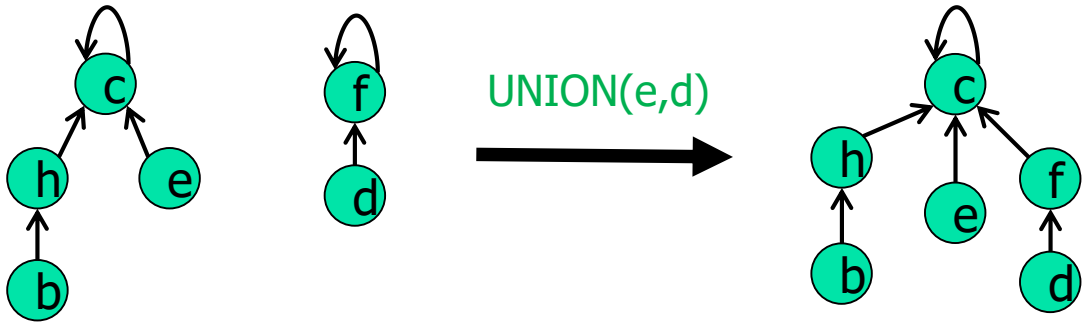


Disjoint-set Forests

- We can achieve an asymptotically optimal disjoint-set data structure by using the following heuristics
  - Union by rank
  - Path compression

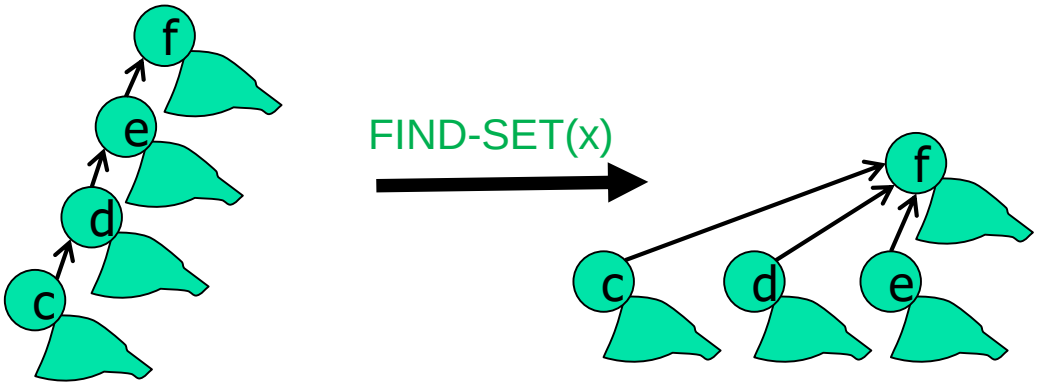
Union by Rank

- For each node, we maintain a rank, which is an upper bound on the height of the node.
- In union by rank, we make the root with smaller rank point to the root with larger rank during a UNION operation



Path Compression

- During FIND-SET operations, it makes each node on the find path to point directly to the root.
- Path compression does not change any ranks.

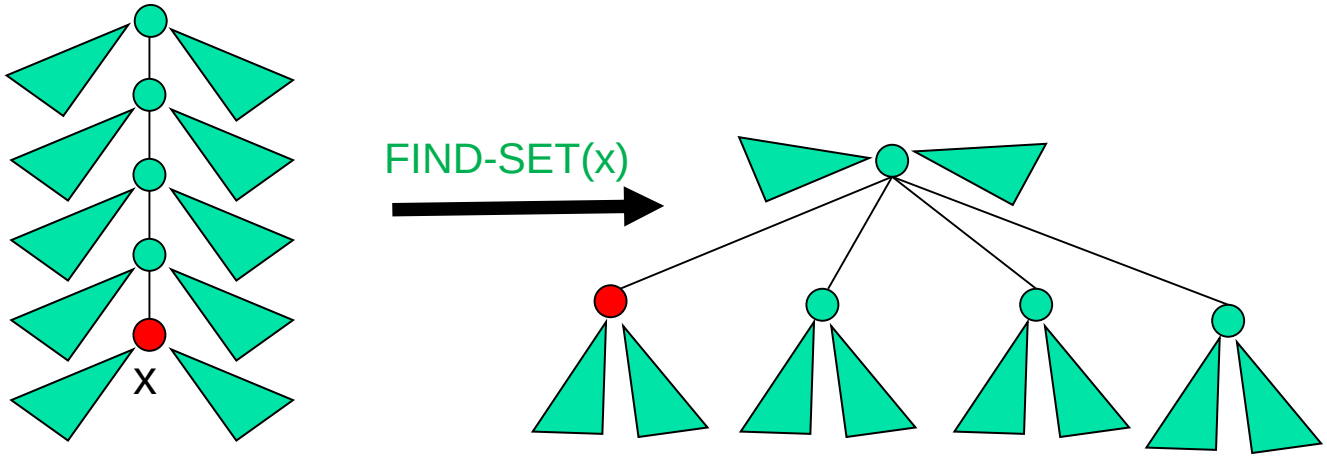


Before executing FIND-SET(c)

After executing FIND-SET(c)

Path Compression

- During FIND-SET operations, it makes each node on the find path to point directly to the root.
- Path compression does not change any ranks.



Pseudocodes for Disjoint-set Forests

- To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks.
- With each node x, we maintain the integer value x.rank, which is an upper bound on the height of x (the number of edges in the longest simple path between x and a descendant leaf).
- When MAKE-SET creates a singleton set, the single node in the corresponding tree has an initial rank of 0.
- Each FIND-SET operation leaves all ranks unchanged.
- The UNION operation has two cases, depending on whether the roots of the trees have equal rank.
  - If the roots have unequal rank, we make the root with higher rank the parent of the root with lower rank, but the ranks themselves remain unchanged.
  - If the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.
- We designate the parent of node x by x.p.

Pseudocodes for Disjoint-set Forests

```
MAKE-SET(x)
1.  x.p = x
2.  x.rank = 0

UNION(x,y)
1.  LINK(FIND-SET(x), FIND-SET(y))

LINK(x,y)
1.  if x.rank>y.rank
2.     y.p = x
3.  else x.p = y
4.       if x.rank == y.rank
5.          y.rank = y.rank + 1

FIND-SET(x)
1.  if x ≠ x.p
2.     x.p = FIND-SET(x.p)
3.  return x.p
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

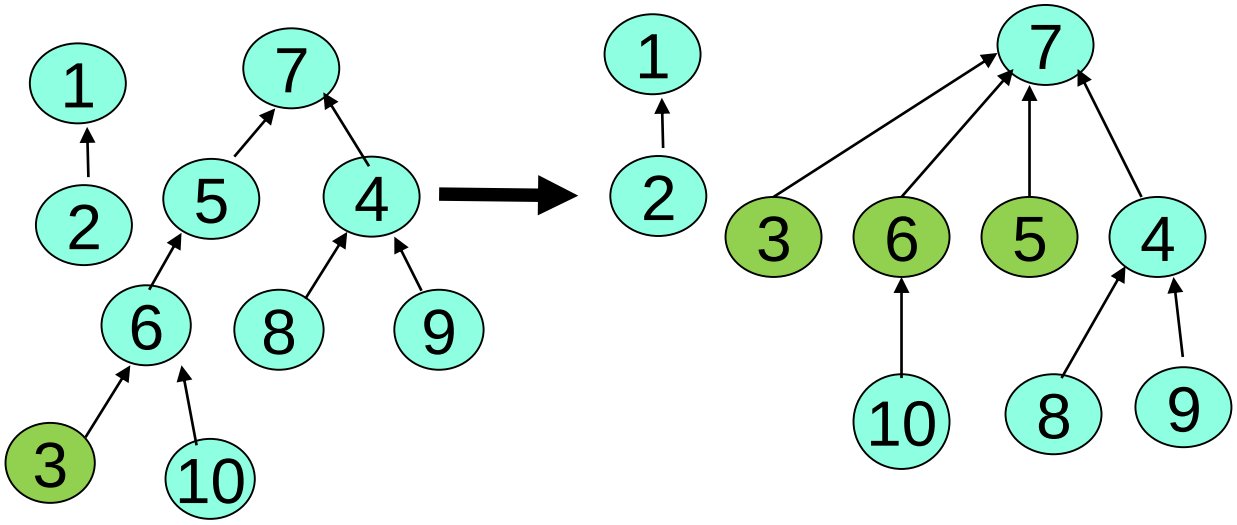
FIND-SET Procedure

- A two-pass method
  - As it recurses, it makes one pass up the find path to find the root.
  - As the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root.
- If x is the root, then FIND-SET skips line 2 and instead returns x.p, which is x.
- Otherwise, line 2 executes, and the recursive call with parameter x.p returns a pointer to the root.
  - Line 2 updates node x to point directly to the root, and line 3 returns this pointer.

```
FIND-SET(x)
1. if x ≠ x.p
2.     x.p = FIND-SET(x.p)
3. return x.p
```

Path Compression

- FIND-SET(3)

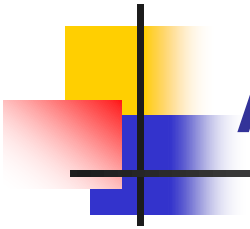


Effect of the Heuristics on the Running Time

- Separately, either union by rank or path compression improves the running time of the operations on disjoint-set forests, and the improvement is even greater when we use the two heuristics together.
- The union by rank heuristic alone yields a running time of O(m lg n) (see Exercise 21.4-4), and this bound is tight (see Exercise 21.3-3).
- Although we shall not prove it here, for a sequence of n MAKE-SET operations (and hence at most n-1 UNION operations) and f FIND-SET operations, the path-compression heuristic alone gives a worst-case running time of $\Theta(n + f(1 + \log_{2+f/n} n))$.

Effect of the Heuristics on the Running Time

- The union by rank heuristic alone yields a running time of O(m lg n) (see Exercise 21.4-4), and this bound is tight (see Exercise 21.3-3).
- For a sequence of n MAKE-SET operations (and hence at most n-1 UNION operations) and f FIND-SET operations, the path-compression heuristic alone gives a worst-case running time of $\Theta(n + f(1 + \log_{2+f/n} n))$.
- When we use both union by rank and path compression, for a sequence of m MAKE-SET, UNION, FIND-SET operations with n MAKE-SET operations, the worst-case running time is $O(m\,\alpha(n))$,
  - $\alpha(n)$ is a very slowly growing function which we define in Section 21.4.
  - In any conceivable application of a disjoint-set data structure, $\alpha(n) \leq 4$.
  - Thus, we can view the running time as linear in m in all practical situations.



Any Question?