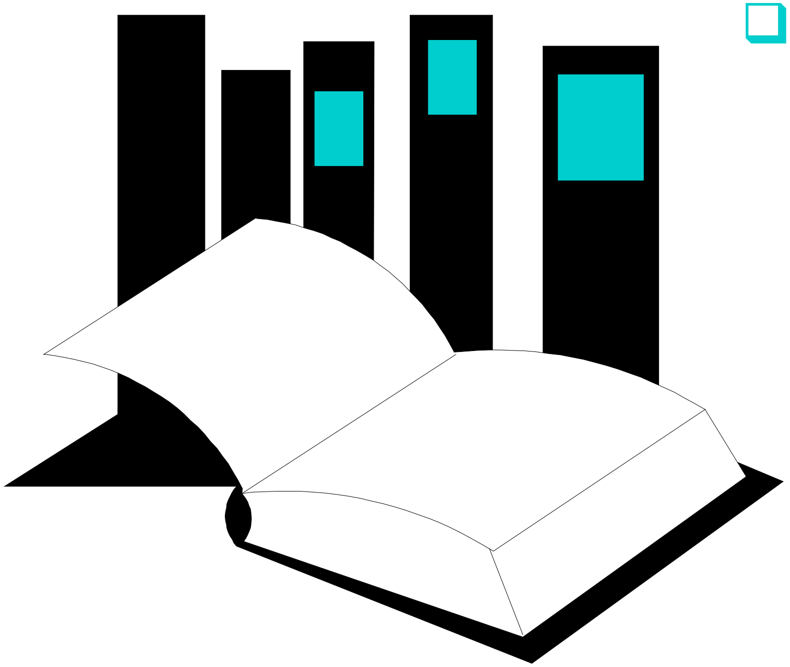
Chapter 4. The Greedy Approach



CHAPTER 4

Foundations of Algorithms

- Chapter 4 introduces an algorithm design technique called "***Greedy Approach***".

Greedy Algorithm

- ***Greedy Algorithm***

  arrives at a solution by making a ***sequence of choices***, each of which simply ***looks the best at the moment***

  \- ***Once*** each choice has been made and ***added*** to a partial solution, it ***will always*** be ***in*** the solution set.

  \- Each choice is locally optimal, ***but not necessarily globally optimal*.**

Greedy Algorithm

- Example: Giving Change for a Purchase

Solution: a smallest number of bills/coins totaling the amount of change

₩7,870 : one ₩5,000 bill,
two ₩1,000 bills,
one ₩500 coin,
three ₩100 coins,
one ₩50 coin,
two ₩10 coins.

Greedy Algorithm

- Example: Giving Change for a Purchase

```
while (there are more bills/coins and the instance is not solved)
{
    grab the largest remaining bill;  //selection  procedure
    if (adding the bill/coin makes the change exceed the amount owed)
        // feasibility check
        reject the bill/coin ;
    else
        add the bill/coin to the change ;
    if (the total value of the change equals the amount owed)
        // solution check
        the instance is solved ;
}
```

Greedy Algorithm

- General Greedy Approach

- starts with an empty set, and adds items to the set in sequence until the set represents a solution to an instance of a problem

- each iteration consists of the following components:

1. ***Selection Procedure***

   chooses the next item to add to the set according to a greedy criteria

2. ***Feasibility Check***

   checks whether it is possible to complete the set in such a way as to give a solution to the instance.

3. ***Solution Check***

   determines whether the new set constitutes a solution to the instance.

4.1 Minimum Spanning Trees

- Definitions
  - ***Undirected Graph***

    - a graph where its edges do not have directions
  - ***Connected Graph***

    - an undirected graph where there is a path between every pair of vertices
  - ***Acyclic Graph***

    - a graph with no cycles
  - ***Tree***

    - an acyclic, connected, undirected graph

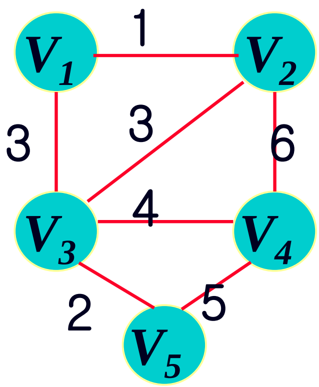
4.1 Minimum Spanning Trees

- Definitions
  - ***Spanning Tree*** for undirected graph
    - a connected subgraph that contains all the vertices in G and is a tree
  - ***Minimum Spanning Tree*** for undirected graph
    - a spanning tree of G with ***minimum weight***

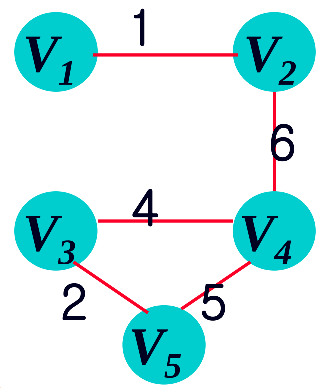
4.1 Minimum Spanning Trees

- Examples of Graph

G_1:

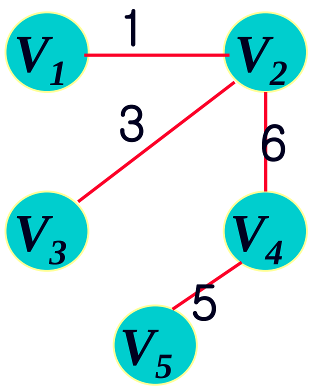


G_2:

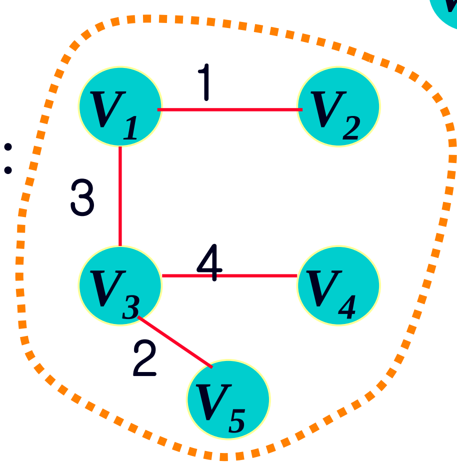


G_4,G_5: minimum spanning trees of G_1

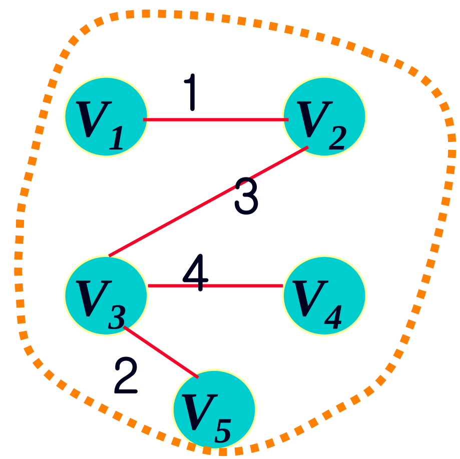
G_3:



G_4:



G_5:



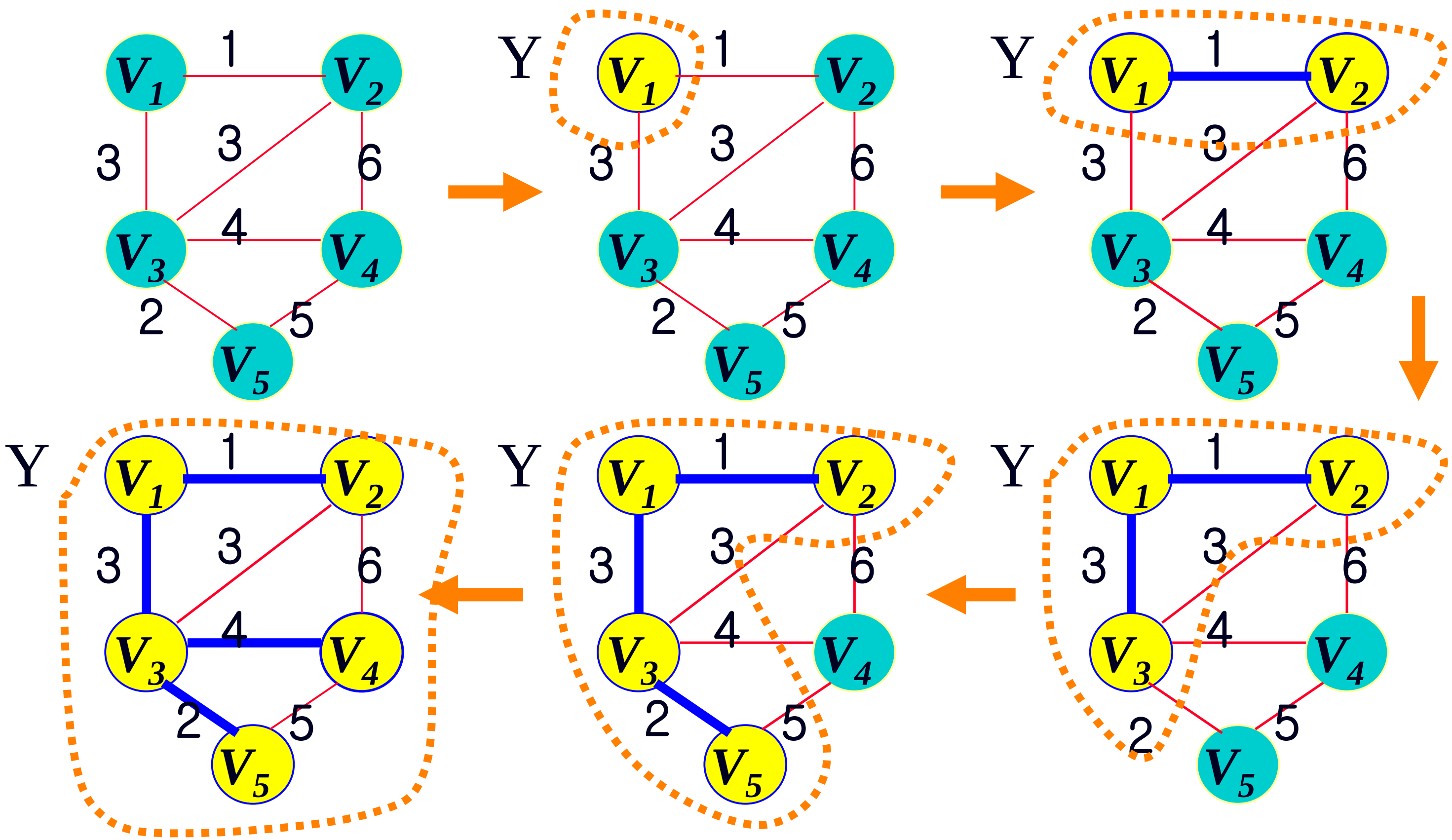
4.1 Minimum Spanning Trees

4.1.1 Prim's Algorithm

```
F = Ø ;
Y = {v1}
while (the instance is not solved)
{
   select a vertex in V-Y that is nearest to Y ;

   add the vertex to Y ;
   add the edge to F ;

   if (Y==V)
            the instance is solved ;
}
```

4.1 Minimum Spanning Trees

4.1.1 Prim's Algorithm



4.1 Minimum Spanning Trees

4.1.1 Prim's Algorithm

- To find the nearest vertex from Y, we use the following arrays:

  W[*i*][*j*] : Weight table (symmetric)

  nearest[*i*] = index of the vertex in Y nearest to V_i

  distance[*i*] = weight on edge between V_i and the vertex indexed by nearest[i]

- nearest[i] is initialized to 1
- distance[i] is initialized to *W*[*1*][*i*]
- as vertices are added to Y, two arrays are updated to reference the new vertex in Y nearest to each vertex outside Y

4.1 Minimum Spanning Trees

4.1.1 Prim's Algorithm

```
public static set_of_edges prim(
                int n, const number W[][])
{
   index i, vnear;
   index [ ] nearest = new index[2..n] ;
   number min;
   number[ ] distance = new number[2..n] ;
   edge e;
   set_of_edges  F = Ø;

   for (i=2; i<=n; i++)  {
        nearest[i]= 1;
        distance[i] = W[1][i];
   }
   repeat( n-1 times ) { . . . }
   return F ;
}
```

```
min = ∞ ;
for (i=2; i<=n; i++)
   if (0<=  distance[i] < min) {
         min = distance[i] ;  vnear = i ;
   }
e = edge connecting vertices indexed
        by vnear and nearest[vnear] ;
add e to F ;
distance[vnear] = -1 ;
for (i=2; i<=n; i++)
     if (W[i][vnear] < distance[i]) {
          distance[i] = W[i][vnear] ;
          nearest[i] = vnear ;
     }
```

4.1 Minimum Spanning Trees

- Time Complexity of Prim's Algorithm

Basic Operation:

if-statement inside two for-i loops

Input Size:

n, the number of vertices

➔ $\mathbf{T(n) = 2*(n-1)*(n-1) \in \Theta(n^2)}$

4.1 Minimum Spanning Trees

- Does Prim's Algorithm always produce an optimal solution?

  - Although greedy algorithms are often ***easier to develop*** than dynamic programming algorithms, usually it is more ***difficult to determine whether or not*** a greedy algorithm always produces ***an optimal solution*.**

  ➔ ***For a greedy algorithm, we usually need a formal proof to show that it actually does.***

4.1 Minimum Spanning Trees

Optimality of Prim's Algorithm

- We will use the concept of "***a promising*** set of edges" to prove the optimality of Prim's Algorithm.

Definition

- A subset F of E is called ***promising*** if edges can be added to it so as to form a minimum spanning tree.

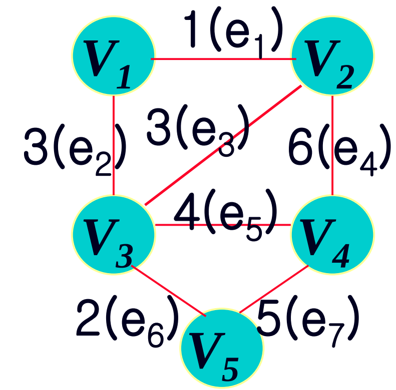
Proof:

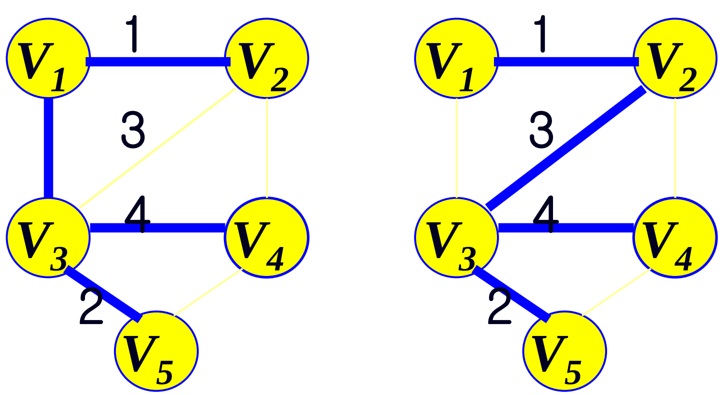
- Will show by induction that the set F is promising after each iteration of the repeat loop.

4.1 Minimum Spanning Trees

- Example:

G:





2 minimum spanning trees of G

1. **F = { }** ➔ ***promising***
2. **F = {e_4}** ➔ ***Not promising***
3. **F = {e_1,e_2}** ➔ ***promising***
4. **F = {e_1,e_2,e_3 }** ➔ ***Not promising***
5. **F = {e_1,e_2,e_5}** ➔ ***promising***
6. **F = {e_1,e_2,e_5,e_6}** ➔ ***promising***

4.1 Minimum Spanning Trees

- Proof of the Optimality of Prim's Algorithm
  - **Induction Basis**

    Obviously, the empty set is ***promising*****.**
  - **Induction Hypothesis**

    Assume that, after a given iteration of the repeat loop, the set of edges so far selected, namely F, is ***promising*****.**
  - **Induction Step**

    We need to show that $F \cup \{e\}$ is ***promising***, where e is an edge of minimum weight that connects a vertex in Y to a vertex in V-Y.

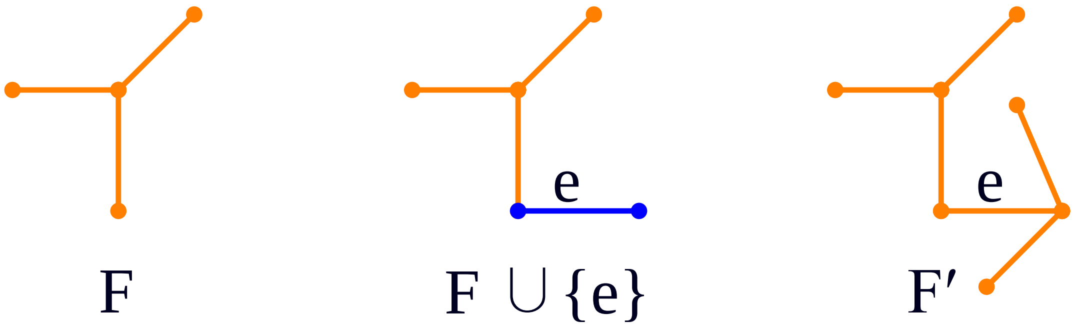
4.1 Minimum Spanning Trees

- Proof of the Optimality of Prim's Algorithm
  - **Induction Step - continued**

    Because F is promising, there must be some set of edges F′ such that $F \subseteq F'$ and (V,F′) is a minimum spanning tree.

  Case 1: $\mathbf{e} \in F'$

    $F \cup \{e\} \subseteq F'$, so $F \cup \{e\}$ is promising. Done.



4.1 Minimum Spanning Trees

- Proof of the Optimality of Prim's Algorithm
  - **Induction Step - continued**

Case 2: **e** $\notin F'$

Because (V,F') is a spanning tree, $F' \cup \{e\}$ must contain exactly one cycle and e must be in the cycle. Thus, there must be another edge $e' \in F'$ in the cycle that also connects a vertex in Y to one in V-Y.

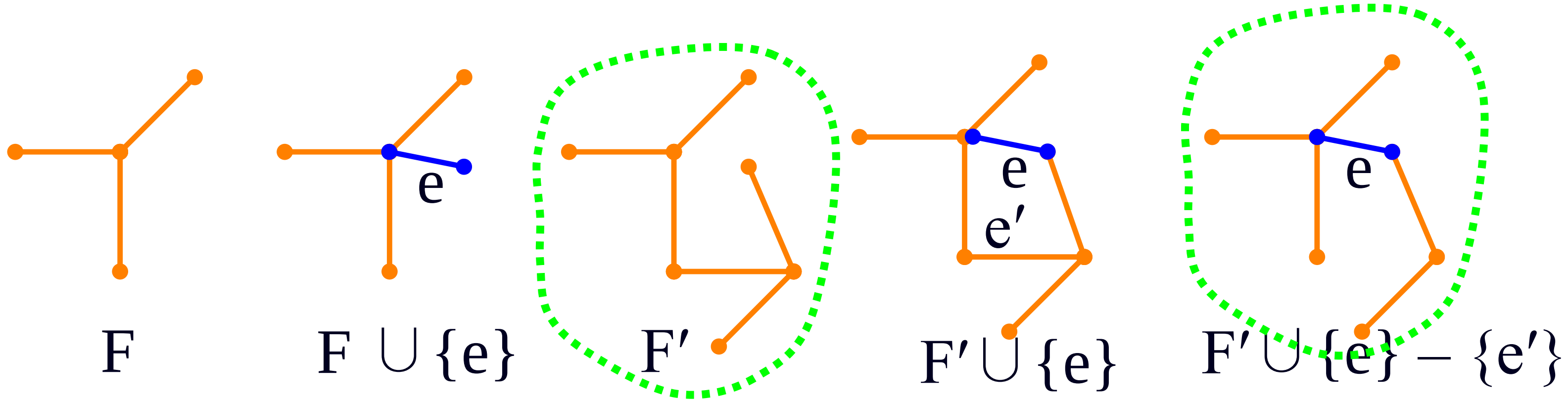
If we remove e' from $F' \cup \{e\}$, the cycle disappears, which means we have a spanning tree. Because e is an edge of minimum weight that connects a vertex in Y to one in V-Y, the weight of e must be less than or equal to that of e' (***in fact they must be equal***).

4.1 Minimum Spanning Trees

- Proof of the Optimality of Prim's Algorithm
  - **Induction Step**

  Case 2: e $\notin$ F′ - continued

  So, $F' \cup \{e\} - \{e'\}$ is a minimum spanning tree.
  Therefore, $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$,
  which means $F \cup \{e\}$ is ***promising***.



4.1 Minimum Spanning Trees

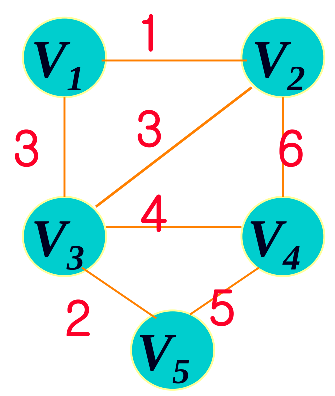
4.1.2 Kruskal's Algorithm

```
1. F = Ø ;
2. create disjoint subsets of V, one for each vertex and containing only
     that vertex;
3. sort the edges in E in non-decreasing order ;
4. while (the instance is not solved)    {
        select next edge from the sorted list
        if (the edge connects two vertices in disjoint subsets)  {
              merge the subsets ;
              add the edge to F ;
        }
        if (all the subsets are merged)
              the instance is solved ;
   }
```

4.1 Minimum Spanning Trees

4.1.2 Kruskal's Algorithm

- **Example:**



1. Sorted Edges:

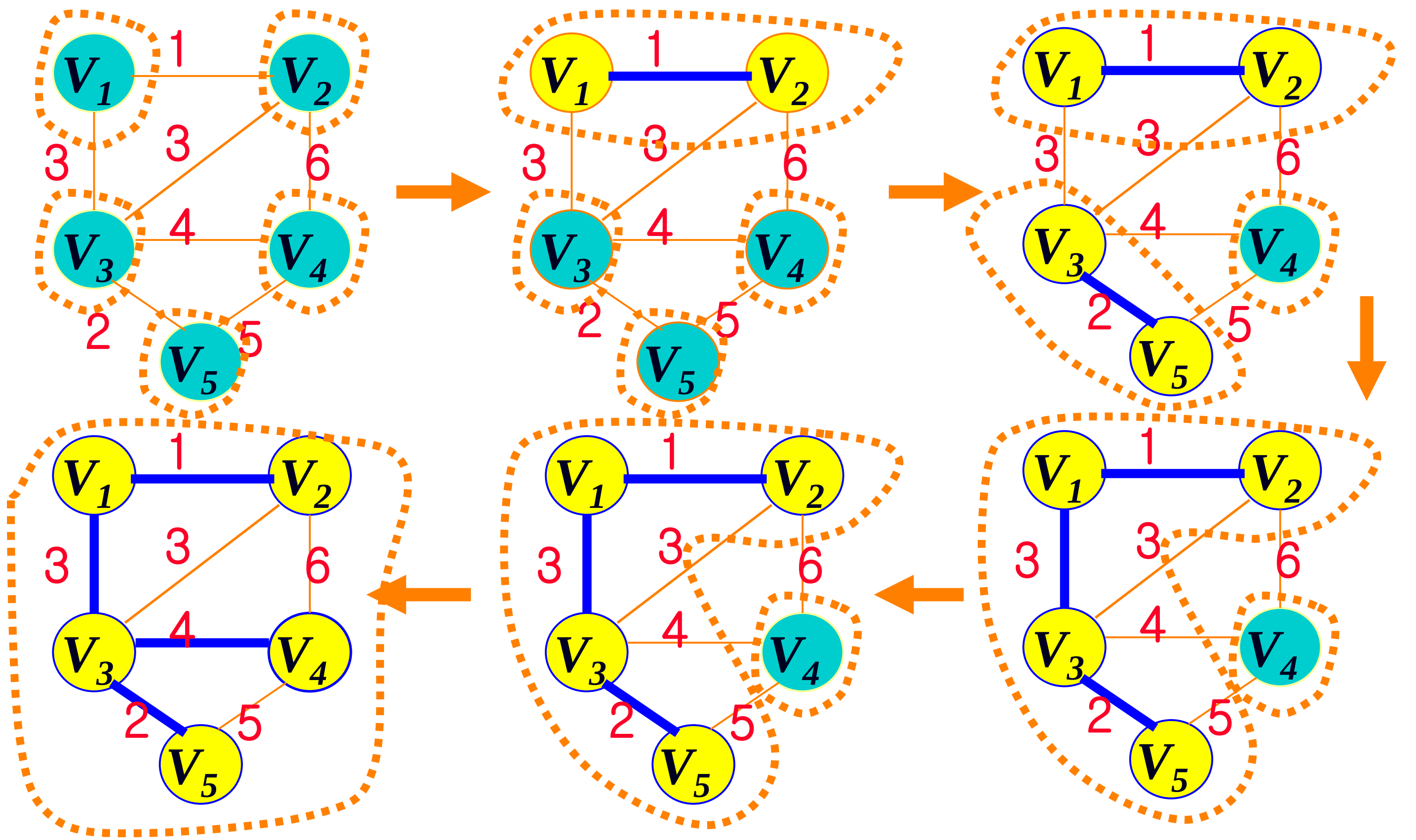
(v_1,v_2)	1
(v_3,v_5)	2
(v_1,v_3)	3
(v_2,v_3)	3
(v_3,v_4)	4
(v_4,v_5)	5
(v_2,v_4)	6

4.1 Minimum Spanning Trees

4.1.2 Kruskal's Algorithm

Sorted Edges

(v_1,v_2)	1
(v_3,v_5)	2
(v_1,v_3)	3
(v_2,v_3)	3
(v_3,v_4)	4
(v_4,v_5)	5
(v_2,v_4)	6



4.1 Minimum Spanning Trees

4.1.2 Kruskal's Algorithm

- To write a formal version of Kruskal's Algorithm, we use the following:

  - ***Initial(n)*** initializes n disjoint subsets, each of which contains exactly one of the indices between 1 and n
  - ***p= find(i)*** makes p point to the set containing index i
  - ***merge(p,q)*** merges the two sets, to which p and q point, into a set.
  - ***equal(p,q)*** returns true if both p and q point to the same set.

4.1 Minimum Spanning Trees

4.1.2 Kruskal's Algorithm

```
public static set_of_edges kruskal(
            int n, int m, set_of_edges E) {
  index i, j;
  set_pointer p,q ;
  edge e;
  set_of_edges  F = Ø ;

  Sort the m edges in E by weight
       in nondecreasing order;
  initial(n) ;
  while( |F| <  n-1)
  {      . . .        }
  return F ;
}
```

```
e = edge with the least weight
        not yet considered ;
i,j = indices of vertices connected by e;

p = find(i) ;
q = find(j) ;
if (! Equal(p,q)) {
      merge(p,q) ;
      add e to F ;
}
```

4.1 Minimum Spanning Trees

- Time Complexity of Kruskal's Algorithm

Basic Operation:

A comparison operation

Input Size:

n, the number of vertices

m, the number of edges

1. *Time to sort the edges:*

$W(m) \in \Theta(m \lg m)$ using MergeSort

2. *Time in the while loop:*

$W(m) \in \Theta(m \lg m)$ using DisjointSet implementation

4.1 Minimum Spanning Trees

- Time Complexity of Kruskal's Algorithm

  3. Time to initialize n disjoint sets

  $$T(n) \in \Theta(n)$$

  ➔ $W(m,n) \in \Theta(m \lg m)$ where $n-1 \leq m \leq n(n-1)/2$.

  ➔ Thus, in the worst case, we have

  $$W(m,n) \in \Theta(m \lg m) = \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n).$$

4.1 Minimum Spanning Trees

- Optimality of Kruskal's Algorithm

  - Similar to the proof of Prim's Algorithm

  - Will use the concept of "***a promising*** set of edges" to prove the optimality

4.1 Minimum Spanning Trees

- Proof of the Optimality of Kruskal's Algorithm
  - **Induction Basis**

    Obviously, the empty set is ***promising.***
  - **Induction Hypothesis**

    Assume that, after a given iteration of the repeat loop, the set of edges so far selected, namely F, is ***promising.***
  - **Induction Step**

    We need to show that $F \cup \{e\}$ is ***promising***, where e is an edge of minimum weight in E-F such that $F \cup \{e\}$ has no cycles.

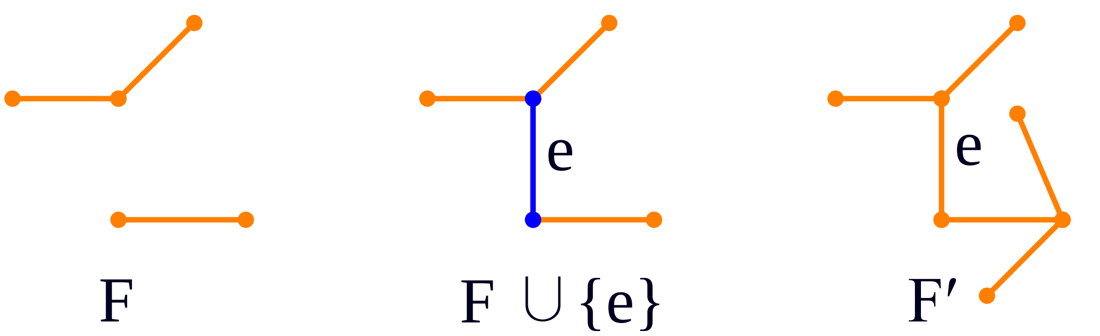
4.1 Minimum Spanning Trees

- Proof of the Optimality of Kruskal's Algorithm
  - **Induction Step - continued**

    Because F is promising, there must be some set of edges F′ such that F $\subseteq$ F′ and (V,F′) is a minimum spanning tree.

Case 1: e $\in$ F′

F $\cup$ {e} $\subseteq$ F′, so F $\cup$ {e} is promising. Done.



4.1 Minimum Spanning Trees

- Proof of the Optimality of Kruskal's Algorithm
  - **Induction Step - continued**

Case 2: e $\notin$ F′

Because (V,F′) is a spanning tree, F′ $\cup$ {e} must contain exactly one cycle and e must be in the cycle. Thus, there must be another edge e′$\in$ F′ in the cycle that is not in F. That is, e′$\in$ E – F.

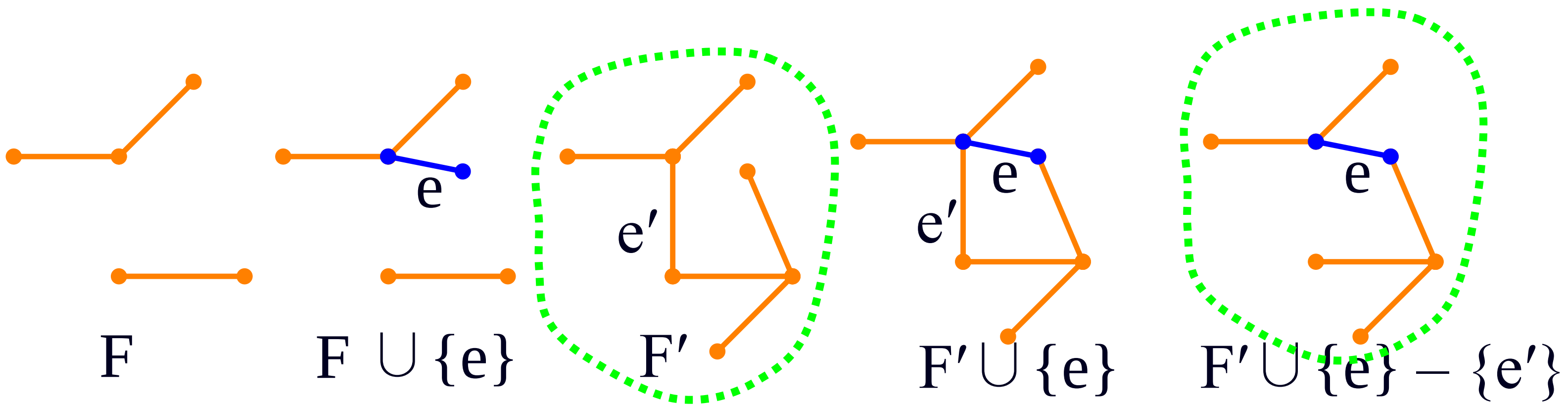
If we remove e′ from F′ $\cup$ {e}, the cycle disappears, which means we have a spanning tree. Because e is an edge of minimum weight that is not in F, the weight of e must be less than or equal to that of e′ (***in fact they must be equal***).

4.1 Minimum Spanning Trees

- Proof of the Optimality of Kruskal's Algorithm
  - **Induction Step**

  Case 2: e $\notin$ F′ - continued

  So, $F' \cup \{e\} - \{e'\}$ is a minimum spanning tree.
  Therefore, $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$,
  which means $F \cup \{e\}$ is promising.



4.1 Minimum Spanning Trees

4.1.3. Prim's Algorithm vs. Kruskal's Algorithm

- **Prim's Algorithm**

  $T(n) \in \boldsymbol{\Theta(n^2)}$

- **Kruskal's Algorithm**

  $W(m,n) \in \Theta(m \lg m)$ where $n-1 \le m \le n(n-1)/2$.

➔ 1. If m is close to n-1, i.e., the graph is ***sparse***, then $\Theta(m \lg m)$ becomes $\Theta(\boldsymbol{n \lg n})$, which is ***better than*** $\Theta(\boldsymbol{n^2})$.

➔ 2. If m is close to n(n-1)/2, i.e., the graph is ***dense***, then $\Theta(m \lg m)$ becomes $\Theta(\boldsymbol{n^2 \lg n})$, which is ***worse than*** $\Theta(\boldsymbol{n^2})$.

4.4 Huffman Code

- Huffman Code
  - an efficient encoding method for data compression
  - uses a ***variable-length binary code*** to represent a text file

Note:

- Variable-Length Binary Code
  - represents different characters using ***different*** numbers of bits
- Fixed-Length Binary Code
  - represents each character using the ***same*** number of bits

4.4 Huffman Code

- Fixed-Length vs. Variable-Length Binary Code

Example: Character Set { a, b, c },
String "ababcbbbc"

- Fixed-Length Binary Code

→ a: 00, b: 01, c: 11
ababcbbbc: 000100011101010111 - ***18*** bits

- Variable-Length Binary Code

→ a: 10, b: 0, c: 11
ababcbbbc: 1001001100011 - ***13*** bits

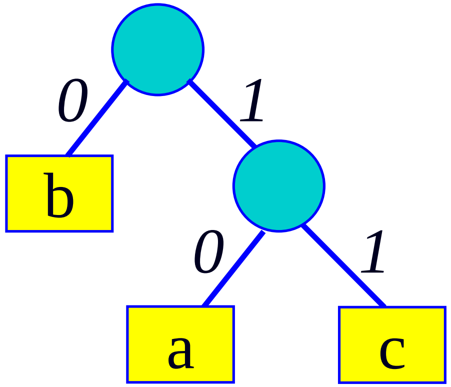
4.4 Huffman Code

4.4.1 Prefix Code

- a variable length-code in which ***no codeword*** for one character constitutes the ***beginning*** of the codeword for ***another character***

- **Example:**

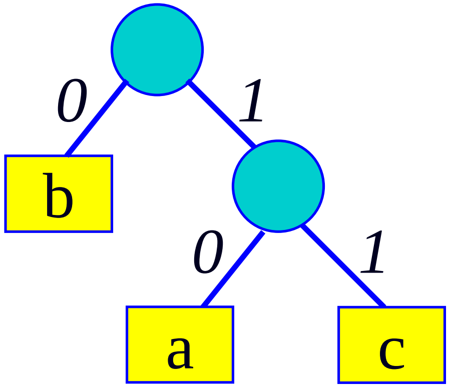
  ➔ a: 10, b: 0, c: 11



4.4 Huffman Code

- The number of bits to encode a text file
  - Given the binary tree *T* corresponding to some prefix code,

$$\text{minimize} \sum_{i=1}^{n} \text{frequency}(v_i) * \text{depth}(v_i)$$



frequency(b) * 1
+ frequency(a) * 2
+ frequency(c) * 2

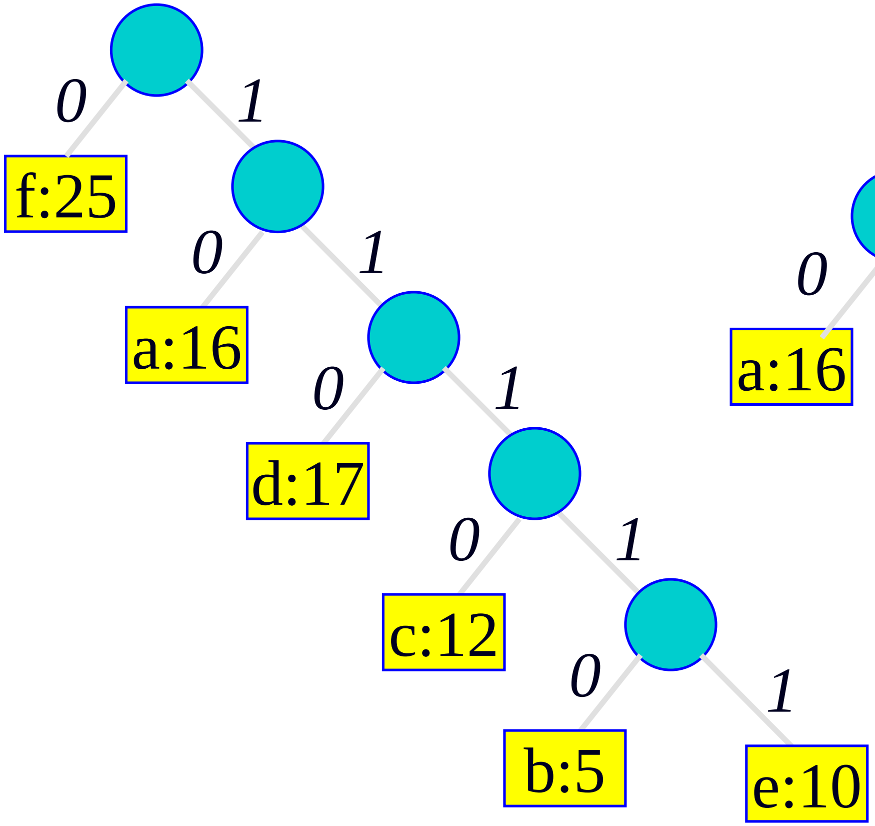
4.4 Huffman Code

Character	Frequency	Code 1 (Fixed)	Code 2 (variable)	Code 3 (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

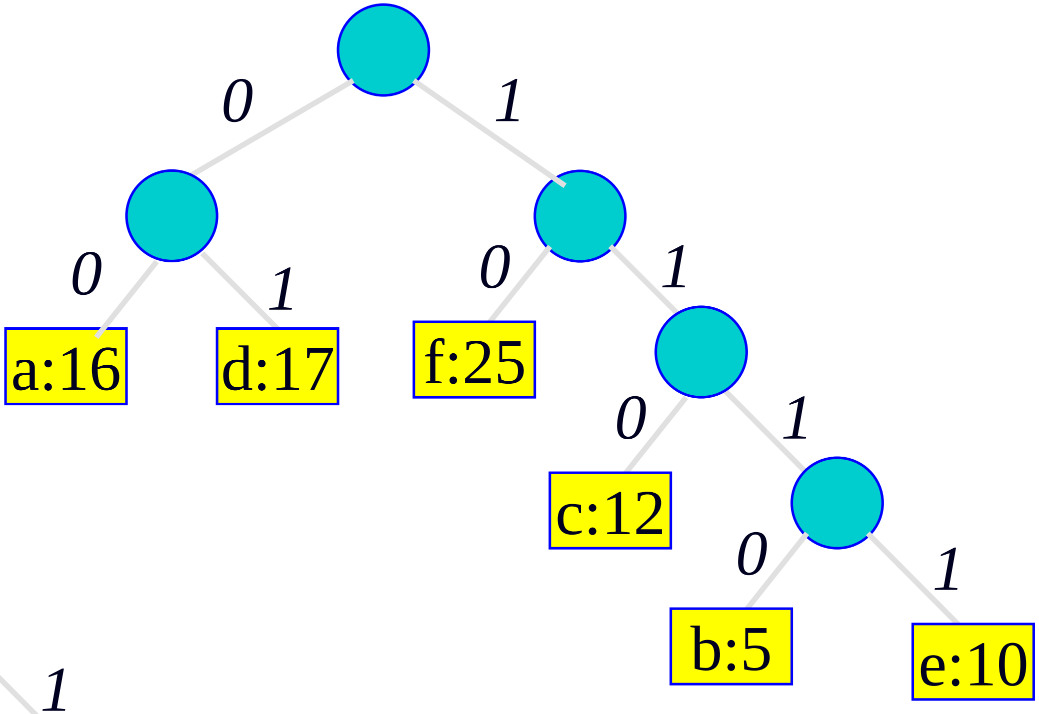
$\sum$ frequency(v) * depth(v) = *255* *231* ***212***

4.4 Huffman Code

Code 2



Code 3



4.4 Huffman Code

4.4.2 Huffman's Algorithm

Regard characters as a forest with *n* single-node trees
repeat
    merge two trees with least frequencies
until it becomes a single tree

4.4 Huffman Code

4.4.2 Huffman's Algorithm

```
for (i = 0; i < n; i++) {
        remove(PQ, p);
        remove(PQ, q);
        r = new nodetype();
        r.left = p;
        r.right = q;
        r.frequency = p.frequency + q.frequency;
        insert(PQ, r);
}
remove(PQ, r);
return r;
```

Min priority queue

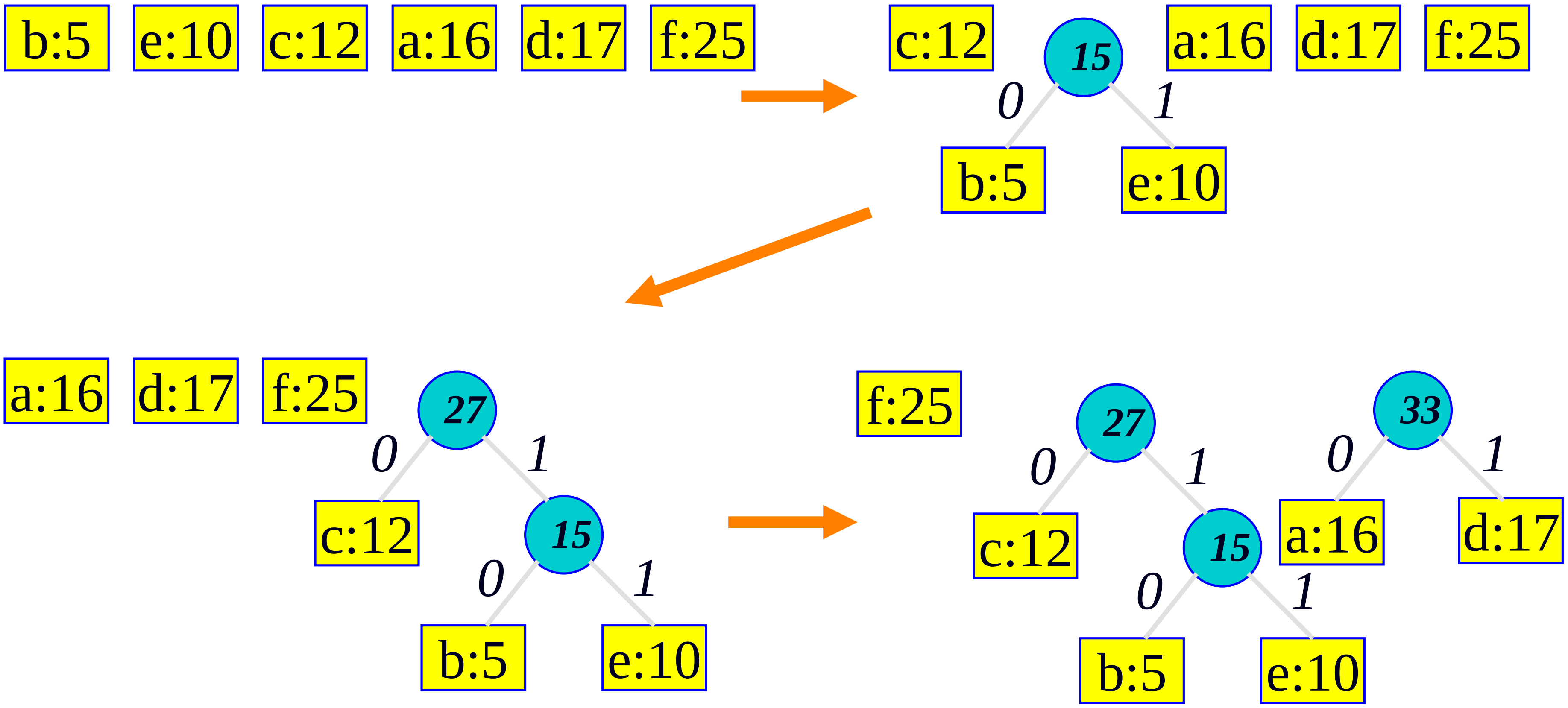
```
Public class nodetype
{
        char symbol;
        int frequency;

        nodetype left;
        nodetype right;
}
```

Time complexity

O(*n* log *n*)

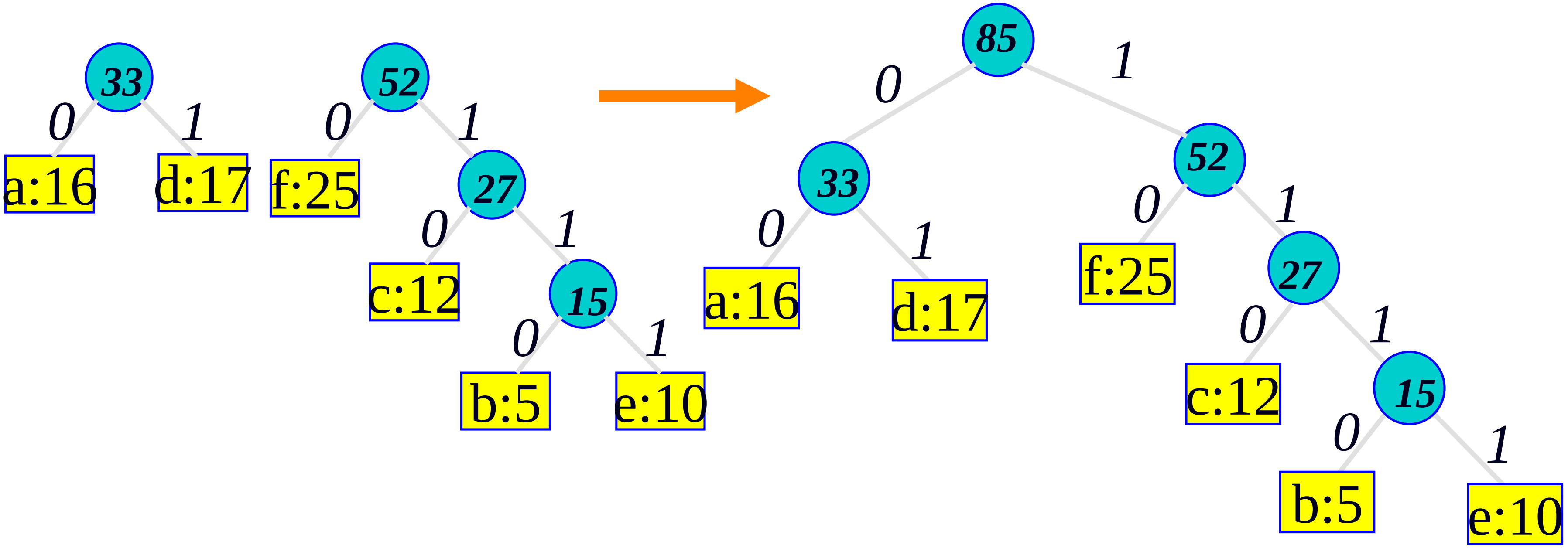
4.4 Huffman Code

- Example



4.4 Huffman Code

- Example (cont’d)



4.4 Huffman Code

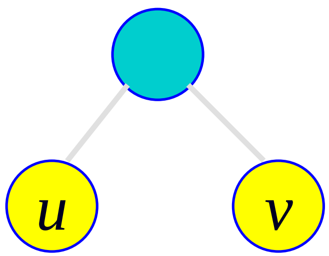
- Proof of the Optimality of Huffman's Algorithm
  - **Induction Basis**
    - The set of single nodes in the 0^{th} step

      → branches in an *optimal prefix binary tree*
  - **Induction Hypothesis**
    - The set of trees in the i^{th} step

      → branches in an optimal prefix binary tree T
  - **Induction Step**
    - u & v: roots of trees combined in the $(i+1)^{th}$ step
    - NEXT PAGE..

4.4 Huffman Code

- Proof of the Optimality of Huffman's Algorithm
  - **Induction Step – continued**

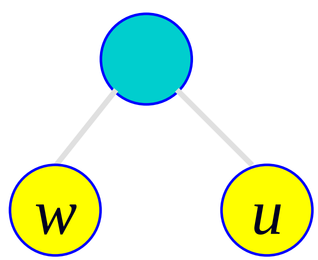
Case 1: 

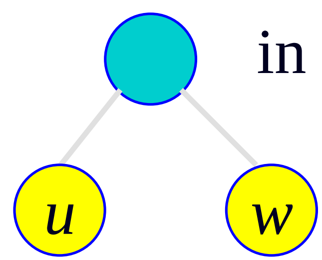
in *T*

Done.

Case 2: parent of *u* ≠ parent of *v* in *T*

WLOG, depth(*u*) ≥ depth(*v*) in *T*

There exists *w* such that 

or 

in *T*

4.4 Huffman Code

- Proof of the Optimality of Huffman's Algorithm
  - **Induction Step – continued**

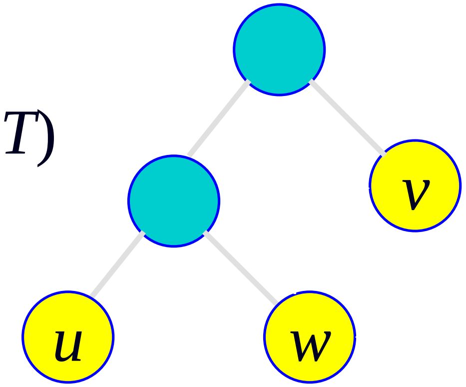
frequency(w) $\geq$ frequency(v) – *why?*

depth(w) $\geq$ depth(v) in T

Create a new tree T' by swapping the positions of the branches rooted at v & w.

cost(T') = cost(T) + (depth(w) – depth(v)) *
(frequency(v) – frequency(w)) $\leq$ cost(T)

Hence, T' is optimal.



4.5 Knapsack Problem

- A Greedy Approach to the 0-1 Knapsack Problem
  - **The 0-1 Knapsack Problem**

Given n items, let

$S = \{ item_1, item_2, \ldots, item_n \}$

w_i = weight of $item_i$

p_i = profit of $item_i$

W = maximum weight the knapsack can hold,

where w_i, p_i, and W are positive integers.

Determine a ***subset A*** of S such that

$\sum_{item_i \in A} p_i$ is ***maximized*** subject to $\sum_{item_i \in A} w_i \leq W$.

4.5 Knapsack Problem

- A Greedy Approach to the 0-1 Knapsack Problem
  - **A Brute-Force Solution to The 0-1 Knapsack Problem**

1. Consider all possible subsets of S.
2. Discard those subsets whose total weight > W.
3. Of those remaining, take one with the max total profit.

→ Since there are 2^n subsets containing up to n items, the complexity is ***exponential in n***.

4.5 Knapsack Problem

- A Greedy Approach to the 0-1 Knapsack Problem
  - **A Simple Greedy Approach**

    Idea: Take items in ***non-increasing order*** according to ***profit***.

    ➔ This approach wouldn't work very well if the most profitable item had a large weight in comparison with its profit.

    Example:

    $item_1$: \$1 M and 25 Kg
    $item_2$: \$0.8 M and 15 Kg
    $item_3$: \$0.5 M and 10 Kg
    $item_4$: \$0.24 M and 8 Kg
    maximum weight = 30 Kg

    ➔ **Greedy Solution:**
    $\{item_1\}$ (Profit : \$ 1M)

    ➔ **Optimal Solution:**
    $\{item_2, item_3\}$
    (Profit : \$ 1.3M)

4.5 Knapsack Problem

- A Greedy Approach to the 0-1 Knapsack Problem
  - **Another Simple Greedy Approach**

    Idea: Take items in ***non-decreasing order*** according to ***weight***.

    ➔ This approach would fail badly when the light items have small profits compared with their weights.

Example:

$item_1$: \$1 M and 25 Kg
$item_2$: \$0.8 M and 15 Kg
$item_3$: \$0.5 M and 10 Kg
$item_4$: \$0.24 M and 8 Kg
maximum weight = 30 Kg

➔ **Greedy Solution:**
$\{item_3, item_4\}$ (Profit: \$ 0.74M)

➔ **Optimal Solution:**
$\{item_2, item_3\}$ (Profit: \$ 1.3M)

4.5 Knapsack Problem

- A Greedy Approach to the 0-1 Knapsack Problem
  - **More Sophisticated Greedy Approach**

Idea: Take items in ***non-increasing order*** according to ***profit per unit weight***.

Example 1:

$item_1$: \$1 M and 25 Kg ➔ **\$40000/Kg**
$item_2$: \$0.8 M and 15 Kg ➔ **\$60000/Kg**
$item_3$: \$0.5 M and 10 Kg ➔ **\$50000/Kg**
$item_4$: \$0.24 M and 8 Kg ➔ **\$30000/Kg**
maximum weight = 30 Kg

➔ **Greedy Solution:** {$item_2$, $item_3$} (Profit: \$ 1.3M) ➔ **Optimal**

4.5 Knapsack Problem

- A Greedy Approach to the 0-1 Knapsack Problem
  - **More Sophisticated Greedy Approach**

Idea: Take items in ***non-increasing order*** according to ***profit per unit weight***.

Example 2:

$item_1$: \$5 M and 5 Kg ➔ **\$1 M/Kg**
$item_2$: \$6 M and 10 Kg ➔ **\$0.6 M/Kg**
$item_3$: \$14 M and 20 Kg ➔ **\$0.7 M/Kg**
maximum weight = 30 Kg

➔ **Greedy Solution:** {$item_1$, $item_3$} (Profit: \$ 19 M)
➔ **Optimal Solution:** {$item_2$, $item_3$} (Profit: \$ 20 M)

4.5 Knapsack Problem

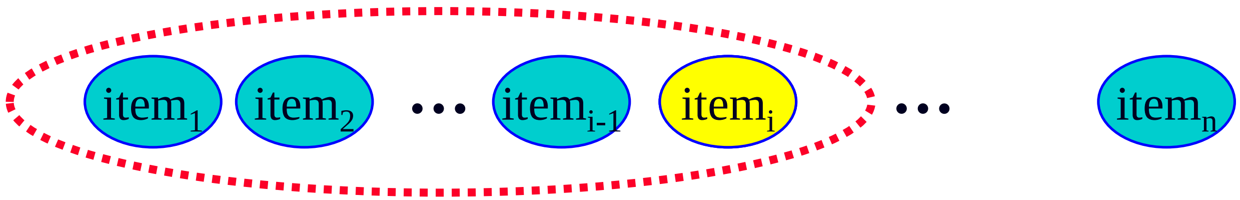
- A Greedy Approach to the ***fractional*** Knapsack Problem

  ➔ If we allow a ***fraction of item*** to be put in the knapsack, the greedy approach based on the ***profit per unit weight*** produces an ***optimal*** solution.

4.5 Knapsack Problem

- Dynamic Programming Approach to the 0-1 Knapsack Problem

Let ***P[i][w]*** be the optimal profit obtained from choosing items only from the first ***i*** items under the restriction that the ***total weight*** cannot exceed ***w***.



$$P[i][w] = \begin{cases} \max(\ P[i-1][w]\ ,\ p_i + P[i-1][w-w_i]\ )\ \text{if } w_i \le w \\ P[i-1][w] \qquad\qquad\qquad\qquad\qquad\qquad \text{if } w_i > w \end{cases}$$

4.5 Knapsack Problem

- Dynamic Programming Approach to the 0-1 Knapsack Problem

$$P[i][w] = \begin{cases} \max(\ P[i-1][w]\ ,\ p_i + P[i-1][w-w_i]\ ) & \text{if } w_i \le w \\ P[i-1][w] & \text{if } w_i > w \end{cases}$$

➔ The value we are looking for is P[n][W].

➔ We can determine this value using a two dimensional array P[0..n][0..W] where

$$P[0][w] = 0 \qquad 0 \le w \le W$$

$$P[i][0] = 0 \qquad 0 \le i \le n$$

4.5 Knapsack Problem

- Dynamic Programming Approach

$$P[i][w] = \begin{cases} \max(\ P[i-1][w]\ ,\ p_i + P[i-1][w-w_i]\ ) & \text{if } w_i \le w \\ P[i-1][w] & \text{if } w_i > w \end{cases}$$

P	0	1	2	…		W-w_n	…	W
0	0	0	0	0	0	0	0	0
1	0							
2	0							
…								
i								
n-1						?		?
n	0							?

➔ 2^{n-1} entries:

$P[1][W]$, $P[1][W-w_1]$, …

$P[1][W-w_n-w_{n-1}\ldots-w_3]$,

$P[1][(W-w_n-w_{n-1}\ldots-w_3)-w_2]$

➔ 2 entries: $P[n-1][W]$, $P[n-1][W-w_n]$

➔ 1 entry: $P[n][W]$

$\therefore\ 1 + 2 + 4 + \ldots + 2^{n-1} \in \Theta(2^n)$