## Chapter 4
# Greedy Approach

- ✓ 일정한 선택 규칙에 따라 순차적으로 Solution을 확장해 나간다.
- ✓ 일반적으로 Polynomial Time 알고리즘
- ✓ Solution이 Global Optimal인지는 별도로 증명해야



# Greedy Approach Algorithm

- ✓ **Basic Approach**
  - ✓ find local optimal solution
  - ✓ don't regard future/global optimal solution
- ✓ **Comparison to Dynamic Programming**
  - ✓ often used for optimization problems (as DP)
  - ✓ more strait-forward
  - ✓ *no division into smaller instances*
  - ✓ get a solution from a sequence of choice

# Coin Change Problem

✓give the change with the minimum # of coins

✓an algorithm for the problem (Fig. 4.1 for 36cents)

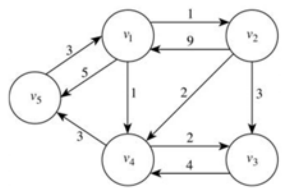
- ✓Coin set: Quarter (25cent), Dime(10cent), Nickel(5cent), Penny

```
while (more coins and not solved) {
  grab the largest remaining coin;          // selection
  if (exceed the amount)              // feasibility check
    reject the coin;
  else
    add the coin to the change;
  if (successful)                     // solution check
    the instance is solved;
}
```

✓Counter example: Fig. 4.2 for 16cents (with 12-cent coin)



# Graph Terminology

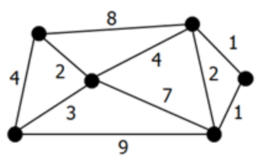


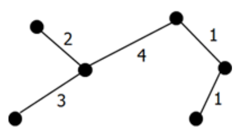
✓Graph G = (V, E)

- ✓V: set of vertexes E: set of edges

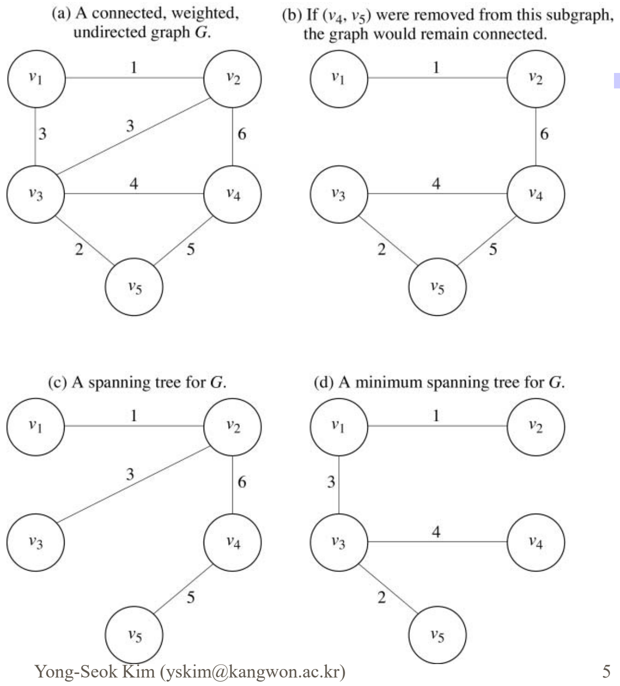
✓용어

- ✓directed / undirected graph
- ✓weighted / unweighted graph
- ✓connected / unconnected graph
- ✓path, length
- ✓cycle, cyclic / acyclic graph
- ✓tree, root, leaf, depth, height
- ✓tree / rooted tree
- ✓spanning tree
- ✓minimum spanning tree





# Minimum Spanning Tree






# Minimum Spanning Tree

- **The Problem** : find the minimum spanning tree for a given undirected graph
- **Applications**
  - Road Planning
  - Cabling (Telecommunication, Electricity)
  - Plumbing (Oil, Gas)
- Outline of a greedy approach algorithm

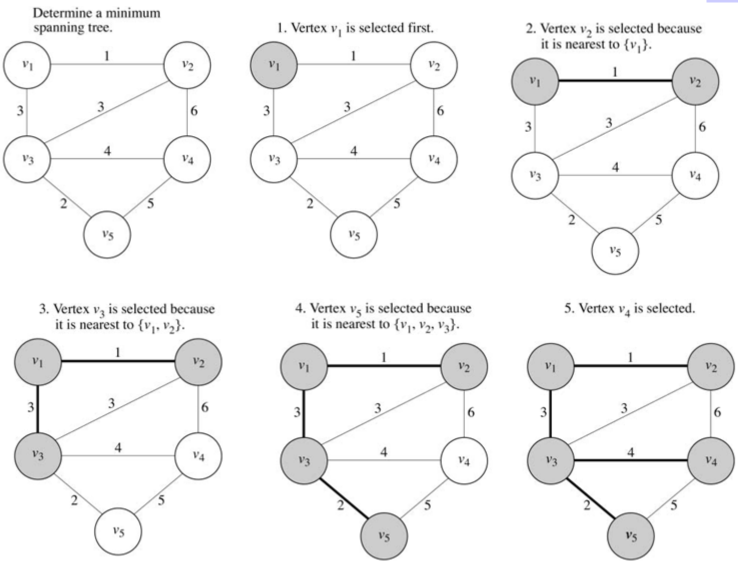
```
F = ∅;
while (the instance is not solved) {
    // selection procedure
    select an edge according to some locally optimal consideration
    // feasibility check
    if (the edge does not result a cycle) add the edge to F
    // solution check
    if (graph <V,F> is a spanning tree)  the instance is solved;
}
```

## Outline of Prim's Algorithm

```
Y = {v1}; F = ∅;
while (the instance is not solved) {
    // selection procedure (and feasibility check)
    select the vertex v in V-Y that is nearest to Y;
    e = the edge connecting v;
    Y = Y ∪ {v};
    F = F ∪ {e};
    if (Y == V)                // solution check
        the instance is solved;
}
```



## Example of Prim's Algorithm (Fig. 4.4)



## Prim's Algorithm

- ✓Data Structure
  - ✓W[1..n][1..n] : the adjacency matrix (input)
  - ✓nearest[i]: index of the vertex in Y nearest to $v_i$
  - ✓distance[i] : weight of the edge <$v_i$, nearest[i]>
- ✓The algorithm
  - ✓Algorithm 4.1 *prim*
- ✓Time Complexity

  $T(n) = (n-1) + (n-1) \cdot 2 \cdot (n-1) \in \Theta(n^2)$



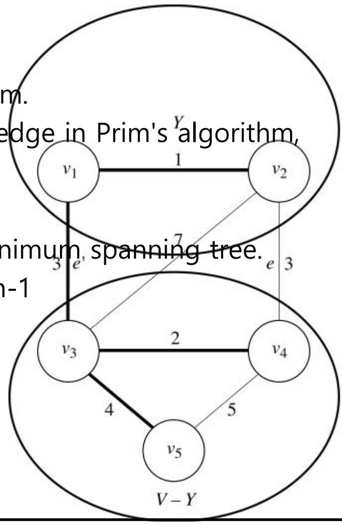
## Algorithm 4.1 Prim

```
void prim (int n,
           const number W[][],
           set_of_edges& F)
{
  index i, vnear;
  number min;
  edge e;
  index nearest[2..n];
  number distance[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){
      nearest[i] = 1;                // For all vertices, initialize v1
      distance[i] = W[1][i];         // to be the nearest vertex in
  }                                  // Y and initialize the distance
                                     // from Y to be the weight
                                     // on the edge to v1.
  repeat (n - 1 times){
                                     // Add all n - 1 vertices to Y.
      min = ∞;
      for (i = 2; i <= n; i++)       // Check each vertex for
         if (0 ≤ distance[i] < min){ // being nearest to Y.
            min = distance[i];
            vnear = i;
         }
      e = edge connecting vertices indexed
         by vnear and nearest[vnear];
      add e to F;
      distance[vnear] = -1;          // Add vertex indexed by
      for (i = 2; i <= n; i++)       // vnear to Y.
         if (W[i][vnear] < distance[i]){ // For each vertex not in
            distance[i] = W[i][vnear];   // Y, update its distance
            nearest[i] = vnear;          // from Y.
         }
  }
}
```

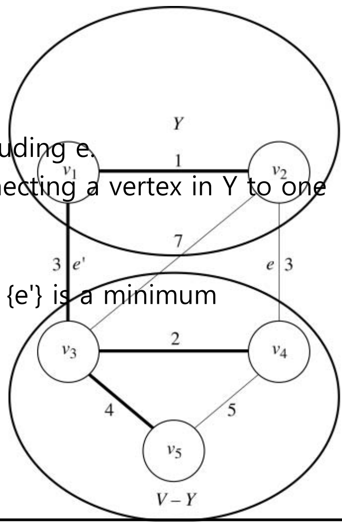
## Proof of Correctness

- Definition
  - A subset F ⊆ E is called ***promising*** if F is a subset of F' where <V,F'> is a minimum spanning tree.
- Lemma 1
  - Let F and Y be those of Prim's algorithm.
  - If F is promising and e is the selected edge in Prim's algorithm, then F ∪ {e} is promising.
- Theorem 1
  - Prim's algorithm always produces a minimum spanning tree.
  - *proof)* by induction from F=∅ to |F| = n-1




## Proof of Lemma 1

- Since F is promising, there exists F' s.t. F⊆F' and <V,F'> is a MST.
- **Case 1)** If e ∈ F',
  - F ∪ {e} ⊆ F' and F ∪ {e} is promising.
- **Case 2)** Otherwise,
  - F' ∪ {e} contains exactly one cycle including e.
  - Then, there is e' ∈ F' in the cycle connecting a vertex in Y to one in V-Y.
  - Then, F' ∪ {e} - {e'} is a spanning tree.
  - Since weight(e) ≤ weight(e'), F' ∪ {e} - {e'} is a minimum spanning tree.
  - F ∪ {e} ⊆ F' ∪ {e} - {e'}
  - Therefore, F ∪ {e} is promising.



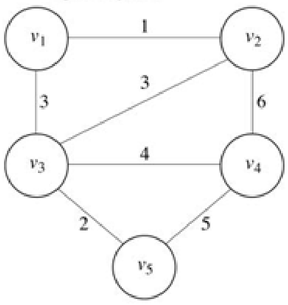
# Outline of Kruskal's Algorithm

```
F = ∅;
create disjoint subsets of V containing only one vertex;
sort the edges in E in non-decreasing order;
while (not solved) {
    select the next edge e;                  // selection
    if (e merges two disjoint subsets) {     // feasibility
        merge the subsets;
        F = F ∪ {e};
    }
    if (all subsets are merged)              // solution
        the instance is solved;
}
```



# Example (Fig. 4.7)

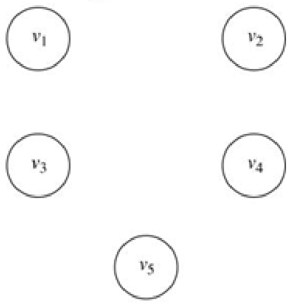
Determine a minimum spanning tree.

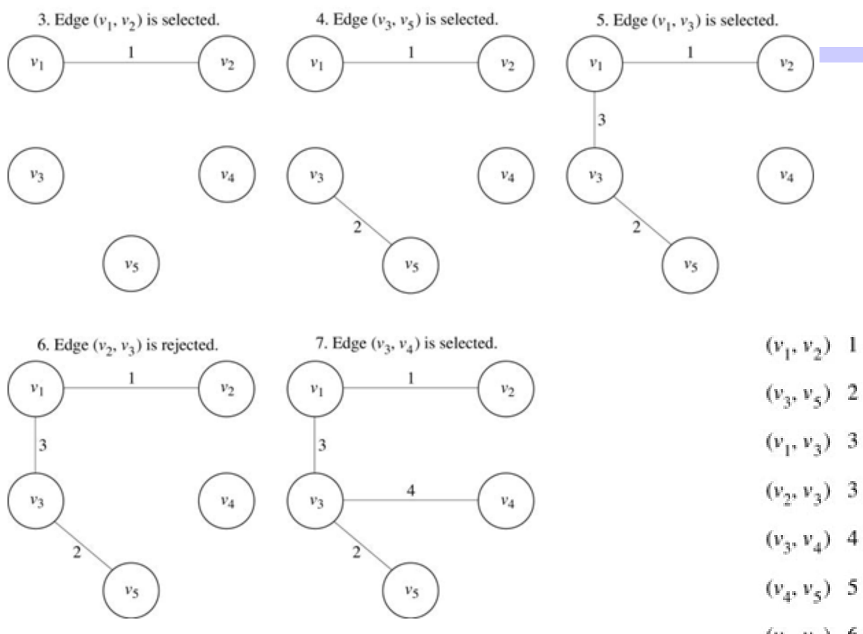



1. Edges are sorted by weight.

$(v_1, v_2)$ 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

2. Disjoint sets are created.








## Kruskal's Algorithm (Alg. 4.2)

- ✓Algorithms for Disjoint Sets (Appendix C)
  - ✓initial(n) : initialize n disjoint subsets
  - ✓find(i) : find the subset containing index i
  - ✓merge(p,q) : merge the two subsets
  - ✓equal(p,q) : check if the two subsets are equal
- ✓Time complexity of the algorithm
  - ✓Time = sort-time + initial-time + while-loop-time
  - ✓T(m,n) = $\Theta(m \log m) + n + \Theta(m \log m)$
    $= \Theta(m \log m)$
- ✓Proof of Correctness
  - ✓Similar to Prim's algorithm

# Algorithm 4.2 Kruskal

```
void kruskal (int n, int m,
              set_of_edges E,
              set_of_edges& F)
{
  index i, j;
  set_pointer p, q;
  edge e;
  Sort the m edges in E by weight in nondecreasing order;
  F = ∅;
  initial(n);                    // Initialize n disjoint subsets.
  while (number of edges in F is less than n − 1){
    e = edge with 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;
    }
  }
}
```



# Comparison of the Two Algorithm

- ✓Time complexity
  - ✓Prim: Θ($n^2$)
  - ✓Kruskal: Θ(m log m)
- ✓Comparison
  - ✓dense graph:
  - ✓Sparse graph:
- ✓More Improvements
  - ✓Prim's Algorithm : Prim(1957) ← originated from Jarnik (1930)
  - ✓Kruskal's Algorithm : Kruskal (1956)
  - ✓Johnson (1977): Θ(m log n) ← use heap to improve Prim
  - ✓Fredman and Tarjan: Θ(m + n log n) ← use Fibonacci heap

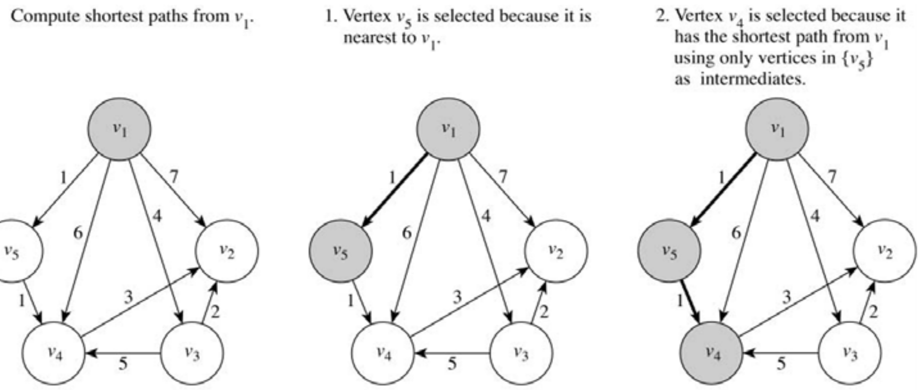
## Single-Source Shortest Paths

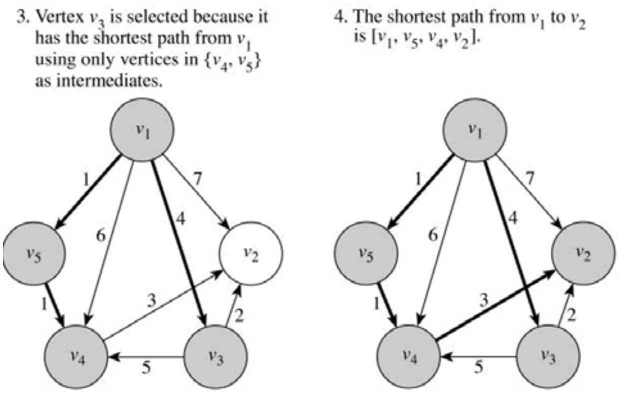
- **Shortest Path Problems**
  - All Pairs : $\Theta(n^3)$ (Floyd by Dynamic programming)
  - **Single Source :** $\Theta(n^2)$ **(Dijkstra by Greedy)**
  - Single Pair : $\Theta(n^2)$
- **Outline of the Algorithm**

Y = {$v_1$}; F = ∅;
while (not solved) {
    select v from V-Y with the shortest path from $v_1$ using only vertices in Y;    ← Prim: nearest to Y
    e = the edge touching v;
    Y = Y ∪ {v};    ← Prim: connecting v
    F = F ∪ {e};
    if (Y == V)
        the instance is solved;
}



## Example (Fig. 4.8)








# Dijkstra's Algorithm

- ✓Almost the same as Prim's algorithm
- ✓Data structure
  - ✓W[1..n][1..n] : the adjacency matrix
  - ✓touch[i]: index of the vertex in Y touching $v_i$
  - ✓length[i] : length of the path from $v_1$ to $v_i$
- ✓Time Complexity
  - ✓$T(n) = (n-1) + (n-1)((n-1)+(n-1)) \in \Theta(n^2)$

```
void dijkstra (int n, const number W[][], set_of_edges& F)
{
  index i, vnear;
  edge e;
  index touch[2..n];
  number length[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){          // For all vertices, initialize v1
     touch[i] = 1;                   // to be the last vertex on the
     length[i] = W[1][i];            // current shortest path from
  }                                  // v1, and initialize length of
                                     // that path to be the weight
                                     // on the edge from v1.
   repeat (n − 1 times){                // Add all n − 1 vertices to Y.
      min = ∞;
      for (i = 2; i <= n; i++)          // Check each vertex for
         if (0 ≤ length[i] < min){      // having shortest path.
            min = length[i];
            vnear = i;
         }
      e = edge from vertex indexed by touch[vnear]
         to vertex indexed by vnear;
      add e to F;
      for (i = 2; i <=n; i++)
         if (length[vnear] + W[vnear][i] < length[i]){
            length[i] = length[vnear] + W[vnear][i];
            touch[i] = vnear;           // For each vertex not in Y,
         }                              // update its shortest path.
      length[vnear] = −1;               // Add vertex indexed by vnear
   }                                    // to Y.
}
```



# Scheduling

- ✓Scheduling criteria
  - ✓1. Minimize time in the system (response time) (*Note: include waiting time*
  - ✓2. Maximize the total profit (real-time scheduling)
- ✓Minimize average response time
  - ✓Example 4.2 (3 jobs): t1 = 5, t2 = 10, t3 = 4
  - ✓[1,2,3]: 5 + (5+10) + (5+10+4) = 39
  - ✓[1,3,2]: 33
  - ✓...
  - ✓[3,1,2]: 4 + (4+5) + (4+5+10) = 32
- ✓**Greedy approach algorithm?**

## Shortest Job First Scheduling

- Outline of the algorithm

```
sort jobs by service time in non-decreasing order;
while (not solved) {
    schedule the next job;    // selection (and feasibility)
    if (no more jobs)         // solution check
        the instance is solved;
}
```

- Theorem 4.3 (optimality)
  - SJF results minimum response time schedule
  - *proof)* If there is at least one i such that $t_i > t_{i+1}$ in an optimal solution, we can **exchange the order of job i and job i+1.**
    Then, the total response time is $T' = T + t_{i+1} - t_i < T$.
    This is a contradiction



## Variants of Scheduling Problem

- Job characteristics
  - Execution time
  - Arrival time
  - Deadline
  - Preemption
  - Multiple servers
  - …
- Objectives
  - Minimize response time
  - Maximize profit
  - …

## Scheduling with Deadlines

- Real-time scheduling
- The problem
  - the objective is maximizing the total profit
  - the profit of a job is 0 if missed its deadline
  - not all jobs have to be scheduled
  - the execution time is 1 for all jobs
  - all jobs are arrived at time 0
  - see Example 4.3
- Terminology
  - feasible job sequence (feasible schedule)
  - feasible job set
  - optimal job sequence (optimal schedule)
  - optimal set of jobs



## Example 4.3

- Execution time is 1
- (Job, Deadline, Profit)
  - (1, 2, 30), (2, 1, 35), (3, 2, 25), (4, 1, 40)
- Profits of feasible schedules
  - [1,3]: 30 + 25 = 55
  - [2,1]: 35 + 25 = 65
  - ...
  - [4,1]: 40 + 30 = 70
- **Greedy approach algorithm?**

## Earliest-Deadline-First (EDF) scheduling

✓ Outline of the algorithm

```
sort jobs by profit in non-increasing order;
S = empty;
while (not solved) {
    select next job j;          // selection procedure
    if (S U {j} is feasible)    // feasibility check
        S = S U {j};
    if (no more jobs)           // solution check
        the instance is solved;
}
```

✓ Time complexity (S is maintained in deadline order)

- ✓ T(n) <= n log n + n x (feasibility check and addition) = $O(n^2)$



## Correctness of EDF Scheduling

✓ **Lemma 4.3**

- ✓ A set S of jobs is feasible iff EDF sequence of S is feasible.
- ✓ *proof) (basic idea)*

If there is at least one $i$ such that $d_i > d_{i+1}$ in a feasible sequence, we can exchange the order of job $i$ and job $i+1$.

Then, the new sequence is feasible.

✓ **Theorem 4.4**

- ✓ The algorithm always produces an optimal set of jobs.
- ✓ *proof) skip*. (use mathematical induction)

## Huffman Code

- Data compression (Variable length binary code)
  - 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당
  - 압축파일을 복원할 때 명확하게 처리할 수 있어야 함 → Prefix property
- Prefix property
  - 각 문자에 할당된 이진 코드는 어떤 다른 문자에 할당된 이진 코드의 prefix가 되지 않음
  - 장점: 코드와 코드 사이를 구분할 특별한 코드가 필요 없음
  - (예) 문자 ‘a’에 할당된 코드가 ‘101’이라면, 모든 다른 문자의 코드는 ‘101’로 시작되지 않으며, 또한 ‘1’이나 ‘10’도 아님
- Prefix property 가 없으면
  - 코드와 코드 사이를 구분할 특별한 코드가 필요함
  - 예를 들어, 101#10#1#111#0#…에서 ‘#’ 부분에 특수 코드 필요
- Huffman code:
  - an optimal variable length binary code with prefix perperty



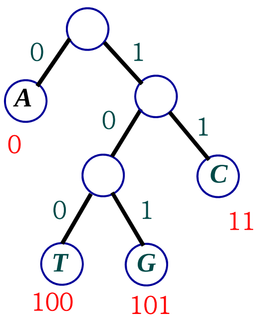
## Huffman Code와 Binary Tree

- Huffman code 와 대응되는 이진 트리를 구성
- 빈도가 높은 것이 루트에 가깝도록

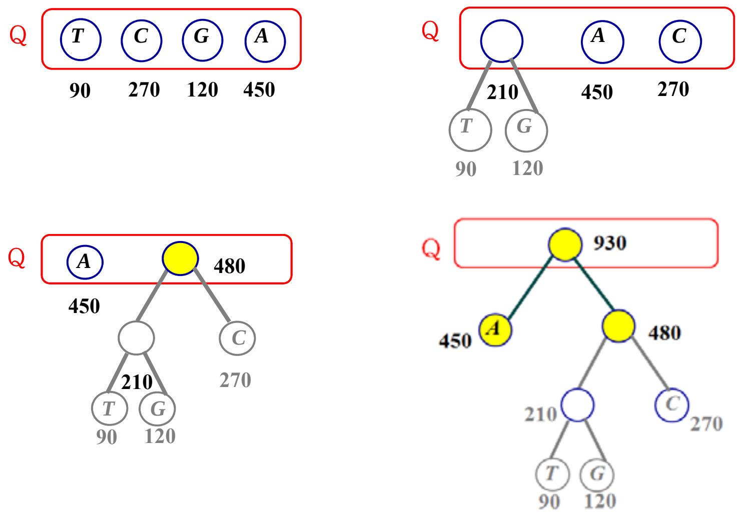
← 빈도가 낮은 것들끼리 묶으면서 이진 트리 구성

- 문자별 빈도 예
  - A: 450 T: 90 G: 120 C: 270

| 문자 | 코드 |
|---|---|
| A | 0 |
| C | 11 |
| G | 101 |
| T | 100 |



# Build the Binary Tree






# Huffman Code

- ✓ Data structure for Huffman code

```
struct nodetype {
    char symbol;
    int freq;
    struct nodetype *left, *right;
}
```

- ✓ Priority Queue
  - ✓ remove(PQ, p)
  - ✓ insert(PQ, p)

    (lower frequency means higher priority)

## Huffman Code Algorithm

```
nodeType *huffman(int n, queue PQ)
{       int i;   nodeType *p, *q, *r;
        for (i=1; i <= n-1; i++) {
                remove(PQ, p); remove(PQ, q);
                r = new nodeType;
                r->left = p; r->right = q;
                r->freq = p->freq + q->freq;
                insert(PQ, r);
        }
        remove(PQ, r);
        return r;
}
```



- ✓Time complexity
  - ✓Init. priority queue(heap): $T_{init}(n) \le n \log n$
  - ✓$T(n) \le T_{init}(n) + (n-1) \times 3 \log n$
  - ✓$T(n) \in O(n \log n)$ and actually $\Theta(n \log n)$
- ✓Proof of correctness
  - ✓Lemma 4.4 and Theorem 4.5
  - ✓(skip)

## The Knapsack Problem

- ✓The 0-1 knapsack problem
  - ✓select items for a given knapsack
  - ✓the capacity of knapsack is restricted
  - ✓maximize the total profit
  - ✓items can't be divided
- ✓Definition of the problem

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

  $w_i$ = weight of $item_i$

  $p_i$ = profit of $item_i$

  $W$ = maximum capacity of the knapsack

  The solution: maximize the sum of profits

  subjects to the sum of weights ≤ W



## The Knapsack Problem

- ✓Example : Fig. 4.13
  - ✓<profit, weight> = <50,5>, <60,10>, <140,20>
  - ✓capacity = 30
  - ✓Greedy1 : most profitable item first
  - ✓Greedy2 : lightest item first
  - ✓Greedy3 : most profitable/weight item first
- ✓Fractional knapsack problem
  - ✓use Greedy3 and a fraction of the last item
- ✓0-1 Knapsack problem
  - ✓Dynamic programming algorithm of 0-1 Knapsack
  - ✓(skip)
  - ✓NP-Hard problem