

---

“본 강의 동영상 및 자료는 대한민국 저작권법을 준수합니다. 본 강의 동영상 및 자료는 상명대학교 재학생들의 수업목적으로 제작·배포되는 것이므로, 수업목적으로 내려받은 강의 동영상 및 자료는 수업목적 이외에 다른 용도로 사용할 수 없으며, 다른 장소 및 타인에게 복제, 전송하여 공유할 수 없습니다. 이를 위반해서 발생하는 모든 법적 책임은 행위 주체인 본인에게 있습니다.”

---

# 5. Greedy algorithm

---

## 5.0 Basics

## 5.1 Minimum spanning trees

## 5.2 Knapsack problem

## 5.3 Job sequencing with deadline

## 5.4 Optimal merge patterns

## 5.5 Huffman encoding

## 5.4 Optimal merge pattern

---

- Problem:
  - Merge  $k$  files  $\rightarrow$  Various combinations of merging patterns
  - c.f. Merging two sorted files containing  $n$  and  $m$  records into one file takes  $O(n+m)$ .
  - Determine an optimal way to pairwise merge  $k$  sorted files together.

## 5.4 Optimal merge pattern

---

- Example:
  - X1, X2 and X3 are three sorted files of length 30, 20, and 10 records each.
  - Merge pattern 1:
    - Merge X1 & X2  $\rightarrow$  Y1 (50 steps)
    - Merge Y1 & X3  $\rightarrow$  Y2 (60 steps)
  - Merge pattern 2:
    - Merge X2 & X3  $\rightarrow$  Y1 (30 steps)
    - Merge Y1 & X1  $\rightarrow$  Y2 (60 steps)
  - Compare the time required

## 5.4 Optimal merge pattern

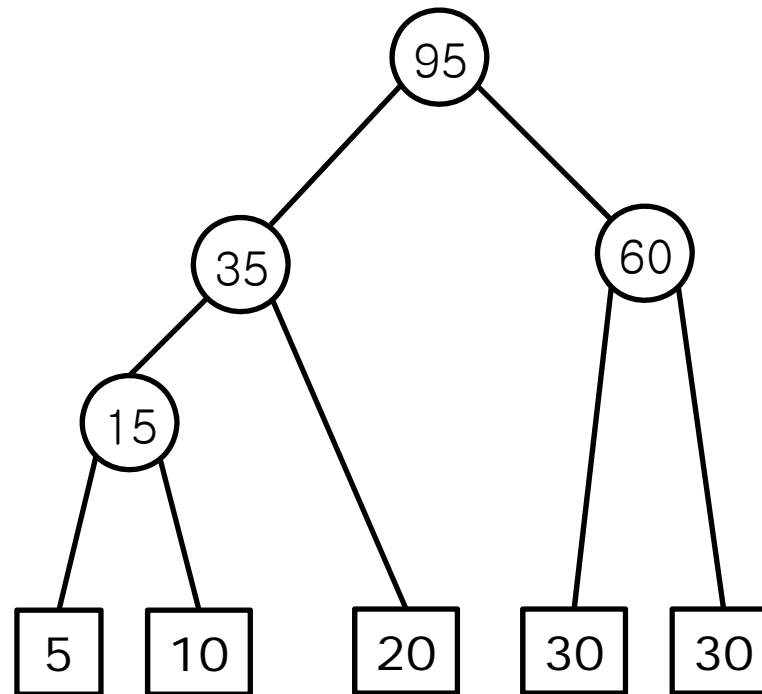
---

- Solution strategy:
  - 2-way merge pattern can be represented as a binary merge tree with minimum weighted external path length
  - Example:
    - Five patterns with (20, 30, 10, 5, 30)

## 5.4 Optimal merge pattern

---

- Solution strategy:
  - Corresponding binary tree



## 5.4 Optimal merge pattern

---

- Solution strategy:
  - If a pattern of length  $q_i$  is stored at a node whose depth is  $d_i$ , then the number of moves of the pattern is  $q_i d_i$ .
  - The total moves of the patterns is:

$$\sum_{1 \leq i \leq n} d_i q_i$$

- The weighted external path length of a tree

## 5.4 Optimal merge pattern

---

- Optimal merge pattern

```
int build_tree ( node *tree, int n, int L[] )
{
    int sum = 0;
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← L.pop ( ); // get the minimum of L
        tnode->right ← L.pop ( ); // get the minimum of L
        tnode->weight ← tnode->left->weight + tnode->right->weight;
        sum += tnode->weight;
        insert ( tnode, tree );
        L.insert (tnode->weight);
    }

    return sum;
}
```



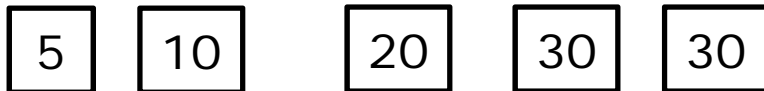
## 5.4 Optimal merge pattern

---

- Solution strategy:
  - Corresponding binary tree

```
int build_tree ( node *tree, int n, int L[] )
{
    int sum = 0;
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← L.pop ( ); // get the minimum of L
        tnode->right ← L.pop ( ); // get the minimum of L
        tnode->weight ← tnode->left->weight + tnode->right->weight;
        sum += tnode->weight;
        insert ( tnode, tree );
        L.insert (tnode->weight);
    }

    return sum;
}
```



## 5.4 Optimal merge pattern

---

- Time complexity
  - L: priority queue
    - L.pop ( )  $\rightarrow O(1)$
    - L.insert ( )  $\rightarrow O(\log n)$
  - $O(n) * O(\log n) \rightarrow O(n \log n)$

```
int build_tree ( node *tree, int n, int L[] )
{
    int sum = 0;
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← L.pop ( ); // get the minimum of L
        tnode->right ← L.pop ( ); // get the minimum of L
        tnode->weight ← tnode->left->weight + tnode->right->weight;
        sum += tnode->weight;
        insert ( tnode, tree );
        L.insert (tnode->weight);
    }

    return sum;
}
```