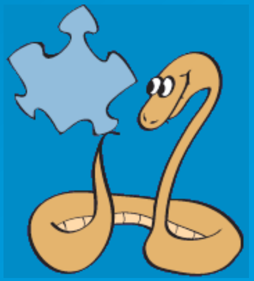
파이썬 알고리즘

ALGORITHMS IN PYTHON

최영규 지음



CHAPTER 05

분할 정복 기법

학습 내용

5.1 순환 관계식과 마스터 정리

5.2 병합 정렬

5.3 퀵 정렬

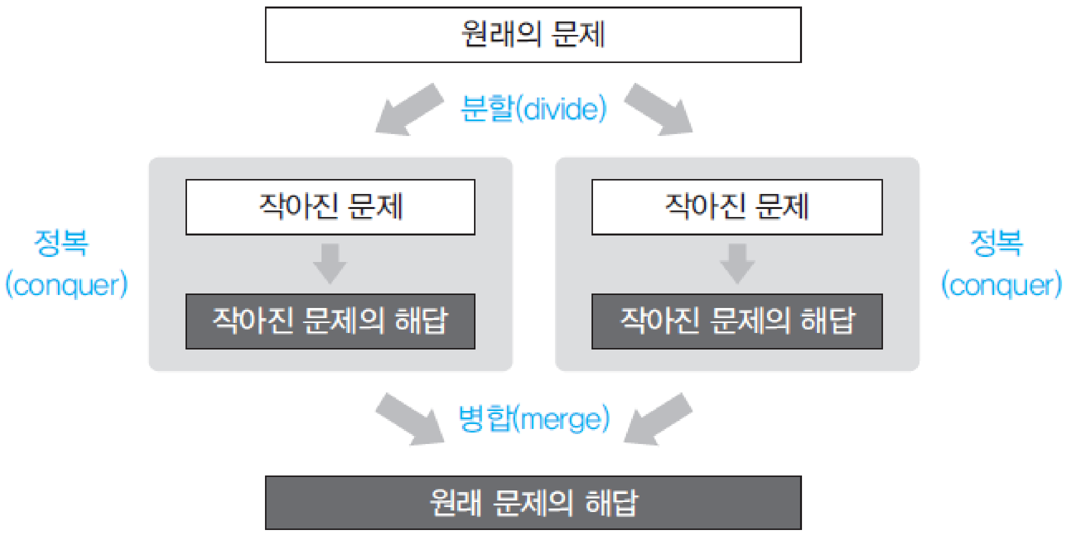
5.4 이진트리 관련 문제

5.5 최근접 쌍의 거리 문제(심화)

5.6 행렬 곱셈(심화)

5.7 피보나치수열과 분할 정복의 주의점

분할 정복 기법(divide-and-conquer)

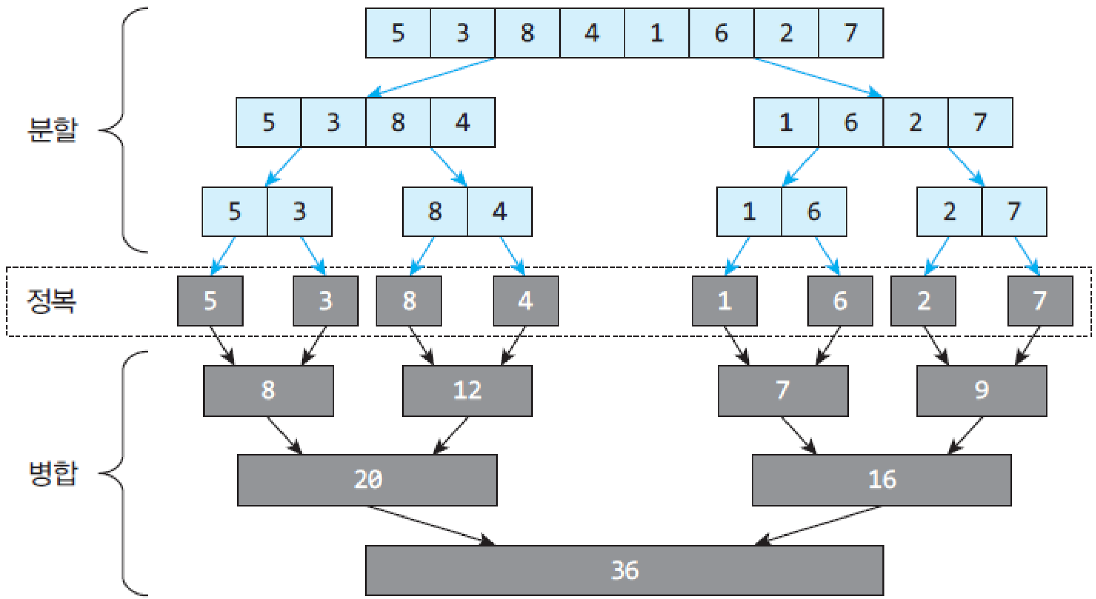


- 분할 정복 단계

- **분할**(divide): 주어진 문제를 같은 유형의 부분문제들로 분할한다.
- **정복**(conquer): 부분문제들을 해결하여 부분 해를 만든다.
- **병합**(combine): 부분적인 결과들을 묶어 최종 해를 만든다.

리스트의 합 구하기

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots a_{n-1})$$



– 정렬이 나 탐색과 같은 많은 중요한 문제에서 상당한 효과
– 항상 억지 기법보다 모든 문제에서 더 효율적인 것은 아님.

5.1 순환 관계식과 마스터 정리

- 순환 관계식
  - 크기가 *n*인 문제 → *b*개의 같은 크기의 부분 문제
  - 이들 중에서 *a*개만 다시 해결해야 하는 경우
    - 단순화를 위해 $n = b^k$이라 가정
  - 순환 관계식

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

    - $f(n)$: 문제를 부분문제들로 나누는 과정과 부분문제들의 해를 병합하여 최종해를 만드는데 사용되는 연산들의 합

  - 예: 리스트의 합 구하기
    - 숫자들은 두 그룹: b=2
    - 모든 그룹이 해결되어야 함: a=2
    - 병합을 위해 한번 더함: f(n) = 1

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

순환 관계식의 계산: 연속 대치법

$$
\begin{aligned}
T(n) &= 2T(n/2)+1 \\
&= 2[2T(n/2^2)+1]+1 \\
&= 2^2 T(n/2^2)+2+1 \\
&= 2[2T(n/2^3)+1]+2+1 \\
&= 2^3 T(n/2^3)+2^2+2+1 \\
&\cdots \\
&= 2^i T(n/2^i)+2^{i-1}+2^{i-2}+\cdots+2+1 \\
&= 2^k T(n/2^k)+2^{k-1}+2^{k-2}+\cdots+2+1 \\
&= 2^k T(1)+\sum_{i=0}^{k-1} 2^i \\
&= 2^k \cdot 0+\frac{1-2^k}{1-2} = 0+\frac{1-n}{-1} = n-1 \quad \in O(n)
\end{aligned}
$$

순환 관계식의 계산: 마스터 정리

다음과 같은 순환식이 주어졌다고 하자.

$$T(n) = \begin{cases} c & n = 1 \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

이때, $a \geq 1, b > 1$ 이고, $f(n) \in O(n^d)$ 이라면, 다음과 같이 점근적 수행 시간을 계산할 수 있다.

$$T(n) = \begin{cases} O(n^d) & \text{if} \quad a < b^d \\ O(n^d \log n) & \text{if} \quad a = b^d \\ O(n^{\log_b a}) & \text{if} \quad a > b^d \end{cases}$$

이것은 Ω −표기와 Θ −표기에서도 동일하게 적용된다.

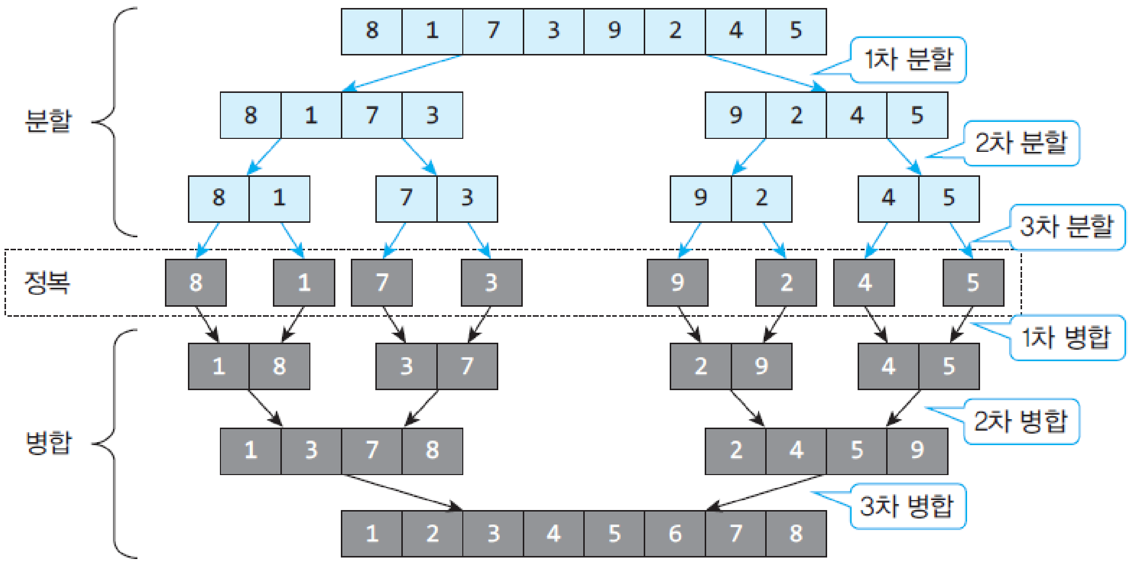
- 리스트의 합 문제의 $T(n) = 2T(n/2) + 1$ 에 대해 적용해 보자. $a=b=2$이고, $f(n) = 1 \in O(1) = O(n^0)$ 이므로 $d=0$이다. 따라서 세 번째 경우 $a > b^d (2 > 2^0 = 1)$ 에 해당하고, 점근적 복잡도는 $\boxed{O(\log^{\log_2 2}) = O(n)}$에 속한다.

5.2 병합 정렬

① **분할(Divide)**: 입력 리스트를 같은 크기의 2개의 부분 리스트로 분할한다.

② **정복(Conquer)**: 부분 리스트를 정렬한다. 이때, 부분 리스트의 크기가 충분히 작지 않으면 순환 호출을 이용하여 다시 분할 정복 기법을 적용한다. 리스트의 크기가 1이면 이미 정복(정렬)된 것이다.

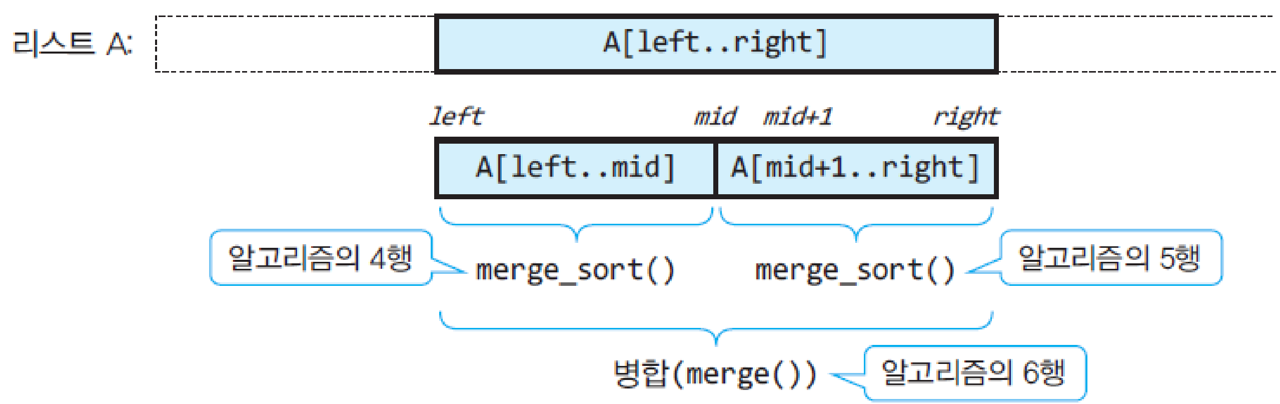
③ **병합(Combine, merge)**: 정렬된 부분 리스트들을 하나의 배열에 통합한다.



알고리즘: merge_sort()

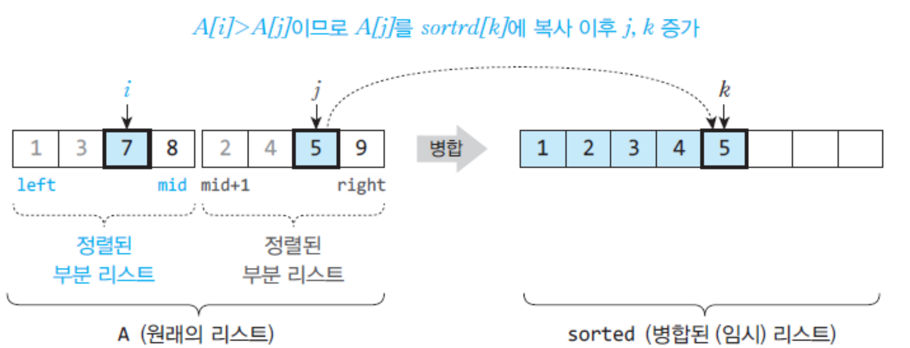
알고리즘 5.1 병합 정렬

```
01  def merge_sort(A, left, right) :
02      if left<right :
03          mid = (left + right) // 2
04          merge_sort(A, left, mid)
05          merge_sort(A, mid + 1, right)
06          merge(A, left, mid, right)
```



[그림 5.4] 알고리즘 5.1의 구조

정렬된 두 리스트의 병합



알고리즘 5.2 병합 알고리즘(정렬된 두 리스트의 병합)

```
def merge(A, left, mid, right) :
    k = left
    i = left
    j = mid + 1
    while i<=mid and j<=right :
        if A[i] <= A[j] :
            sorted[k] = A[i]
            i, k = i+1, k+1
        else:
            sorted[k] = A[j]
            j, k = j+1, k+1

    if i > mid :
        sorted[k:k+right-j+1] = A[j:right+1]
    else :
        sorted[k:k+mid-i+1] = A[i:mid+1]

    A[left:right+1] = sorted[left:right+1]
```

복잡도 분석

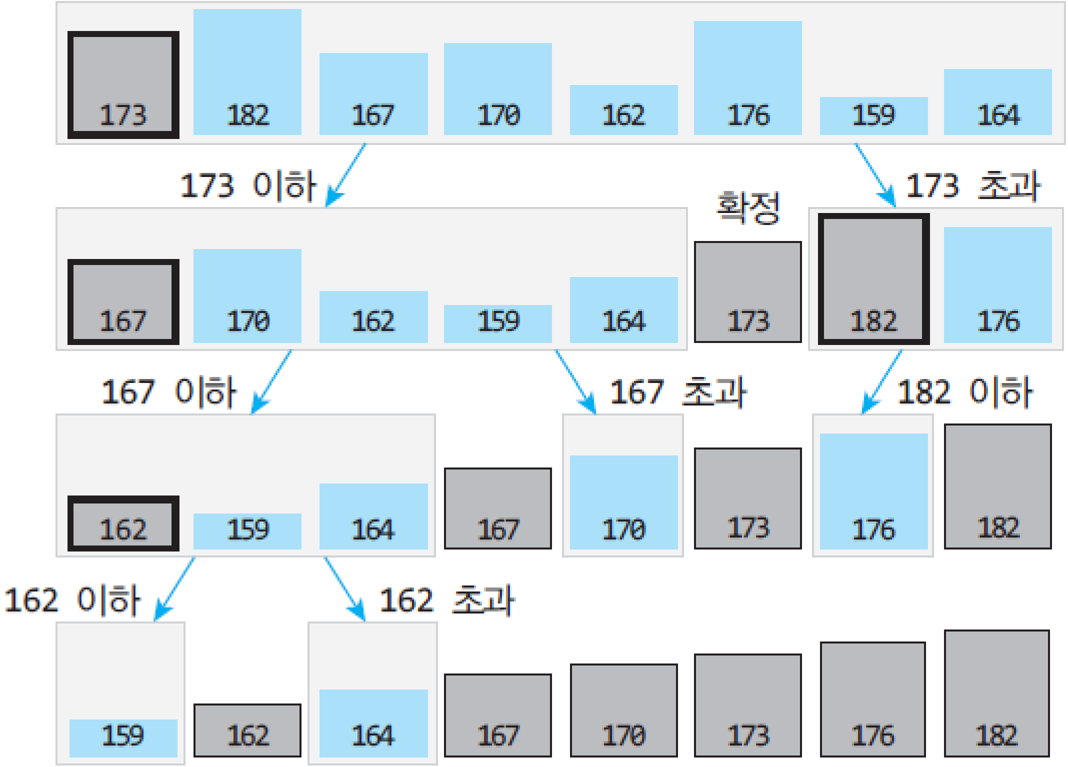
- 입력 구성에 따른 차이가 없음
  - 항상 동일한 시간에 정렬이 완료
- $n = 2^k$이라 가정
  - $k = log_2 n$
  - 부분 배열의 크기: 2^k → 2^{k-1}→… → 2^1 → 2^0
- 복잡도 함수

$$T(n) = 2T\left(\frac{n}{2}\right) + T_{merge}(n)$$
$$T(1) = 0$$

  - 마스터 정리 : $O(nlog_2 n)$
  - 연속 대치법 : $O(nlog_2 n)$
- 특징
  - 장점: 효율적. 안정성 만족. 동일한 시간.
  - 단점: 임시 리스트 사용

5.3 퀵 정렬

- 위치에 따른 분할(병합)이 아니라 값에 따른 분할



피벗 173을 기준으로 잡고 리스트를 173 이하인 그룹과 173 초과인 그룹으로 나눔

피벗 173은 이제 위치가 확정되었음.
왼쪽 그룹은 167을 기준으로 다시 나누고,
오른쪽 그룹은 182를 기준으로 다시 나눔

167, 182는 이제 위치 확정.
170, 176은 그룹에 하나뿐이므로
위치 확정. 162를 기준으로 다시 나눔

162는 위치 확정
159와 164도 그룹에 항목이 하나이므로
위치 확정

알고리즘: quick_sort()

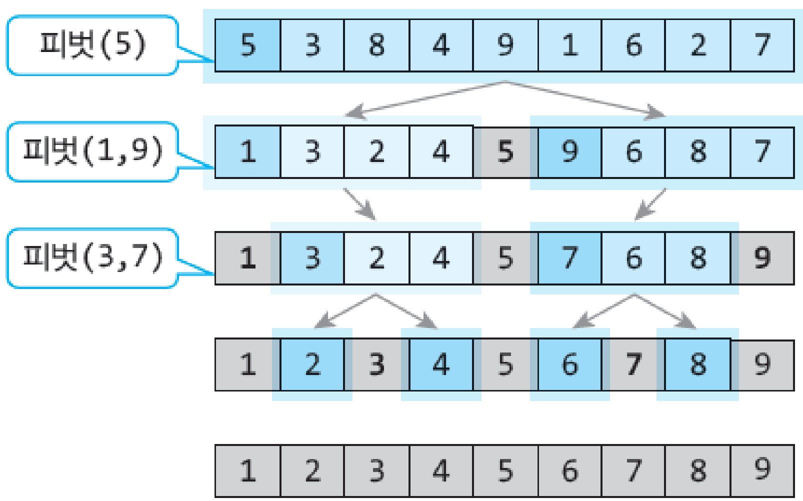
알고리즘 5.3 퀵 정렬

```
def quick_sort(A, left, right) :
    if left<right :
        mid = partition(A, left, right)
        quick_sort(A, left, mid-1)
        quick_sort(A, mid+1, right)
```

알고리즘 테스트 퀵 정렬 테스트

```
data = [ 5, 3, 8, 4, 9, 1, 6, 2, 7 ]          #
print("Original  : ", data)
quick_sort(data, 0, len(data)-1)              #
print("QuickSort : ", data)
```

```
C:\WINDOWS\system32\cmd.exe
Original  :  [5, 3, 8, 4, 9, 1, 6, 2, 7]
QuickSort :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```



복잡도 분석

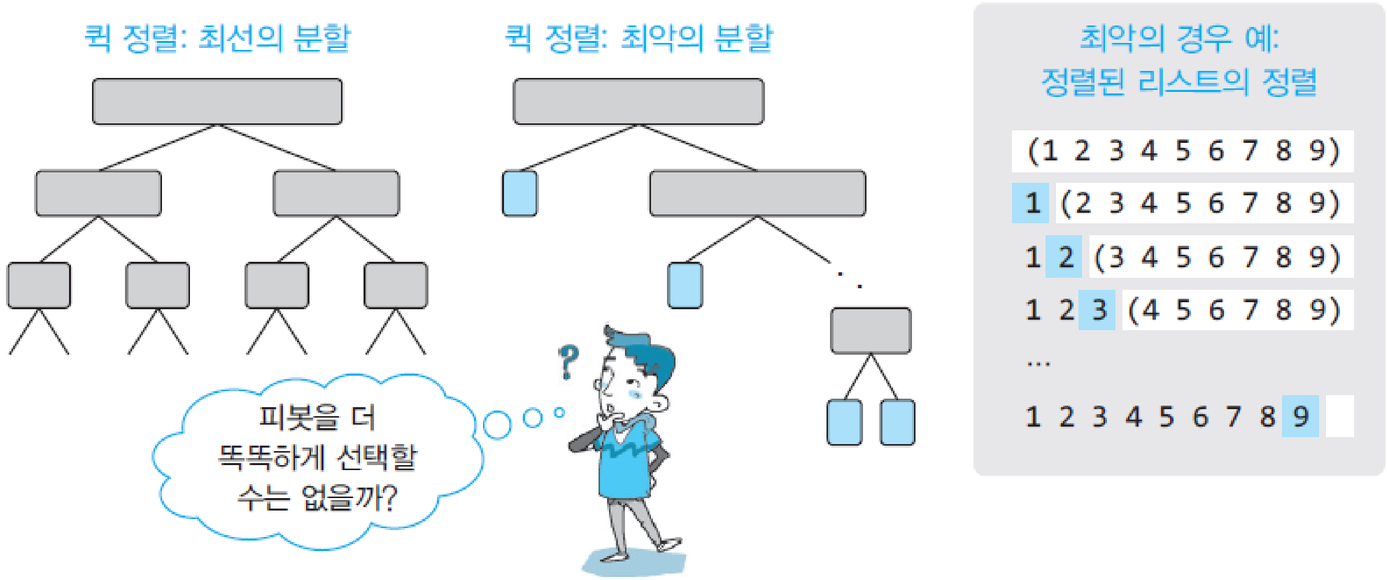
- 입력 구성에 따른 차이가 있음
- 최선의 경우 $\boxed{O(n\log_2 n)}$
  - 분할이 항상 균등하게 이루어지는 경우.
  - $n = 2^k$라 가정

$$
\begin{aligned}
T_{best}(n) &= 2T_{best}(n/2) + T_{merge}(n) \quad \text{for } n > 1 \\
&= 2T_{best}(n/2) + n - 1
\end{aligned}
$$

$$T_{best}(1) = 0$$

- 최악의 경우 $\boxed{O(n^2)}$
  - 불균등 분할: 이미 정렬된 리스트의 정렬

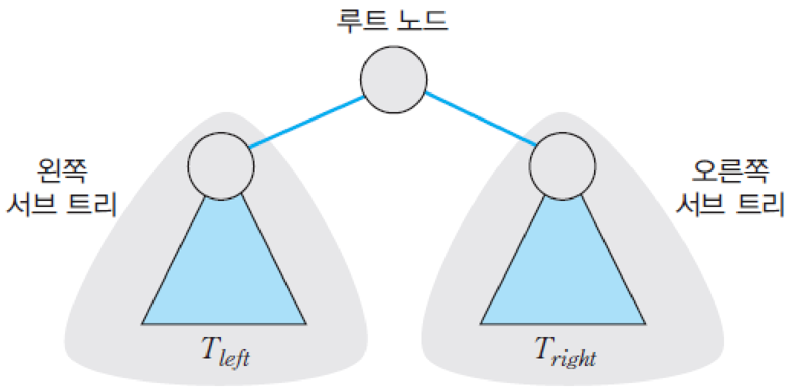
$$
\begin{aligned}
T_{worst}(n) &= T(0) + T_{worst}(n-1) + n - 1 \\
&= \frac{n(n-1)}{2} \in O(n^2)
\end{aligned}
$$



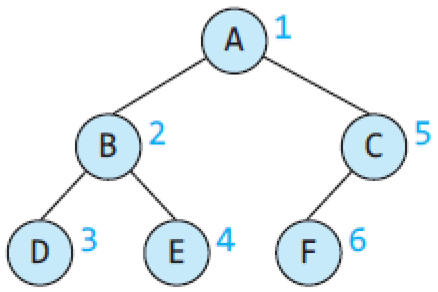
[그림 5.9] 퀵 정렬의 최선의 분할과 최악의 분할 예

- 개선을 위한 노력들
  - median of three
  - 이중 피벗 퀵 정렬(dual pivot quick sort)
  - 하이브리드(hybrid) 정렬

5.4 이진트리 관련 문제



- 이진트리 관련 문제들

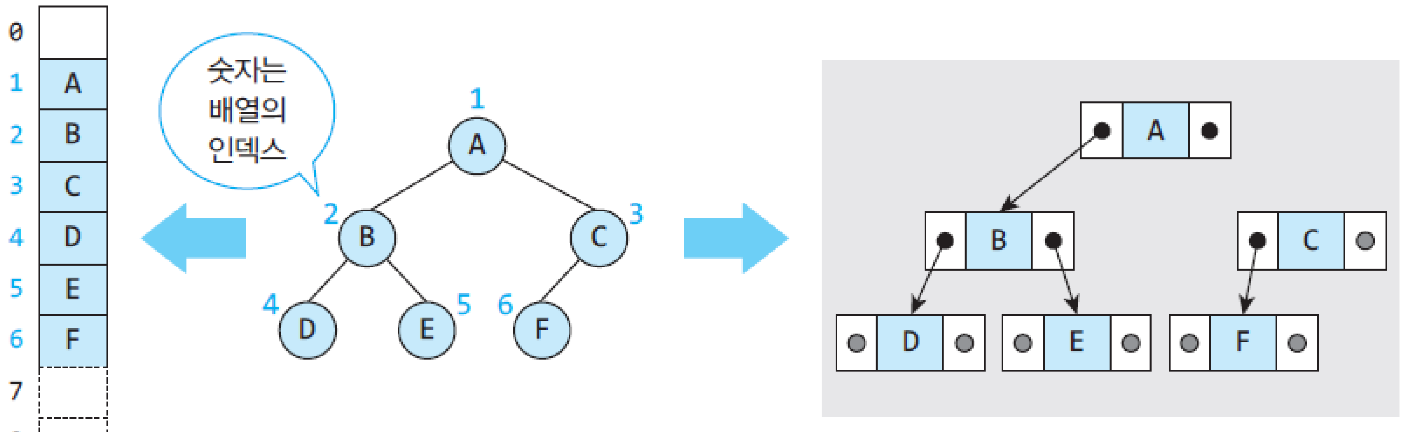


문제 1: 이진트리의 전체 노드의 수는? 답) 6
문제 2: 이진트리의 높이는? 답) 3
문제 3: 이진트리의 단말노드의 수는? 답) 3
문제 4: 모든 노드를 한번씩 방문하는 방법은?
…

  – 분할 정복 기법이 활용

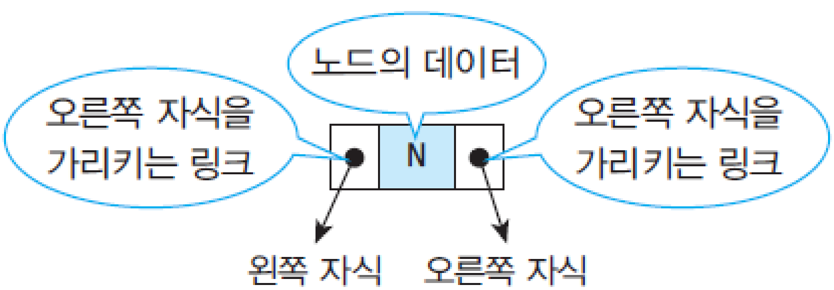
이진트리의 표현

- 배열구조 / 연결된 구조



연결된 구조 표현

배열 구조 표현

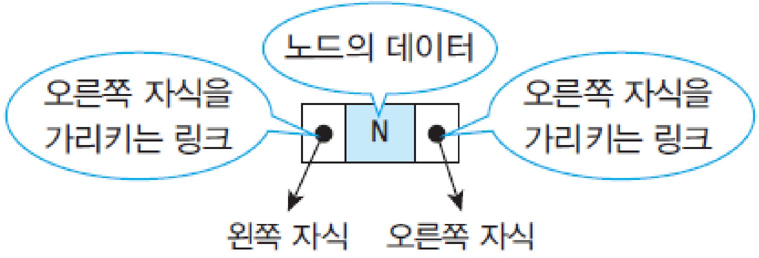


[그림 5.13] 이진트리의 노드 구조

연결된 구조

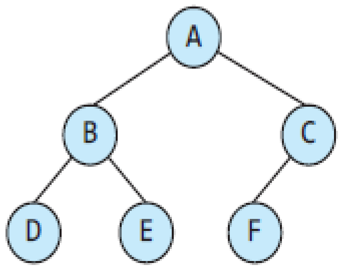
- 노드 클래스

```
class TNode:
    def __init__ (self, data, left, right):
        self.data = data
        self.left = left
        self.right = right
```



[그림 5.13] 이진트리의 노드 구조

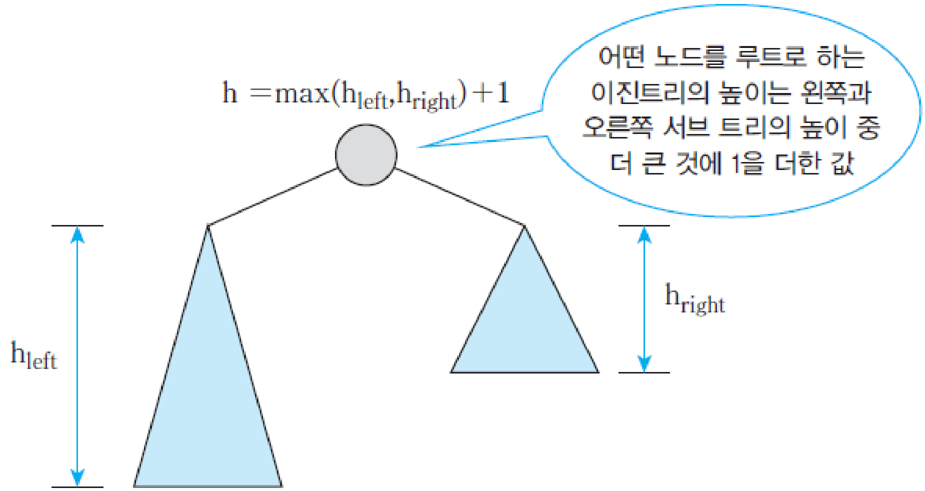
- 예)




```
d = TNode('D', None, None)
e = TNode('E', None, None)
b = TNode('B', d, e)
f = TNode('F', None, None)
c = TNode('C', f, None)
root = TNode('A', b, c)        # 루트 노드
```

이진트리의 높이 문제

- 분할정복 아이디어



알고리즘 5.4 이진트리의 높이

```
def calc_height(root) :
    if root is None :
        return 0
    hLeft = calc_height(root.left)
    hRight = calc_height(root.right)
    return max(hLeft,hRight) + 1
```

알고리즘 테스트 이진트리의 높이 테스트

```
...                                         # 그림 5.14의
print(" 트리의 높이 =", calc_height(root))
```

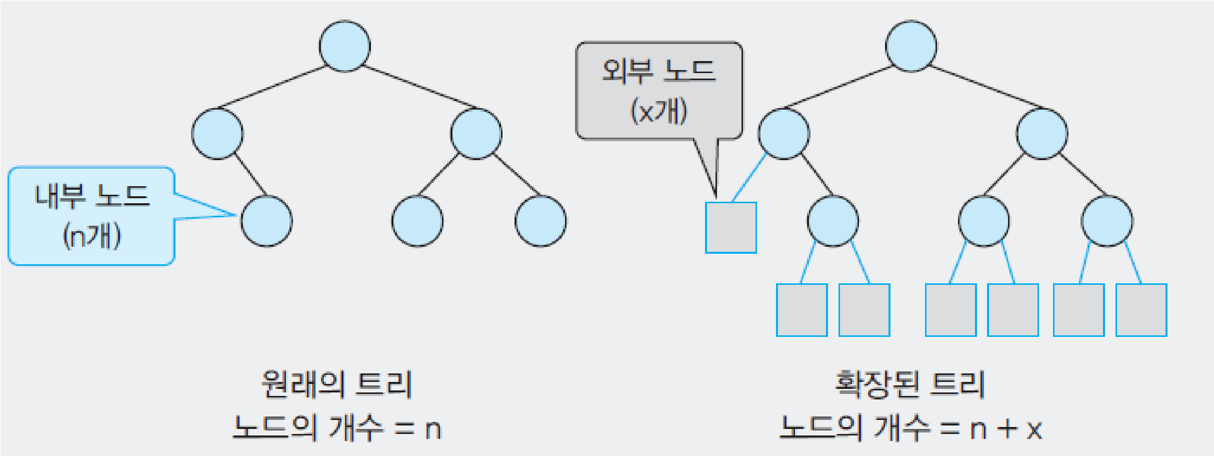
C:₩WINDOWS₩system32₩cmd.exe

```
트리의 높이 = 3
```

복잡도 분석

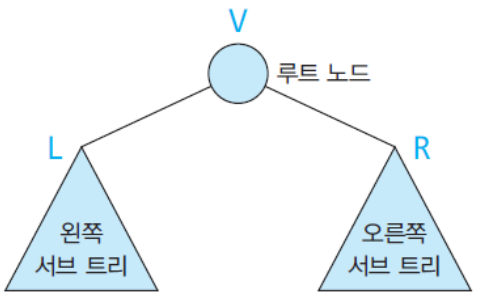
- 노드의 수 : n
- 복잡도 함수
  - $T(n) = T(n_{left}) + T(n_{right}) + 1 \quad for\ n > 0$
  - $T(0) = 0$

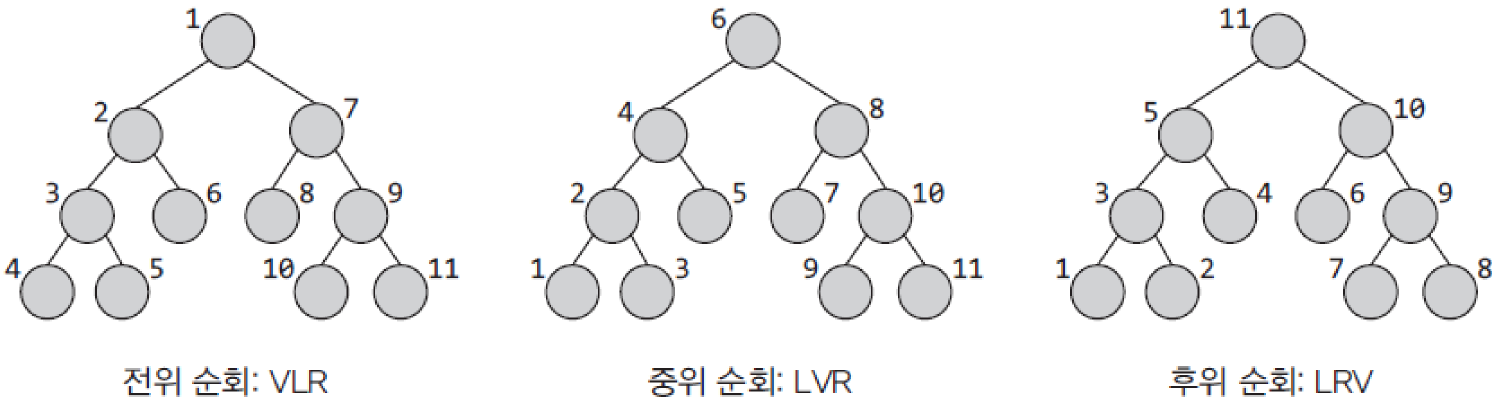
- 기본연산
  - 6행의 덧셈: 내부 노드만 검사 → $T(n) \in O(n)$
  - 2행의 공백검사: 외부 노드까지 검사 → $T(n) \in O(n)$ 동일



이진트리의 표준 순회 문제

- 분할정복 아이디어
  - 전위 순회(preorder traversal): VLR
  - 중위 순회(inorder traversal): LVR
  - 후위 순회(postorder traversal): LRV





알고리즘

- 표준 순회 알고리즘

알고리즘 5.5 이진트리의 전위 순회

```
def preorder(n) :
    if n is not None :
        print(n.data, end=' ')
        preorder(n.left)
        preorder(n.right)
```

알고리즘 5.6 이진트리의 중위 순회

```
def inorder(n) :
    if n is not None :
        inorder(n.left)
        print(n.data, end=' ')
        inorder(n.right)
```

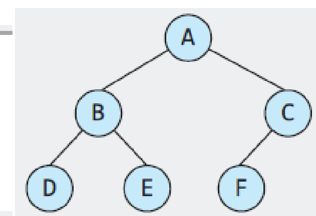
알고리즘 5.7 이진트리의 후위 순회

```
def postorder(n) :
    if n is not None :
        postorder(n.left)
        postorder(n.right)
        print(n.data, end=' ')
```

- 실행결과

C:₩WINDOWS₩system32₩cmd.exe

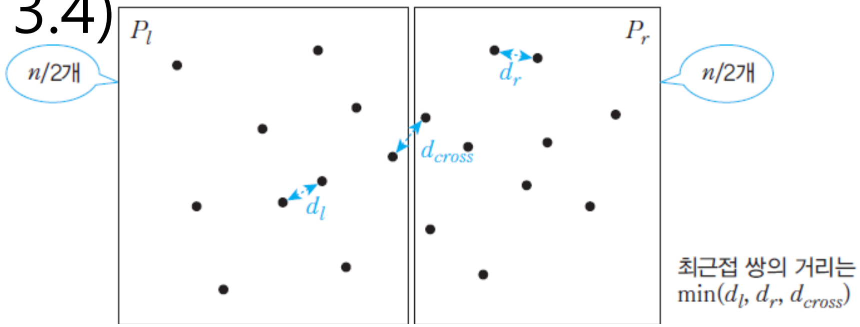
```
  In-Order  : D B E A F C
 Pre-Order  : A B D E C F
Post-Order  : D E B F C A
```



- 복잡도: $T(n) \in O(n)$

5.5 최근접 쌍의 거리 문제(심화)

- 억지 기법: 3.3절 복습(문제 3.4)
- 분할 정복 아이디어



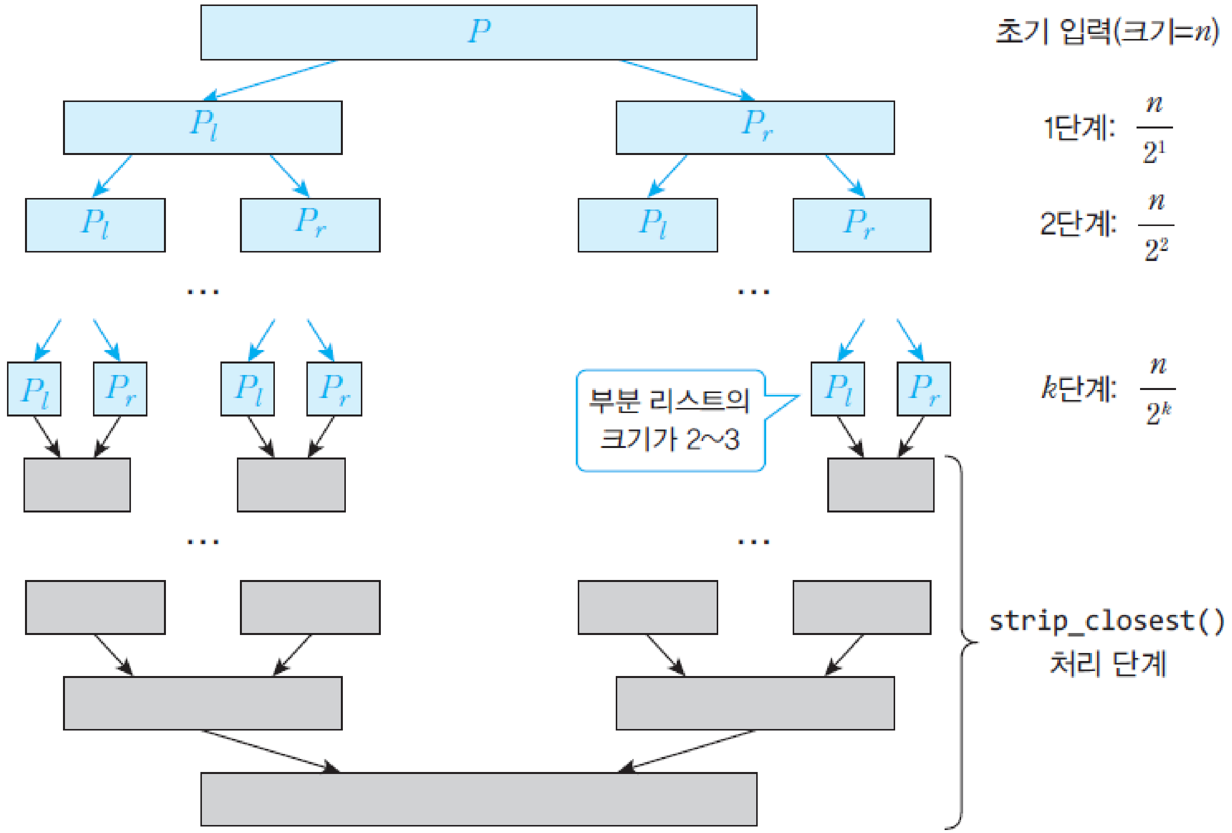
① **분할(Divide)**: 리스트 P를 두 개의 부분 리스트 P_l과 P_r로 분할한다. P가 x좌표로 정렬되어 있으므로 단순히 리스트의 왼쪽 절반을 P_l, 오른쪽을 P_r로 나눌 수 있다. 예를 들어, 그림 5.19에서는 20개의 점이 주어졌는데, 좌우를 각각 10개로 나누었다.

② **정복(Conquer)**: 만약 분할된 리스트의 크기(점들의 개수)가 3 이하인 경우는 바로 결과를 계산(정복)한다. 이때, 알고리즘 3.4의 억지 기법을 이용하면 된다. 만약 점들이 4개 이상이라면 분할 정복 기법을 다시 적용한다.

③ **결합(Combine, merge)**: 분할된 두 그룹 P_l과 P_r에서 최근접 쌍의 거리가 각각 d_l과 d_r로 계산되었다고 하자. 그렇다면 최종 결과는 d_l과 d_r, 그리고 양쪽 그룹에 걸쳐 있는 최근접 점의 쌍에 의한 거리 (d_{cross}) 중에서 가장 작은 값이 되어야 한다. 이 값을 구해 반환하면 된다.

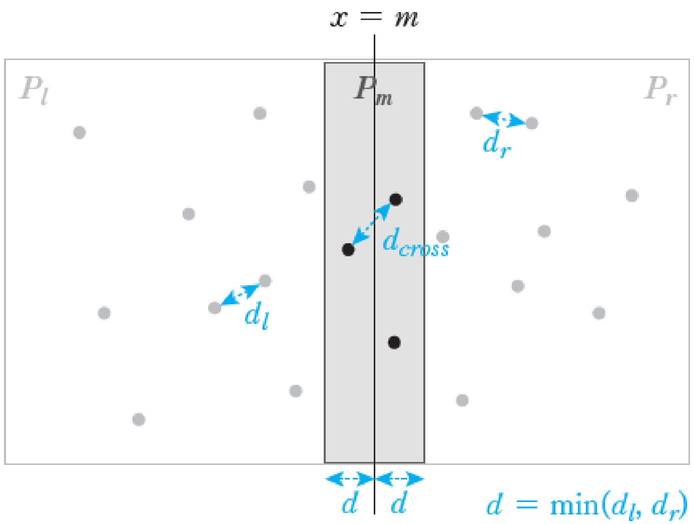
분할 정복 아이디어

- 최근접 쌍의 거리 알고리즘 전체 처리 과정

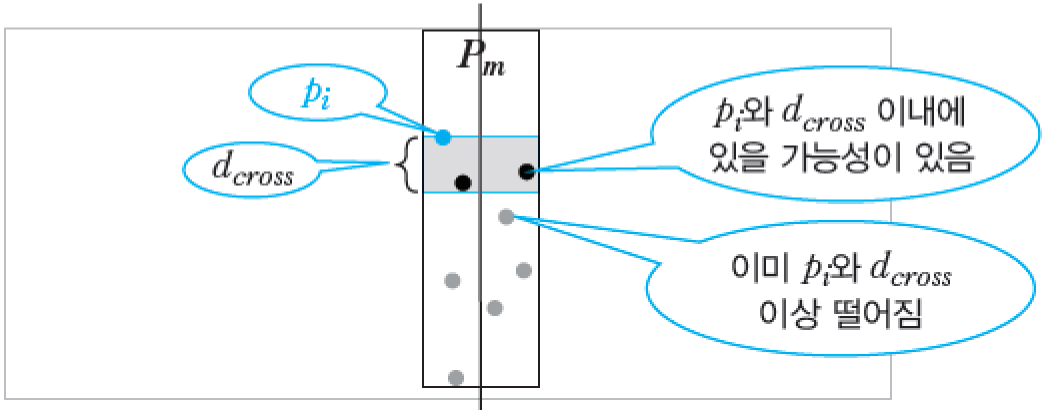


d_{cross} 의 계산

- 폭이 2d인 띠 영역에서만 처리하면 됨. Why?



- 추가적인 Tip



알고리즘: 띠 영역 처리(정렬 이용)

알고리즘 5.8 띠(strip) 영역 내에서 d보다 작은 최근접 쌍의 거리 찾기

```python
def strip_closest(P, d):
    n = len(P)
    d_min = d
    P.sort(key = lambda point: point[1])

    for i in range(n):
        j = i + 1
        # P[i].y와 P[j].y의 차이가 d_min 이내일 때까지만 처리
        while j < n and (P[j][1] - P[i][1]) < d_min:
            dij = dist(P[i], P[j])
            if dij < d_min :
                d_min = dij
            j += 1
    return d_min
```

알고리즘: 전체

알고리즘 5.9 최근접 쌍의 거리

```python
def closest_pair_dist(P, n):
    if n <= 3:
        return closest_pair(P)

    mid = n // 2
    mid_x = P[mid][0]

    dl = closest_pair_dist(P[:mid], mid)
    dr = closest_pair_dist(P[mid:], n-mid)
    d = min(dl, dr)

    Pm = []
    for i in range(n):
        if abs(P[i][0] - mid_x) < d:
            Pm.append(P[i])

    ds = strip_closest(Pm, d)
    return min(d, ds)
```

알고리즘 테스트 최근접 쌍의 거리

```python
p = [(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)]
p.sort(key = lambda point: point[0])      # 전처리: 점들을 x순으로 정렬
print("가장 가까운 두 점의 거리", closest_pair_dist(p, len(p)))
```

```
C:\WINDOWS\system32\cmd.exe
가장 가까운 두 점의 거리 1.4142135623730951
```

복잡도 분석

- 알고리즘 5.8의 strip_closest()
  - $O(nlog_2n)$
- 전체 처리 단계의 수
  - $k = log_2n$
- 복잡도
  - $O(k\ nlog_2n) = O(n(log_2n)^2)$

- 개선
  - strip_closest()를 $O(n)$에 처리할 수 있음. *How?*
  - 복잡도: $O(nlog_2n)$ → 이 문제의 최적 알고리즘

5.6 행렬 곱셈(심화)

두 개의 $n \times n$ 행렬 A와 B가 주어졌다. 이 행렬의 $A \times B$를 구하라.

- 억지 기법: $O(n^3)$
- 부분 행렬로 계산

$$\begin{array}{c} \\ n/2 \end{array} \overbrace{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}}^{n/2}_{n} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

  – 8번의 곱셈과 4번의 덧셈 사용

쉬트라쎈의 전략

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22}) \times B_{11}$$
$$M_3 = A_{11} \times (B_{12} - B_{22})$$
$$M_4 = A_{22} \times (B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12}) \times B_{22}$$
$$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$\boxed{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}}$$

- 연산 수
  - 7번 의 곱셈
  - 18번의 덧셈/뺄셈
  - → 억지기법보다 효율적일까?

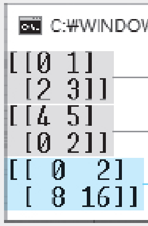
알고리즘

알고리즘 5.11 행렬 곱셈(쉬트라쎈 알고리즘)

```python
import numpy as np
def strassen(A, B):
    n = len(A)
    if n == 1:
        return A * B

    n2 = n//2
    A11, A12, A21, A22 = A[:n2, :n2], A[:n2, n2:], A[n2:, :n2], A[n2:, n2:]
    B11, B12, B21, B22 = B[:n2, :n2], B[:n2, n2:], B[n2:, :n2], B[n2:, n2:]

    M1 = strassen(A11+A22, B11+B22)
    M2 = strassen(A21+A22, B11)
    M3 = strassen(A11, B12-B22)
    M4 = strassen(A22, B21-B11)
    M5 = strassen(A11+A12, B22)
    M6 = strassen(A21-A11, B11+B12)
    M7 = strassen(A12-A22, B21+B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 + M3 - M2 + M6

    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))

    return C
```

알고리즘 테스트 쉬트라쎈 알고리즘

```python
A = np.array([[0,1], [2,3]])
B = np.array([[4,5], [0,2]])
C = strassen(A,B)
print( A )
print( B )
print( C )
```



$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 4 & 5 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 8 & 16 \end{bmatrix}$$

복잡도 분석

- 곱셈 연산을 기본 연산으로 한 경우
  - $T(n) = 7\,T\left(\frac{n}{2}\right)$, $T(1) = 1$
  - 마스터 정리
    - $a = 7, b = 2$
    - $f(n) = 0 = O(n^0)$ → $d = 0$

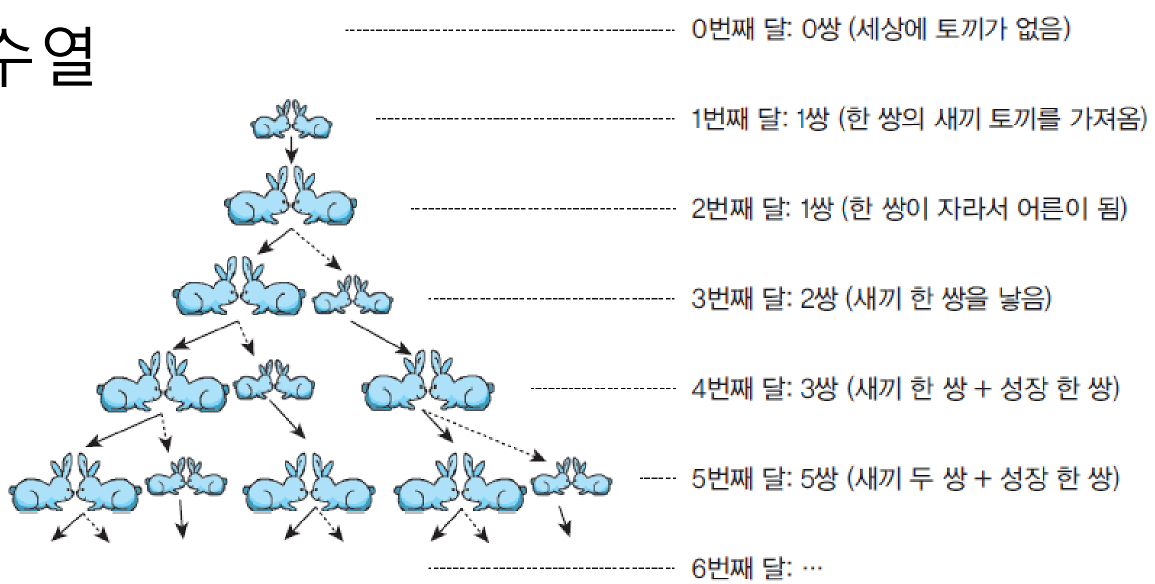
$$\boxed{T(n) = n^{\log_2 7} \approx n^{2.81} \in O(n^{2.81})}$$

- 덧셈/뺄셈을 기본 연산으로 한 경우
  - $T(n) = 7\,T\left(\frac{n}{2}\right) + 18(\frac{n}{2})^2$
  - → 복잡도는 같음

- 자주 사용되지는 않음 : Why?

5.7 피보나치수열과 분할 정복의 주의점

- 피보나치수열

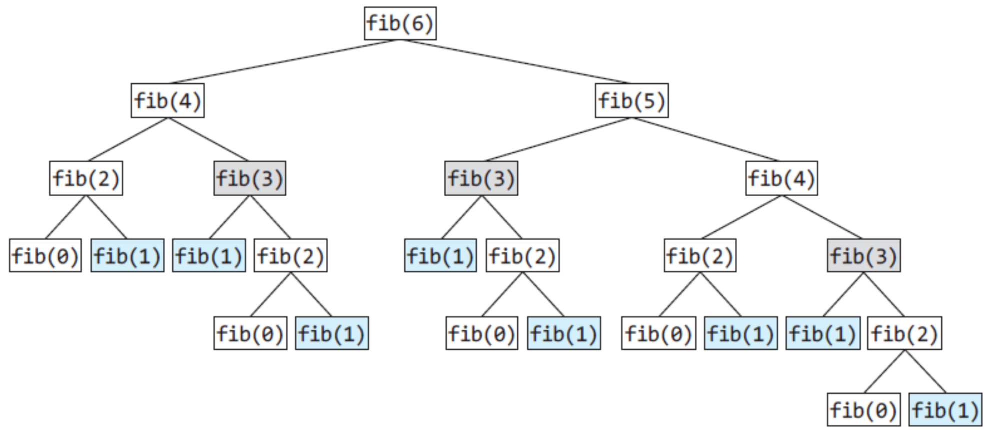


$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & \text{otherwise} \end{cases}$$

피보나치 수열 분할 정복 알고리즘

알고리즘 5.12 피보나치수열(분할 정복)

```
def fib(n) :
    if n == 0 :
        return 0
    elif n == 1 :
         return 1
    else :
        return fib(n - 1) + fib(n - 2)
```



- 같은 문제 중복 계산
  - $fib(n) \rightarrow fib(n-1), fib(n-2)$
  - 문제의 크기가 거의 두배로 커짐
  - $O(2^n)$

피보나치 수열 다른 알고리즘들

- 반복 구조: 알고리즘 5.13
  - $O(n)$
- 행렬 거듭제곱: 알고리즘 5.14

$$\boxed{F^n} = \begin{bmatrix} fib(2) & fib(1) \\ fib(1) & fib(0) \end{bmatrix}^n = \begin{bmatrix} fib(n+1) & fib(n) \\ \boxed{fib(n)} & fib(n-1) \end{bmatrix}$$

  - 복잡도: $O(log_2 n)$
- 분할 정복 기법 적용
  - 같은 부분 문제가 여러 번 반복되어 나타나지 않을 때만 사용!

실습 과제

감사합니다!