# 알고리즘및실습

제3장 분할 정복

## 1. 분할 정복 방법

분할 정복 방법이란 문제의 사례를 2개 이상의 더 작은 같은 종류의 문제에 대한 사례로 나누어 각 작은 사례에 대한 해답을 얻고, 이들을 결합하여 원 문제에 대한 해답을 얻는 알고리즘 설계 방법을 말한다. 보통 분할 정복 방법을 이용하여 문제를 해결하면 전주소사를 하는 것보다 성능을 개선할 수 있다. 하지만 성능이 급격하게 개선되는 것은 아니라 표준 성능 분류에서 한 수준 정도 성능을 개선해 준다. 즉, 지수 시간이 필요한 것을 다차 시간으로 바꾸어 주지는 못한다.

1장에서 살펴본 합병 정렬이 대표적인 분할 정복 기법을 사용하는 알고리즘이다. $O(n^2)$인 일반 정렬 알고리즘을 $O(n \log n)$로 개선해 준다. 합병 정렬은 왼쪽 절반을 정렬하고, 오른쪽 절반을 정렬한 다음, 이 둘을 결합하여 최종 정렬 결과를 얻는다. 이때 왼쪽 절반, 오른쪽 절반의 정렬은 재귀적 기법을 사용하여 정렬한다. 이처럼 분할 정복 기법은 전수조사와 마찬가지로 보통 재귀 알고리즘으로 구현한다. 이때 각 재귀 단계에서 문제의 크기가 줄어드는 정도와 비재귀 부분의 비용이 전체 성능에 가장 큰 영향을 준다. 당연하지만 크기가 빠르게 줄어들면 재귀 호출 트리의 깊이가 짧아 더 빨리 종료할 수 있으며, 비재귀 부분의 비용이 저렴할수록 효과적이다. 우리가 잘 알고 있는 이진 검색도 분할 정복 기법을 사용하는 알고리즘이다.

분할 정복 기법의 기본 알고리즘은 다음과 같다.

- 단계 1. 문제를 같은 유형의 작은 문제로 나눈다.
- 단계 2. 작은 문제를 재귀적으로 해결한다.
- 단계 3. 작은 문제의 결과를 결합하여 원 문제에 대한 해결책을 제시한다.

이 기법을 적용하기 위해서는 주어진 문제를 같은 유형의 작은 문제로 나눌 수 있어야 한다. 단계 2에서 문제를 나누는 과정은 해답을 쉽게 얻을 수 있는 수준까지 반복적으로 이루어진다. 모든 재귀 알고리즘이 그러하듯이 재귀를 종료하는 시점이 알고리즘 설계에 중요한 요소이다. 이처럼 큰 문제를 작은 문제로 반복적으로 나누기 때문에 분할 정복은 하향식 접근(top-down) 방법이라 한다. 10장에서 살펴보는 동적 프로그래밍 중 테뷸레이션 방법은 거꾸로 상향식 접근(bottom-up) 방법이다. 분할 정복 기법의 성능은 재귀 깊이와 단계 3의 시간 복잡도에 의해 결정된다.

## 2. 역쌍 개수 구하기

**역쌍 개수 구하기 문제**

- 입력. $n$개의 정수로 구성된 배열 $A$
- 출력. 배열 $A$에 존재하는 역쌍의 개수
- 역쌍의 정의. $i < j$에 대해 $A[i] < A[j]$인 쌍

주어진 입력 배열 $A$가 오름차순으로 정렬되어 있으면 역쌍이 하나도 없으며, 거꾸로 내림차순으로 정렬되어 있으면 모든 쌍이 역쌍이 된다. 따라서 역쌍의 개수는 0에서 $n(n-1)/2$이다.

이 문제는 이중 `for` 문을 이용하여 간단히 전수조사를 하여 해결할 수 있다. 이 알고리즘의 시간 복잡도는 $O(n^2)$이며 공간 복잡도는 입력 배열 외에 추가로 사용하는 공간이 일정하므로 $O(n)$이다.

**알고리즘 3.1** 역쌍 개수 구하기 전주조사 알고리즘

```
function INVERSIONCOUNT(A)
    count := 0
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            if A[i] > A[j] then ++count
    return count
```

어떤 문제를 분할 정복으로 해결하고 싶으면 먼저 합병 정렬처럼 입력을 왼쪽 절반과 오른쪽 절반으로 나누어 같은 문제의 소 문제로 각 절반을 해결할 수 있는지 생각해 보아야 한다. 왼쪽 절반에 존재하는 역쌍은 전체 역쌍 수에 포함해야 하고, 오른쪽 절반에 존재하는 역쌍도 전체 역쌍 수에 포함해야 한다. 그런데 각 절반에 있는 역쌍이 전부는 아니며, 역쌍 중 쌍의 첫 번째 요소가 왼쪽에 있고, 다른 하나는 오른쪽에 있는 역쌍까지 찾아 이 역쌍도 전체 역쌍 수에 포함해야 한다. 이것을 쪼개진 역쌍이라 하면 왼쪽 절반, 오른쪽 절반에 있는 역쌍은 재귀적으로 구하면 되고, 쪼개진 역쌍은 비재귀적 부분에서 구하면 된다. 쪼개진 역쌍만 효과적으로 찾을 수 있다면 전수조사하는 것보다 효과적으로 이 문제를 분할 정복하여 전체 역쌍을 찾을 수 있다. 합병 정렬에서는 비재귀적 부분에서 합병을 진행하며, 이것의 시간 복잡도는 $O(n)$이다. 쪼개진 역쌍을 $O(n)$에 찾을 수 있으면 분할 정복으로 역쌍을 찾는 것의 시간 복잡도는 합병 정렬과 마찬가지로 $O(n \log n)$이 된다.

쪼개진 역쌍을 어떻게 효과적으로 찾을 수 있을까? 합병 정렬에서 합병하는 것을 생각하여 보자. 왼쪽 절반의 첫 번째 요소와 오른쪽 절반의 첫 번째 요소를 비교하여 작은 것을 합병의 첫 번째 요소로 선택한다. 이때 왼쪽 절반에 있는 요소가 선택되면 이 요소로 구성되는 역쌍은 없다. 하지만 오른쪽 요소가 선택되면 왼쪽에 남아 있는 요소만큼 역쌍이 존재하게 된다. 즉, 합병 정렬을 수행하면서 합병할 때 쪼개진 역쌍을 구할 수 있다.

역쌍 개수 구하기는 추천 시스템에서 많이 사용하는 협업 필터링을 구현할 때 활용할 수 있다. 예를 들어 정해진 10개의 영화에 대해 길동이, 춘향이, 몽룡이가 선호하는 순서가 있다고 하자. 그러면 춘향이와 몽룡이 중 길동이와 취향이 더 가까운 사람은 누구일까? 춘향이와 몽룡이의 선호 순서를 길동이의 기준을 이용하여 다음과 같은 정수 배열을 만들 수 있다.

$$\text{춘향} = [6, 7, 8, 1, 2, 4, 3, 5, 9, 10]$$
$$\text{몽룡} = [4, 2, 1, 5, 3, 7, 8, 6, 10, 9]$$

즉, 길동이가 6번째로 선택한 영화를 춘향이는 가장 좋아한다는 것이며, 길동이가 4번째로 선택한 영화를 몽룡이는 가장 좋아한다는 것이다. 이 두 배열에서 역쌍을 구해보면 춘향이 배열은 16개 역쌍이 있고, 몽룡이 8개 역쌍이 있다. 그러면 길동이와 취향이 더 가까운 친구는 몽룡이다.

역쌍 개수 구하기와 유사한 문제로 연속 부분 구간의 합계 최댓값 구하기 문제가 있다. 이 문제도 모든 연속

**알고리즘 3.2** 역쌍 개수 구하기 알고리즘

```
function countInversion(A[])
    return countInversion(1, n, A)
function countInversion(lo, hi, A[])
    if lo < hi then
        mid := lo + (hi − lo)/2
        leftInv := countInversion(1, mid − 1, A)
        rightInv := countInversion(mid, hi, A)
        splitInv := countSplitInversion(lo, mid, hi, A)
        return leftInv + rightInv + splitInv
    return 0
function countSplitInversion(lo, mid, hi, A[])
    left[] := A[lo...mid − 1]
    right[] := A[mid...hi]
    L, R, i, count := 1, 1, lo, 0
    while L <= mid − 1 and R <= hi do
        if left[L] < right[R] then
            A[i++] := left[L++]
        else
            A[i++] := right[R++]
            count += len(left)−L + 1
    while L <= mid − 1 do
        A[i++] := left[L++]
    while R <= hi do
        A[i++] := right[R++]
    return count
```

부분 구간의 합계를 이중 **for** 문을 이용하여 계산하여 최댓값을 구할 수 있다. 이것의 시간 복잡도는 $O(n^2)$이다. 이 문제도 합병 정렬과 동일하게 왼쪽 절반, 오른쪽 절반으로 나누면 찾고자 하는 구간은 왼쪽 절반에 있을 수 있고, 오른쪽 절반에 있을 수 있고, 구간이 왼쪽에서 시작하여 오른쪽에서 끝날 수 있다. 이렇게 쪼개진 구간을 선형에 찾을 수 있으면 역쌍 개수 구하기와 마찬가지로 $O(n \log n)$으로 성능을 개선할 수 있다. 이 문제는 실제 동적 프로그래밍 기법을 적용하여 $O(n)$에 해결할 수 있다.

## 3. 빠른 합

1부터 $n$까지 합은 $n(n+1)/2$ 공식을 이용해 쉽게 계산할 수 있다. 하지만 이 문제도 분할 정복을 이용해 해결할 수 있다. $n$이 짝수일 때 1부터 $n$까지 합은 다음과 같이 1부터 $n/2$까지 합과 $n/2+1$부터 $n$까지 합으로 나눌 수 있다.

$$
\begin{aligned}
1 + 2 + \cdots + n &= \left(1 + 2 + \cdots + \frac{n}{2}\right) + \left(\frac{n}{2} + 1 + \frac{n}{2} + 2 + \cdots \frac{n}{2} + \frac{n}{2}\right) \\
&= \left(1 + 2 + \cdots + \frac{n}{2}\right) + \frac{n}{2} \times \frac{n}{2} + \left(1 + 2 + \cdots + \frac{n}{2}\right)
\end{aligned}
$$

따라서 다음이 성립한다.

$$\mathrm{sum}(n) = 2 \times \left(\frac{n}{2}\right) + \frac{n^4}{4}$$

하지만 위 식은 $n$이 짝수일 때만 성립한다. $n$이 홀수이면 다음을 이용할 수 있다.

$$\mathrm{sum}(n) = \mathrm{sum}(n-1) + n$$

위 식에서 $\mathrm{sum}(n-1)$를 분할 정복하여 구할 수 있다.

1부터 $n$까지 합을 이와 같이 분할 정복으로 구하였을 때 시간 복잡도는 어떻게 되는지 분석하여 보자. 원래 직관적인 합계는 $n-1$개의 합이 필요하다. 따라서 직관적인 합계의 시간 복잡도는 $O(n)$이다. 분할 정복은 $\log n$번의 재귀 호출이 필요하고, 내부적으로 곱셈 2번, 나눗셈 2번이 필요하다. 물론 이 곱셈과 나눗셈은 2를 이용하는 것이기 때문에 비트 연산으로 매우 효과적으로 할 수 있다. 따라서 비재귀적 부분의 비용은 $O(1)$이므로 전체 비용은 $O(\log n)$이다. 시간 복잡도만 보면 분할 정복 방법이 더 효과적이다. 더 효과적이지만 실제 합계를 분할 정복을 이용하여 구하지는 않을 것이다. 간단하게 공식을 이용하여 $O(1)$에 계산할 수 있기 때문이다.

## 4. 수의 거듭제곱

$n^m$은 기본적으로 $m-1$번의 곱셈을 이용하여 구할 수 있다. 수의 거듭 제곱을 구할 때 가장 많이 사용하는 알고리즘은 square-and-multiply이다. 예를 들어 $3^{53}$은 총 52번의 곱셈이 필요하지만 square-and-multiply 기법을 적용하면 8번의 곱셈으로 계산할 수 있다. 기본적인 생각은 $m$을 2의 거듭제곱 합의 표현을 구하는 것이다. 예를 들어 $53 = 32 + 16 + 4 + 1$이다. 따라서 다음이 성립한다.

$$3^{53} = 3^{32} \cdot 3^{16} \cdot 3^{4} \cdot 3$$

그런데 $3^{32}$, $3^{16}$, $3^{4}$은 총 5번의 곱셈을 통해 구할 수 있기 때문에 전체를 8번의 곱셈을 통해 구할 수 있다. 이것을 다르게 표현하면 53의 비트 표현인 110101를 구하고 1인 위치에 해당하는 3의 거듭제곱을 모두 곱하면 우리가 원하는 $3^{52}$를 계산할 수 있다.

**알고리즘 3.3** square-and-multiply 알고리즘

```
function POW(n, m)
    ret := 1
    while m > 0 do
        if m%2 = 1 then
            ret := ret × n
        n := n × n
        m := m/2
    return ret
```

이 알고리즘에서 `while` 반복문 내에서 $m$을 계속 2로 나누면서 $m$이 홀수이면 지금까지 구한 $n$의 거듭제곱을 최종 결과에 곱하고 있다. 이것은 비트 표현이 1인 위치의 $n$의 거듭제곱을 $n^0$부터 차례로 구하여 이를 이용하여 $n^m$을 구하는 과정이다. 이 알고리즘에서 `while` 문은 총 $\log m$번 반복하며, 반복문 내에 연산은 일정하므로 이 알고리즘의 시간 복잡도는 $O(\log m)$이다.

square-and-multiply 알고리즘을 모를 경우에는 분할 정복을 이용하여 $n^m$을 구하면 같은 시간 복잡도 알고리즘을 얻을 수 있다. $m$이 짝수이면 $n^m = n^{m/2} \times n^{m/2}$이다. $m$이 홀수이면 $n^{m-1} \times n$를 이용하여 분할 정복할 수 있다. 이것의 알고리즘은 알고리즘 3.4와 같다. 이 알고리즘의 재귀 호출 깊이는 $O(\log m)$이고, 비재귀적 부분에서 비용은 $O(1)$이므로 이 알고리즘의 시간 복잡도는 $O(\log m)$이다.

## 5. 행렬 곱셈

행렬 곱셈 문제

- 입력. 두 개의 $n \times n$ 행렬 $X$와 $Y$, $n$은 2의 거듭제곱
- 출력. $n \times n$ 행렬 $Z = X \times Y$

**알고리즘 3.4** 수의 거듭제곱 분할 정복 알고리즘

```
function POW(n, m)
    if m = 0 then return 1
    if m = 1 then return n
    ret := 1
    if m%2 = 0 then
        ret := POW(n, m/2)
        ret := ret × ret
    else
        ret := n×POW(n, m − 1)
    return ret
```

**알고리즘 3.5** 기본 행렬 곱셈 알고리즘

```
function MATRIXMULTIPLY(X[][], Y[][])
    Z := [][]
    for i := 1 to n do
        for j := 1 to n do
            Z[i][j] := 0
            for k := 1 to n do
                Z[i][j] := Z[i][j] + X[i][k] × Y[k][j]
    return Z
```

이 문제는 1장에서 살펴본 초등학교 곱셈 알고리즘처럼 수학 시간에 배운 방법을 적용한 알고리즘 3.5를 이용하여 구할 수 있다. 이 알고리즘의 시간 복잡도는 $O(n^3)$이다. 그런데 이 문제의 입력 크기는 $2n^2$이므로 $O(n^2)$보다 더 빠르게 해결할 수 없다. 이 문제도 당연히 분할 정복으로 해결할 수 있다.

행렬 곱셈은 각 행렬을 4개의 $n/2$ 행렬로 나누어 다음을 이용하여 계산할 수 있다.

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

위 식을 그대로 이용하여 분할 정복하면 총 8개의 재귀 호출이 필요하며, 이것의 시간 복잡도를 계산하면 $O(n^3)$이다. 이 알고리즘에 대한 자세한 분석은 4장으로 미룬다.

Strassen은 다음과 같이 계산하여 8개의 재귀 호출을 7개로 축소하였다.

$$P_1 = A(F-H), P_2 = (A+B)H, P_3 = (C+D)E, P_4 = D(G-E),$$
$$P_5 = (A+D)(E+H), P_6 = (B-D)(G+H), P_7 = (A-C)(E+F)$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{bmatrix}$$

재귀 호출을 하나 줄이는 것이 큰 효과가 없을 것으로 생각할 수 있지만 이것의 효과는 일회성이 아니라 재귀 호출 트리에 계속 영향을 주는 것이기 때문에 전체 재귀 호출 수에는 큰 영향을 준다. 실제 $O(n^3)$의 시간 복잡도가 $O(n^{2.81})$으로 향상된다.

# 6. 가장 가까운 쌍 찾기

**가장 가까운 쌍 찾기 문제**

- 입력. $n(\geq 2)$개의 $p_i = (x_i, y_i)$ 좌표
- 출력. 가장 가까운 좌표 $p_i$, $p_j$
- 가장 가까운 좌표는 euclidean distance가 가장 짧은 두 좌표

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

**알고리즘 3.6** 가장 가까운 쌍 찾기 전수조사 알고리즘

```
function SHORTESTPAIR(A[])
    min := ∞
    ret := []
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            d := DISTANCE(A[i], A[j])
            if min > d then
                min, ret[0], ret[1] := d, A[i], A[j]
    return ret
```

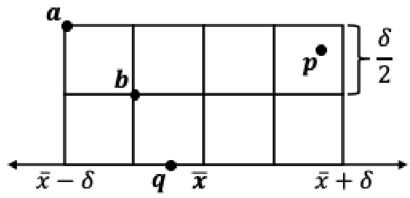
이 문제는 알고리즘 3.6과 같이 전수 조사하여 해결할 수 있다. 이 알고리즘의 시간 복잡도는 $O(n^2)$이다. 좌표의 $y$ 좌표 값이 모두 같거나 $x$ 좌표가 같으면 이들을 반대 좌표를 기준으로 정렬한 후에 인접 좌표를 비교하면 가장 가까운 쌍을 찾을 수 있다. 이 경우 시간 복잡도는 $O(n \log n)$이 되므로 전수조사하는 것보다 효과적이다.

이것을 분할 정복하여 보자. 역쌍 개수를 구한 방법과 비슷하게 이 문제도 해결할 수 있다. 전체 좌표를 $x$ 좌표를 기준으로 왼쪽 절반, 오른쪽 절반으로 나눌 수 있다. 물론 $y$ 좌표를 기준으로 위쪽 절반, 아래쪽 절반으로 나눌 수 있다. 이렇게 $x$ 좌표를 기준으로 나누면 가장 가까운 쌍은 왼쪽 절반에 있을 수 있고, 오른쪽 절반에 있을 수 있고, 양쪽으로 나누어져 있을 수 있다. 양쪽으로 나누어져 있는 쌍을 $O(n)$에 찾을 수 있으면 역쌍 개수를 구하는 알고리즘, 합병 정렬 알고리즘처럼 $O(n \log n)$에 할 수 있다. 그러면 어떻게 쪼개진 쌍을 $O(n)$에 찾을 수 있을까?

쪼개진 쌍은 서로 반대쪽 있는 쌍 간의 비교만 필요하다. 총 $\frac{n}{2} \times \frac{n}{2} = \frac{n^2}{4}$이므로 이를 직관적으로 구하면 $O(n^2)$이 필요하다. 따라서 비교해야 하는 쌍을 줄여야 비용을 $O(n)$으로 줄일 수 있다. 이 문제는 지금까지 본 문제와 달리 기발한 생각이 필요하다. 쉽게 생각해 낼 수 있는 방법은 아니다. 많은 문제는 앞으로 학습하게 되는 알고리즘 설계 방법 중 하나를 직관적으로 적용하면 해결할 수 있다. 하지만 이 문제처럼 기발한 방법을 생각해 내지 못하면 해결하기 어려운 문제도 있다.

이 문제는 두 가지 기발한 생각을 이용하여 해결하고 있다. 첫 번째는 조금 쉽게 생각해 낼 수 있는 것이다. 왼쪽 절반, 오른쪽 절반에서 구한 가장 가까운 쌍의 거리를 이용하는 것이다. 왼쪽 절반에서 구한 가장 가까운 쌍 거리, 오른쪽 절반에서 구한 가장 가까운 쌍 거리 중 더 짧은 거리를 $\delta$라 하자. 이 거리보다 기본적으로 멀리 있는 쌍의 비교는 필요가 없다. $|x_i - x_j| > \delta$인 쌍의 비교는 제외할 수 있다. 절반을 나누기 위해 $x$ 좌표를 기준으로 정렬하기 때문에 좌표들 중 $\bar{x}$가 중간 좌표에 해당하면 $\bar{x} - \delta$와 $\bar{x} + \delta$ 사이에 있는 좌표만 고려하면 된다.

이렇게 하여 불필요한 좌표들을 제거하더라도 비교해야 하는 좌표 쌍의 수가 $O(n)$이 된다는 보장은 없다. 이 때문에 두 번째 기발한 생각의 적용이 필요하다. 이 부분은 쉽게 생각해 낼 수 있는 부분은 아니다. 우리는 $\delta$보다 거리가 짧은 쌍이 있는지 조사하고 싶은 것이다. 따라서 $\bar{x} - \delta$와 $\bar{x} + \delta$ 집합에 있는 모든 쌍을 비교해야 하는 것은 아니다. $\delta$ 이내에 있을 수 있는 좌표 간에 비교만 필요하다. 그런데 이들을 $y$ 좌표를 기준으로 정렬하면 $|y_i - y_j| > \delta$인 쌍의 비교는 필요 없다.




<그림 3.1> $y$ 좌표로 정렬되어 있을 때 $q$가 좌표들 중 가장 $y$ 좌표가 작은 경우

두 번째 생각은 이를 이용하는 것이다. 면밀히 분석해 보면 $\bar{x}-\delta$와 $\bar{x}+\delta$ 집합에 있는 하나의 좌표와 비교해야 하는 좌표의 수는 최대 7개이다. 그 이유는 그림 3.1을 생각해 보면 알 수 있다. 이 그림에서 $q$는 $\bar{x}-\delta$와 $\bar{x}+\delta$ 집합에 있는 좌표이며, $y$ 좌표가 가장 작은 좌표이다. 그러면 이 그림에는 최대 7개의 다른 좌표가 존재할 수 있다. 그 이유는 그림에 나타나는 각 정사각형 영역에 두 개의 좌표가 존재할 수 없기 때문이다. 존재할 경우에는 $\delta$가 왼쪽 절반 또는 오른쪽 절반의 가장 가까운 거리라는 것에 모순이 된다. 예를 들어 $a$와 $b$와 같은 좌표가 있다고 하자. 이 두 좌표의 거리는 $\frac{\delta}{\sqrt{2}}$가 되기 때문이다.

**알고리즘 3.7** 가장 가까운 쌍 찾기 분할 정복 알고리즘

**function** CLOSESTPAIR($P_x[], P_y[]$)
  **if** LEN($P_x$)<= 3 **then**
    **return** BRUTEFORCECLOSESTPAIR($P_x$)
  ▷ *$L_x$, $L_y$, $R_x$, $R_y$ must be obtained by $O(n)$* ◁
  $L_x$ := left of $Px$
  $L_y$ := left of $P_x$ sorted by $y$ coordinate
  $R_x$ := right of $P_x$
  $R_y$ := right of $P_x$ sorted by $y$ coordinate
  $l_1, l_2$ := CLOSESTPAIR($L_x, L_y$)
  $r_1, r_2$ := CLOSESTPAIR($R_x, R_y$)
  $d$ := MIN(DISTANCE($l_1, l_2$), DISTANCE($r_1, r_2$))
  ret := closest pair between ($l1, l2$) and ($r_1, r_2$)
  **return** CLOSESTSPLITPAIR($P_x, Py, d$, ret)

알고리즘 3.7에 좌표를 전달하기 전에 원래 좌표 배열이 있으면 이것을 복제한 후에 $x$ 축을 기준으로 오름차순 정렬하여 $P_x$를 만들고, $y$ 축을 기준으로 오름차순 정렬하여 $P_y$를 만들어 이 둘을 인자로 전달해야 한다. 이때 좌표의 $x$ 값이 같은 좌표가 여러 개 있을 수 있으므로 $x$ 값이 같으면 $y$를 기준으로 오름차순으로 정렬해야 한다. 이 알고리즘이 합병 정렬처럼 $O(n \log n)$이 되기 위해서는 비재귀적인 부분의 비용이 $O(n)$이 되어야 한다. 따라서 내부적으로 $L_x$, $L_y$, $R_x$, $R_y$를 얻을 때 정렬 알고리즘을 사용하면 안 된다. $L_x$와 $R_x$는 쉽게 $P_x$를 절반으로 나누면 얻을 수 있다. 여기서 핵심은 $L_y$와 $R_y$를 $L_x$와 $R_x$를 정렬하여 얻는 것이 아니라 $P_y$를 이용하여 $O(n)$에 확보하여야 한다. 이렇게 분할 정복할 준비가 되면 왼쪽 절반과 오른쪽 절반을 재귀 호출하여 왼쪽 절반에서 가장 가까운 쌍을 찾고, 오른쪽 절반에서 가장 가까운 쌍을 찾으면 된다.

**알고리즘 3.8** 쪼개진 쌍 찾기 알고리즘

**function** CLOSESTSPLITPAIR($P_x[], P_y[], d$, bestPair)
  $\bar{x} := P_x[\text{LEN}(P_x)/2].x$
  ▷ *$S_y$ must be obtained by $O(n)$* ◁
  $S_y := \{q_1, q_2, \ldots, q_l\}$, $q_i = (x_i, y_i)$ where $\bar{x}-d < x_i < \bar{x}+d$ and $y_i <= y_{i+1}$
  best := $d$
  **for** $i := 1$ **to** $l-1$ **do**
    **for** $j := 1$ **to** MIN(()7, $l-i$) **do** d := DISTANCE($q_i, q_{i+j}$)
      **if** $d <$ best **then**
        best := $d$
        bestPair := $(q_i, q_{i+j})$
  **return** bestPair

쪼개진 쌍은 **closesetSplitPair**을 이용하여 얻는데, 이때 앞서 설명한 두 가지 기발한 생각을 이용하여 $O(n)$에 가장 가까운 쌍을 찾는다. **closesetSplitPair**의 알고리즘은 알고리즘 3.8과 같다. 이 알고리즘은 이중 **for** 문으로 구성되어 있지만 내부 **for** 문은 항상 7 이하이므로 시간 복잡도는 $O(n)$이다. 위 알고리즘에서 테스트해야 하는 $S_y$ 리스트를 $O(n)$에 확보해야 한다. 이때에는 **closestPair**처럼 $P_y$를 이용하여야 $S_y$를 만든다.

## 7. 분할 정복이 적합하지 않는 경우

분할 정복을 적용하여 문제를 해결하면 다차 시간 알고리즘의 차수를 줄여줄 수 있다. 하지만 다음과 같은 경우에는 분할 정복을 사용하여도 효과가 없다.

- 경우 1. 입력 크기가 $n$인 문제가 입력 크기가 거의 $n$인 두 개 이상의 사례로 분할되는 경우
- 경우 2. 입력 크기가 $n$인 것이 거의 $n$개의 입력 크기가 $n/b$인 사례로 분할되는 경우

예를 들어 피보나찌 수가 경우 1에 해당한다. 피보나찌 수 $f(n) = f(n-1) + f(n-2)$이다. 크기가 $n$인 문제를 $n-1$, $n-2$인 문제로 나누어 해결하는 것은 재귀 호출 깊이가 너무 깊기 때문에 바람직하지 않다. 경우 2는 너무 많은 재귀 호출이 이루어지기 때문에 $b$에 따라 전체 재귀 호출 수가 너무 많아진다. 또 앞서 언급하였듯이 입력 크기가 적절한 크기로 줄어들고, 나누어지는 사례가 적더라도 비재귀적 부분의 비용이 효과적이지 않으면 분할 정복하는 효과가 없다.

## 퀴즈

1. 분할 정복과 관련된 다음 설명 중 틀린 것은
   ① 문제의 사례를 더 작은 사례로 나누어 해결하는 방법이다. 이때 작은 사례는 원 사례와 다른 유형의 문제이어도 분할 정복할 수 있다.
   ② 보통 다차시간 알고리즘의 성능을 개선하기 위해 사용한다.
   ③ 분할 정복 알고리즘에서 작은 사례는 재귀 호출을 통해 해결하기 때문에 분할 정복에서 가장 힘든 부분은 작은 사례의 답들로부터 원 문제의 해답을 효과적으로 얻는 방법을 고안하는 부분이다.
   ④ 한 번에 나누어지는 작은 사례들이 너무 많거나 나누어진 사례의 입력 크기가 원 문제의 크기와 차이가 없으면 분할 정복의 효과가 없다.
2. 크기가 $n$인 정수 배열에서 역쌍의 범위는?
   ① $0, n^2$
   ② $1, n(n-1)/2$
   ③ $0, n(n-1)/2$
   ④ $1, n^2$
3. 수의 거듭제곱 문제도 분할 정복 기법을 적용해 해결할 수 있다. 수 $n$의 $m$ 거듭제곱을 분할 정복으로 구하는 알고리즘 관련 다음 설명 중 틀린 것은?
   ① 기저 사례는 $m$이 0 또는 1일 때이며, $m$이 0이면 1, $m$이 1이면 $n$을 반환한다.
   ② $m$이 짝수일 때 $n$의 $m/2$ 거듭제곱을 구하는 재귀 호출로 두 번 호출하여 $n$의 $m$ 거듭제곱을 구한다.
   ③ $m$이 홀수이면 $m-1$을 이용하여 재귀 호출한 결과에 $n$을 곱하여 $n$의 $m$ 거듭제곱을 구한다.
   ④ 재귀 호출마다 $m$의 크기를 반으로 줄기 때문에 재귀 호출의 깊이는 $\log m$에 비례한다.
4. 가장 가까운 쌍 찾기를 해결하는 분할 정복 알고리즘 관련 다음 설명 중 틀린 것은?
   ① 이 알고리즘도 역쌍과 마찬가지로 좌표를 반으로 나누어 각 반에서 가장 가까운 쌍을 찾고, 양쪽으로 분리된 쌍 중에서 가장 가까운 쌍을 찾아 최종적으로 가장 가까운 쌍을 찾는다.

② 양쪽으로 분리된 쌍 중에서 가장 가까운 쌍을 O(n)에 찾기 위해 좌표 중 서로 비교할 필요가 없는 좌표를 걸러낸다. 이때 왼쪽 절반과 오른쪽 절반에서 찾는 가장 가까운 쌍의 거리 정보를 이용한다.

③ 이 알고리즘은 한 축으로만 정렬하여 알고리즘을 진행한다.

④ 양쪽으로 분리된 쌍 중에서 가장 가까운 쌍을 찾을 때 최초 양 쪽으로 나누었을 때 사용한 축과 다른 축을 중심으로 정렬하여 비교한다. 예를 들어 최초 x 축으로 반으로 나누었으면 분리된 쌍에서는 y 축으로 정렬하고 y 축 값이 가장 작은 좌표를 그다음 7개와 비교하여 가장 가까운 쌍을 찾는다.

## 연습문제

1. 중복 요소가 있는 오름차순으로 정렬된 배열에서 정수 $k$가 주어지면 $k$의 첫 번째 시작 위치와 마지막 위치를 찾아주는 시간 복잡도가 $O(\log n)$인 분할 정복 알고리즘을 제시하라. 힌트. 이진 검색을 하여 정수 $k$를 찾았을 때, 왼쪽과 오른쪽 선형 검색하는 것은 최악의 경우 비용이 $O(n)$이 된다.
2. $n$개의 정수가 주어졌을 때, 합계가 가장 큰 연속된 부분 구간을 찾는 분할 정복 알고리즘을 제시하라. 이 구간의 크기는 최소 1이상이어야 한다.
3. 소문자로만 구성된 문자열 $s$와 정수 $k$가 주어지면 $s$의 부분 문자열 중 부분 문자열의 모든 문자의 빈도수가 $k$ 이상인 가장 긴 부분 문자열의 길이 찾아라. 예를 들어 "mmmxx"와 $k = 3$이 주어지면 "mmm"이 이 조건을 만족하는 가장 긴 부분 문자열이므로 3을 반환해야 한다. 이 문제를 해결하는 분할 정복 알고리즘을 제시하라.' 힌트. 전형적인 분할 정복처럼 왼쪽 절반과 오른쪽 절반으로 나누어 분할 정복하는 것이 아님