4. 분할정복

Divide : 문제를 같지만 더 작은 단위로 나눈다.

Conquer : 분할된 작은 문제들을 해결한다.

Combine : 작은 문제에 대한 해결책을 본래의 문제로 조합하여 대입한다.

Recursive case : 분할된 소문제들이 재귀적으로 풀 만큼 충분히 크면 Recursive case라고 부른다.

소문제들을 풀다보면 원래 문제와 동일하지 않을 때도 있는데, 이 때는 Combine 단계의 일부이다.

Recurrences

재귀식을 푸는 세 가지 방법 : 점근적인 Θ 또는 O 상한을 얻는 방법

- **Substitution method** : 수학적 귀납법을 통해 추측을 증명
- **recursion-tree method** : 재귀의 각 단계에서 발생하는 비용을 노드로 하는 트리 생성.
- **master method** :

$$T(n) = aT(n/b) + f(n)$$

-

위와 같은 형태의 재귀식에 대한 방법

Technicalities inrecurrences

보통 floors, ceilings, boundary condition는 생략한다. 하지만 언제 중요한지는 경험을 통해 공부해 놓을 필요가 있음.

4.1 최대 부분 배열 문제

주식을 할때 가장 싸게 사서 비싸게 판다. 하지만 내가 거래하는 기간동안에는 전체 관점에서 가장 비싼 지점과 가장 싼 지점이 존재하지 않을 수 있다. 그렇다면 내 범위 내에서의 최고와 최저를 찾아야 함.

내 범위 내에서 가장 싼 가격 찾기 + 그 오른쪽 지점에서 가장 비싼 가격 찾기 => 둘의 차이

내 범위 내에서 가장 비싼 가격 찾기 + 그 왼쪽 지점에서 가장 싼 가격 찾기 => 둘의 차이

이 둘을 비교하여 더 큰 값을 선택하면 된다.

하지만

<u>최대의 이익은 가장 높은 가격에서 팔거나 가장 낮은 가격에서 사는 것 둘 중 한곳에 존재하지 않을 수도 있다.</u>

brute-force

모든 경우의 수를 실행해보고 비교한다.

transformation

가격을 보지 않고 가격의 변화를 본다.

divide-and-conquer

기간을 나눈다.

1. 처음부터 중간까지의 부분배열에 있는 경우
2. 중간부터 끝까지의 부분배열에 있는 경우
3. 중간지점을 가로지르는 경우

1,2 의 경우는 원래 문제보다 더 작아진 경우이기 때문에 재귀적으로 해결할 수 있음.

3의 의사코드는 다음과 같다.

`FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)`

```
1  left-sum = -∞
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum = ∞
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

1~7 행 : 왼쪽 절반에서 최대 부분 배열 찾기

8~14 행 : 오른쪽 절반에서 최대 부분 배열 찾기

15행 : 중간지점을 가로지르는 최대 부분 배열을 표시하는 인덱스 max-left와 max-right와 함께, 이 둘을 끝으로 하는 부분배열의 합을 반환하기

```
if high == low
    return (low,high, A[low])
else mid = (low+high)/2 의 내림
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A,low,mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
```

```
14        else return (cross-low, cross-high, cross-sum)
15
```

각 케이스 별로 최대 부분 배열을 구하는 의사코드.

Analyzing the divide-and-conquer algorithm

소문제의 크기가 정수로 되도록 하기 위해 원래 문제크기를 2의 거듭제곱으로 가정한다. 수행 시간을 구해보면 1~3행은 상수. 4~5행은 2T(n/2), 6행은 Θ(n), 7~11행은 Θ(1). 그 결과

T(n) = Θ(1) + 2T(n/2) + Θ(n) + Θ(1) = 2T(n/2) + Θ(n)

따라서

T(n) = Θ(n log n)

Exercises

4.1-1

문제 : A의 모든 원소가 음수일 때 `FIND-MAXIMUM-SUBARRAY`는 무엇을 반환하는가?

나의 답 : 더 할 수록 더 작은 값이 되므로 가장 큰 하나의 원소만 반환한다.

4.1-2

문제 : 최대 부분 배열 문제를 해결하는 brute-force 방식의 의사코드를 작성하라. Θ(n^2)의 수행시간을 가진다.

답 :

```
n = len(A)
max_sum = -∞
for i = 1 to n
    current_sum = 0
    for j = i to n
        current_sum = current_sum + A[j]
        if current_sum > max_sum
            max_sum = current_sum
            start = i
            end = j
    return (start, end, max_sum)
```

하나씩 더 해보면서 최댓값을 업데이트 한다.

4.1-3

자신의 컴퓨터에서 브루트포스와 재귀 알고리즘을 통해 최대 배열 문제를 해결해라. problem size인 n_0이 몇일 때 재귀 알고리즘이 브루트포스 알고리즘과 교차되는가? 재귀 알고리즘을 문제 사이즈가 n_0보다 작을 땐 브루트포스 알고리즘을 사용하도록 수정하라. 교차지점이 바뀌는가?

나의 답 : 결국 재귀 알고리즘이 더 효율적인 방법이 되는 n의 크기는 일정하므로 교차점이 변경되지 않을 것이다.

4.1-4

합이 0인 비어있는 부분 배열에도 적용하기 위해 최대 부분 배열 문제의 정의를 변경한다고 가정하자. 비어있는 부분배열이 결과가 되지 않도록 하는 알고리즘들을 가능하게 하려면 어떻게 수정해야 하는가?

나의 답 : 부분 배열을 도출할 때 0과 비교하는 로직을 추가하면 된다. 0보다 작으면 비어있는 배열을 결과로 반환하도록 한다.

4.1-5

최대 부분 배열 문제를 위한 비재귀적이고 선형함수를 시간복잡도로 가지는 알고리즘을 개발하는데 다음 아이디어를 사용하라. 배열의 왼쪽 끝에서 시작하고 오른쪽으로 진행하며 지금까지의 최대 부분배열을 추적한다. A[1..j]의 최대 부분 배열을 알고 있다면, 다음 관찰을 사용하여 인덱스 j+1에서 끝나는 최대 부분 배열을 찾아 답을 확장하라: A[1..j+1]의 최대 부분 배열은 A[1..j]의 최대 부분 배열이거나, 일부 1 <= i <= j+1에 대한 A[i..j+1]의 부분 배열이다. 인덱스 j에서 끝나는 최대 부분 배열을 알고 있는 것을 기반으로 A[i..j+1] 형태의 최대 부분 배열을 상수 시간 안에 결정하라.

나의 답

kadane 알고리즘이라고 하는 방식이다.

1. 요소를 하나씩 더한다.
2. 더한 값을 변수에 저장한다.
3. 더한 값이 그 마지막 저장해놓은 변수값보다 크면 변수를 대입한다.

이 알고리즘의 핵심은 Dynamic Programming을 적용한 방식이다.

<u>각각의 최대 부분합은 이전 최대 부분합이 반영된 결과값이다.</u>

이것이 핵심이다.

따라서 주목하는 index를 하나씩 옮길 때마다 둘 중 하나를 고르면 된다.

1. 이전 부분 배열에서 구한 최대 부분합을 유지할 것인가?
2. 이전 부분 배열에서 구한 최대 부분합에 현재 index의 값을 더한 것으로 업데이트할 것인가?

DP는 중복되는 부분 문제를 저장하기 때문에 재귀적으로 중복계산하지 않는다.