KNU KYUNGPOOK NATIONAL UNIVERSITY

Lecture #3: Divide and Conquer

School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeong Nam

점근적 표기법(Asymptotic Notation)

g(n)
f(n)
f(n)이 O(g(n))이라는 것은.
충분히 큰 n에 대해
f(n) ≤ c·g(n)을 만족하는. 상수 c가
존재

- 표기법
  - 빅오 표기법(Big O notation)
    - 알고리즘의 최악의 경우 시간 복잡도를 나타내는 표기법
    - 함수 f(n)이 O(g(n))이라는 표기법은 충분히 큰 n에 대해 f(n) <= c * g(n)을 만족하는 상수 c가 존재한다는 것을 의미
      - 알고리즘의 상한을 나타내는 것으로, 최악의 경우 알고리즘의 수행 시간이 g(n)보다 더 느릴 수 없다는 것을 의미
  - 오메가 표기법(Omega notation)
    - 알고리즘의 최선의 경우 시간 복잡도를 나타내는 표기법
    - 함수 f(n)이 Ω(g(n))이라는 표기법은 충분히 큰 n에 대해 f(n) >= c * g(n)을 만족하는 상수 c가 존재한다는 것을 의미
      - 알고리즘의 하한을 나타내는 것으로, 최선의 경우 알고리즘의 수행 시간이 g(n)보다 더 빠를 수 없다는 것을 의미

c1·g(n) ≤ f(n) ≤ c2·g(n)을
만족하는

  - 세타 표기법(Theta notation)
    - 알고리즘의 평균적인 경우 시간 복잡도를 나타내는 표기법
    - 함수 f(n)이 Θ(g(n))이라는 표기법은 충분히 큰 n에 대해 c1 * g(n) <= f(n) <= c2 * g(n)을 만족하는 상수 c1과 c2가 존재한다는 것을 의미
      - "알고리즘의 상한과 하한이 동일한 경우를 의미합니다. (비례)"

점근적 표기법(Asymptotic Notation)

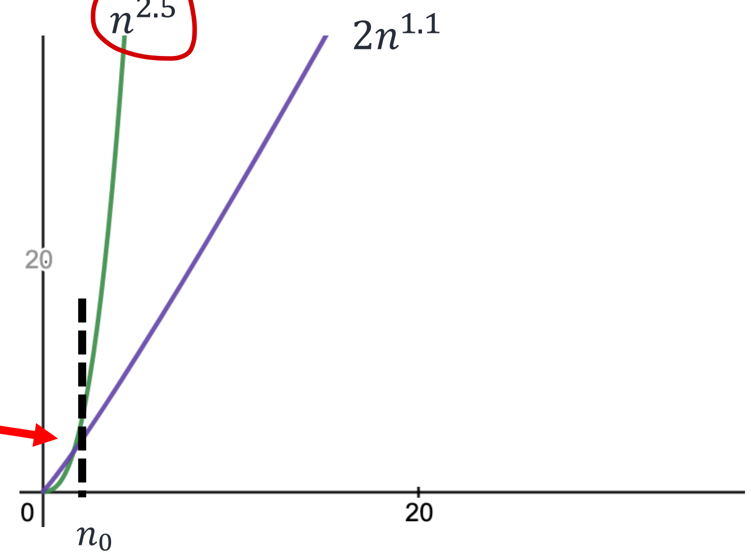
- **Definition**
  - 빅오 표기법(Big O notation) – upper bound

Definition of O-notation

- $O\big(g(n)\big) = \{f(n)$: there exist constants $c > 0, n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

- **Example**
  - $2n^{1.1} \in O(n^{2.5})$
  - **Here,** $c = 1, n_0 = 1.641$

Not interested
(trivial)



점근적 표기법(Asymptotic Notation)

- **Definition**
  - 빅세타 표기법(Big Ω notation) - lower bound

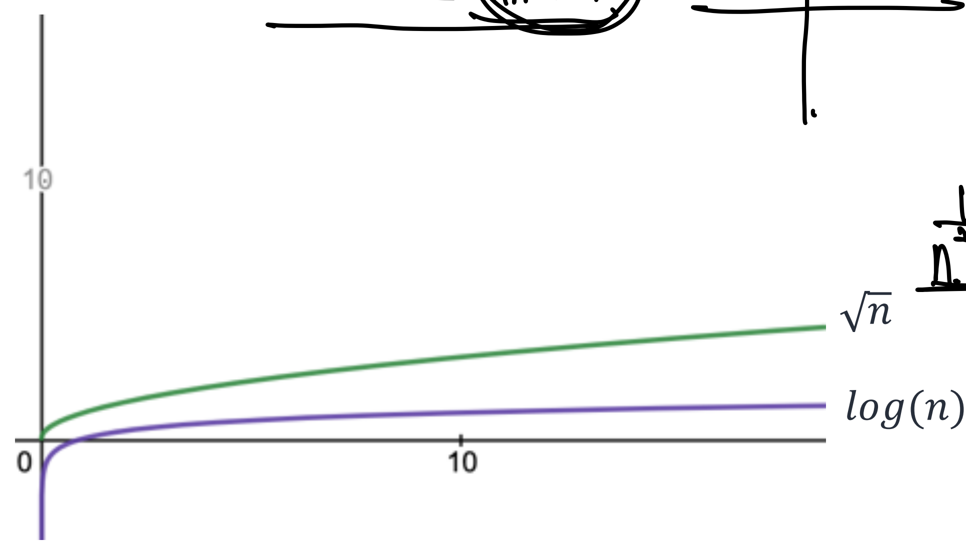
Definition of Ω-notation

- $\Omega(g(n)) = \{f(n)$: there exist constants $c > 0, n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$

$n^2 \le 4n^3+2n^2$

최선

- **Example**
  - $\sqrt{n} \in \Omega(\log(n))$



$n^{\frac{1}{2}}$

점근적 표기법(Asymptotic Notation)

- **Definition**
  - 빅세타 표기법(Big Θ notation) - tight bound

Definition of Ω-notation

- $\Theta(g(n)) = \{f(n)$: there exist constants c_1, c_2, $and\ n_0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

$c_1 g(n) \le f(n) \le c_2 g(n).$

- **Example**
  - $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
    - $c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$
    - $c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$
    - If $c_2 = \frac{1}{2}$?
    - If $c_1 = \frac{1}{14}, n_0 = 7$?

다른 상수도 선택할수 있지만
존재한다는 것이 중요!

점근적 표기법(Asymptotic Notation)

$4n^2$ $\quad$ $4n^3 + 3n^2$

$6n^2 + 9$ $\quad$ $6n^6 + n^4$

$5n^2 + 2n$ $\quad$ $2^n + 4n$

$3\lg n + 8$ $\quad$ $4n^2$

$5n + 7$ $\quad$ $6n^2 + 9$

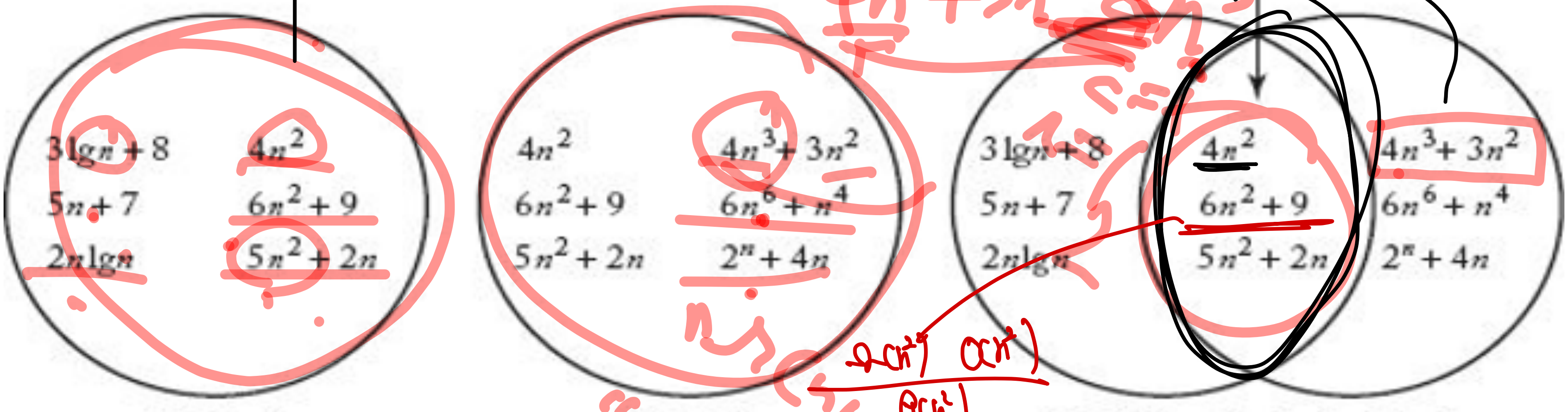
$2n\lg n$ $\quad$ $5n^2 + 2n$

점근적 표기법(Asymptotic Notation)

$4n^3+3n^2$ $\Omega(g(n))$

$4n^3+3n^2 \le c_1 g(n)$

θ(n²)이 될 수 없음
O(n³)이기 때문



(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

$O(n)$

$4n^2$ $\Theta(n^2)$

$6n^2+9$ $O(n^2)$ Ω

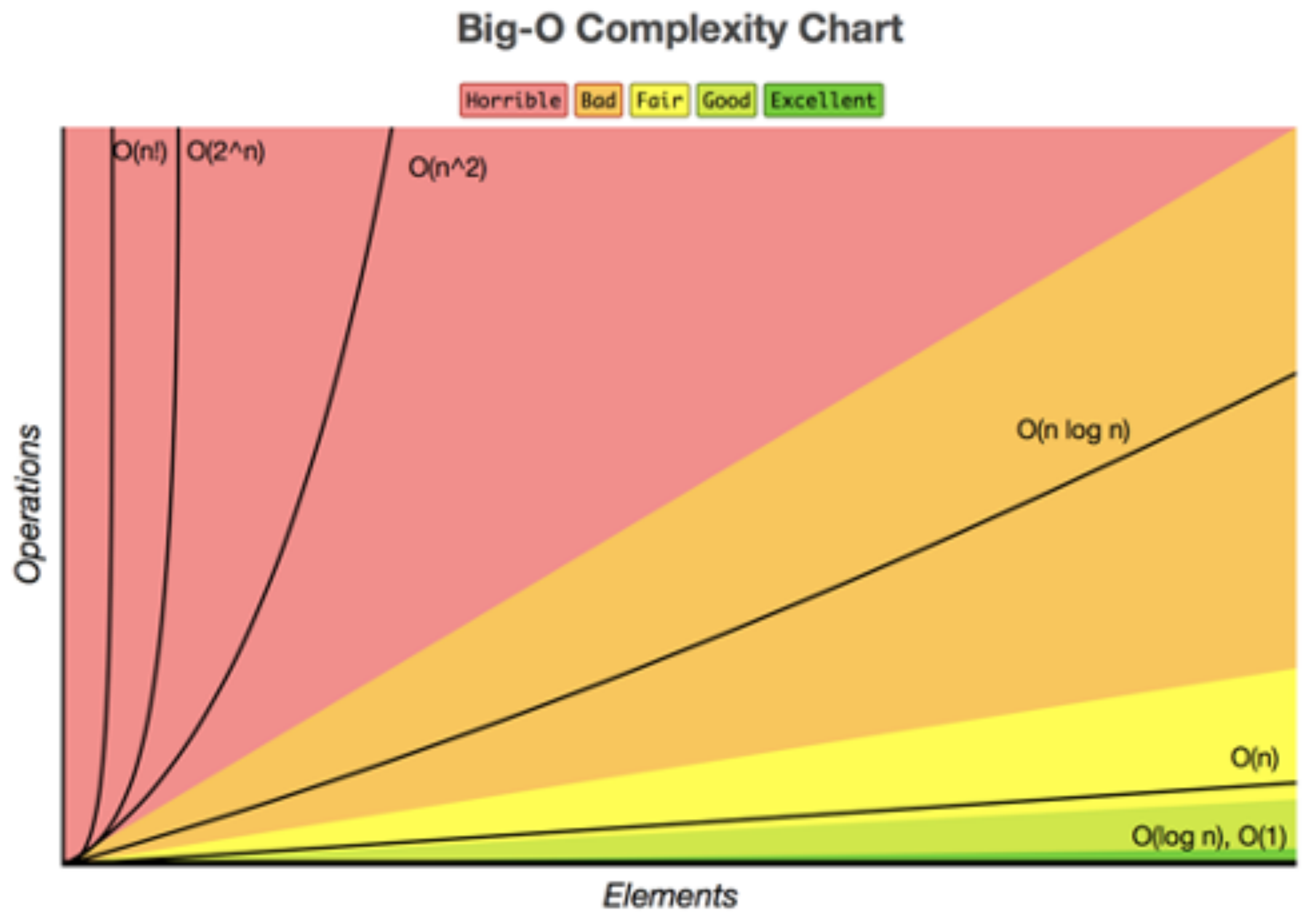
→ 알고리즘의 최악, 최선, 평균 케이스에 대해 O, Ω, Θ를 계산할 수 있다

ex) 퀵 소트의 평균 케이스는 $\Theta(N \log N)$

최악 케이스는 $\Theta(N^2)$

점근적 표기법(Asymptotic Notation)

- **Big-O cheat sheet**
  - https://www.bigocheatsheet.com/



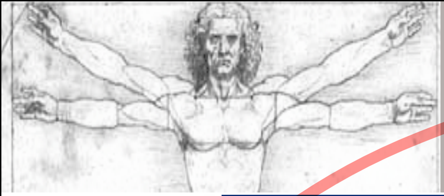
최대부분 배열

- 만약 딱 한번만 미래를 볼 수 있다면…?

최대부분 배열

- 만약 딱 한번만 미래를 볼 수 있다면…?
  - 전 재산 다 팔고 주식에 올인
  - 어떤 주식?
    - 초 고위험 하이리스크 하이리턴

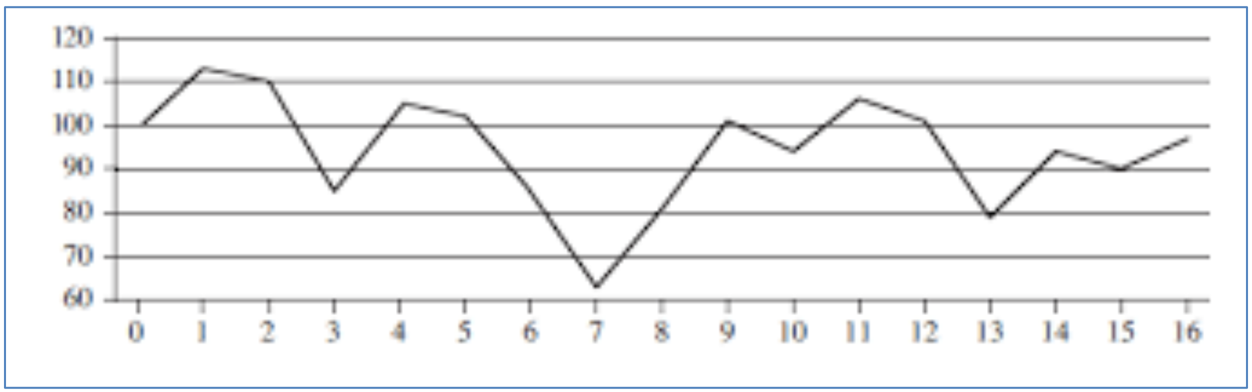
$\sqrt{n}$ $\lg n$



최대부분 배열

- 만약 딱 한번만 미래를 볼 수 있다면...?
  - 전 재산 다 팔고 주식에 올인
  - 어떤 주식?
    - 초 고위험 하이리스크 하이리턴
  - 하지만 딱 한번만 사고 팔 수 있다!
    - 이득을 최대화 해보자

$\binom{n}{2}$



최대부분 배열

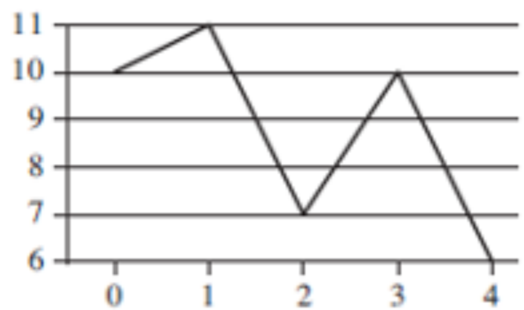
- **Maximum-Subarray Problem**
  - 어떻게 최대부분 배열을 찾을 수 있을까?
- **Brute-force solution**
  - 주먹구구식으로 다 구해보기
  - N일의 기간동안 그런쌍이 $\binom{n}{2}$개존재한다
  - 시간복잡도는 $O(n^2)$

$_nC_2 \rightarrow O(n^2)$

최대부분 배열

- **Maximum-Subarray Problem**
  - 어떻게 최대부분 배열을 찾을 수 있을까?
- **Brute-force solution**
  - 주먹구구식으로 다 구해보기
  - N일의 기간동안 $\binom{n}{2}$개존재한다
  - 시간복잡도는 $O(n^2)$
- 가격말고 가격 간 차이로 변환

$\binom{n}{2} = O(n^2)$



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

→ 차이로 변경

- **Transformation**
  - Let's consider the daily change price
  - *A*[*i*] = (price after day *i*) - (price after day (*i* - 1))
  - Assume that we start with a price after day 0, i.e., just before day 1
  - 배열을 변환하더라도 주먹구구식으로 해결하면 여전히 $O(n^2)$

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

by dp

직전에서의 최대 부분 배열

dp[i-1]

dp[i] = max(dp[i-1] + A[i], 0)

→ 최댓값이 [illegible]

return O(n)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

13

0

최대부분 배열

- **Divide and Conquer**
  - Input: array A[1..n] of numbers
  - Output: Indices i and j such that A[1..n] has the greatest sum of any nonempty, contiguous subarray of A, along with the sum of the values in A[i..j]
  - 즉 합이 최대인 연속된 구간을 찾는게 목적

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

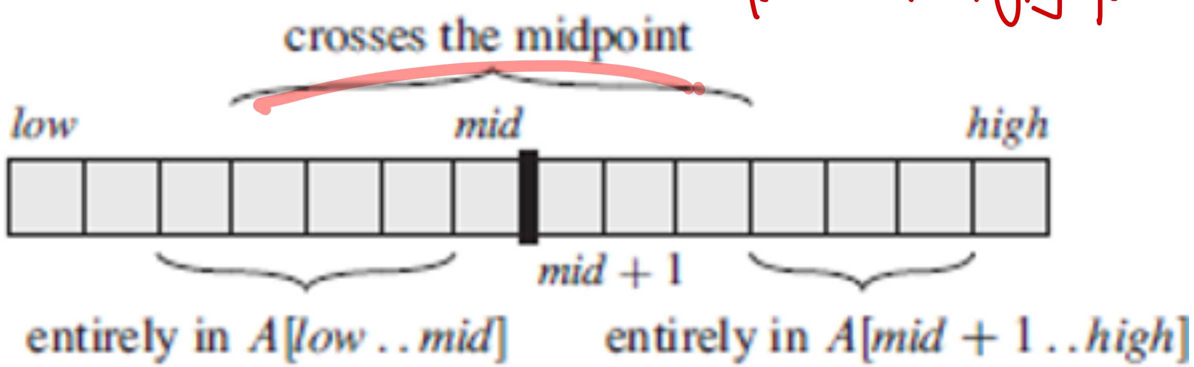
maximum subarray

최대부분 배열 – 분할 정복

- Output에 해당하는 배열이 존재하는 구간에 대해 생각해보자
  - Low, high: 배열의 처음과 끝 인덱스
- 존재 가능한 구간은?
  - entirely in the subarray A[low..mid], so that low ≤ i ≤ j ≤ mid,
  - entirely in the subarray A[mid + 1..high], so that mid < i ≤ j ≤ high, or
  - crossing the midpoint, so that low ≤ i ≤ mid < j ≤ high.

  - 반드시 셋 중 하나에는 존재하게 된다

(A[low... mid]
A[mid+1 ... high]).

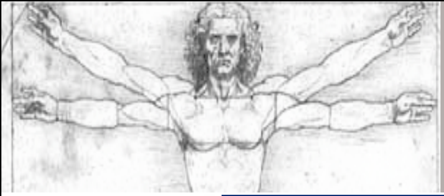


최대부분 배열 – 분할 정복

- 배열을 분할해도 조건은 똑같다
  - 재귀적으로 배열을 분할해서 그 분할된 배열에서 최대를 찾는다
    - 반드시 탈출조건을 지정
    - 만약 한 개의 값이 존재할 시 return
  - 만약 중간에 걸쳐있는 경우?
    - Mid부터 시작해서 low쪽으로 가면서 최대부분배열을 찾고
    - Mid+1 부터 high까지 가면서 최대 부분배열을 찾음
    - 그리고 합친다
  - 세가지 경우의 수중에 최대값을 return하면 찾을 수 있음!

최대부분 배열 – 분할 정복

- 대략적인 코드를 살펴보자



최대부분 배열 – 분할 정복

- 대략적인 코드를 살펴보자

최대부분 배열 – 분할 정복

- **Pseudo code – 가운데 걸쳐있는 부분 찾기**
  - 가운데 걸쳐있는 subarray의 최대부분 배열을 찾는 과정
  - 보면 for문이 한번만 있음으로 시간복잡도가 $O(n)$임을 알 수 있다

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    left-sum = -∞
    sum = 0
    for i = mid downto low
        sum = sum + A[i]
        if sum > left-sum
            left-sum = sum
            max-left = i
    right-sum = -∞
    sum = 0
    for j = mid + 1 to high
        sum = sum + A[j]
        if sum > right-sum
            right-sum = sum
            max-right = j
    return (max-left, max-right, left-sum + right-sum)
```

갱신

i

j

max-left

max-right

합 = left-sum + right-sum

최대부분 배열 – 분할 정복

- **Pseudo code – Recursive 하게 전체에서 찾는 방법**
  - 재귀 함수의 탈출 조건: high==low
    - 배열 내 하나의 원소만 존재 시 탈출
  - 세가지 경우의 수 중 합이 최대인 배열을 return

재귀로 해결

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (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)
    else return (cross-low, cross-high, cross-sum)
```

→ 기저 사례!

→ 재귀 호출.

값 받으로 가져온다

left-sum

right-sum

cross-sum

최대를 return

최대부분 배열 – 분할 정복

- **Recursive function**
  - 함수 내부에서 자기 자신을 호출하는 함수
    - 마치 인셉션처럼 꿈속에 꿈으로 들어간다
  - 종료 조건이 없다면?
    - 영원히 꿈속에 갇힌다
  - 재귀함수의 일반적인 종료조건을 정해놓고 만족할때까지 함수를 반복적으로 호출하여 문제를 해결
  - 코드의 가독성이 높아지고 간결해진다
  - 피보나치의 함수는?

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

최대부분 배열 – 분할 정복

- **Time complexity**

$2T(\frac{n}{2}) + \theta(n)$

T(n)

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
    if high == low
        return (low, high, A[low])            // base case: only one element
    else mid = ⌊(low + high)/2⌋
        (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)
        else return (cross-low, cross-high, cross-sum)
```

θ(1)	$T(1) = \Theta(1)$
T(n/2)	$T(n/2)$
T(n/2)	$T(n/2)$
θ(n)	$\Theta(n)$
θ(1)	$T(1) = \Theta(1)$

최대부분 배열 – 분할 정복

- 배열의 개수가 1일때: 탈출!
  - T(1) = Θ(1)
- 재귀함수 일 경우
  - T(n) = Θ(1) + 2*T*(*n*/2) + Θ(*n*) + Θ(1)
    = 2*T*(*n*/2) + Θ(*n*)
- 두가지 케이스를 합치면 다음의 점화식으로 표현된다

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

- 점화식을 풀면 시간복잡도는
  - *T*(*n*) = Θ(*n* lg *n*)
  - 어떻게 풂?

how?

$T(n) = \theta(1) + 2T(n/2) + \theta(n) + \theta(1)$

$= 2T(n/2) + \boxed{\theta(n)}$

$T(n) = \theta(1)$

$+ 2T(n/2) + \theta(n) + \theta(1)$

$= 2T(n/2) + \theta(n)$

$T(n) = \begin{cases} \theta(1) \\ 2T(n/2) + \theta(n) \end{cases}$

$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T(n/2)+\theta(n) & \text{if } n>1 \end{cases}$

Solving Recurrence

- 점화식(Recurrence) 풀기
  - 치환법(Substitution)
  - 재귀트리 방법(recursion-tree)
  - 마스터 방법(Master)

치환

재귀트리

마스터

↳ 균등분으로 나눠질 때만 성립

Solving Recurrence- Substitution

- **Substitution method**
  - 점화식을 추측한 다음 추측한 해를 이용하여 만족하는지 검증
  - 불행하게도... 정확한 해를 추측하는 방법은 없다
    - 말 그대로 감이다
  - 1. 해의 모양을 추측
  - 2. 상수들의 값을 찾아내기 위해 수학적 귀납법 이용, 해가 제대로 동작함을 보임
- 예시

$$T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ 2T(n/2) + n & \text{if } n > 1\,. \end{cases}$$

$T(n) \leq cn\log n.$

  - $T(n) = O(nlogn)$ 이라고 추측
  - 적당한 상수 c에 대해서 $T(n) \leq cnlogn$이 됨을 만족해야 된다

$T(n) = O(n\log n)$

$\exists c \in \mathbb{N}$

$T(n) \leq cn\log n$

$T(n) = O(n\log n)$ 이라 추측

$\exists c$ "$T(n) \leq cn\log n$" → "그 밖은 trivial"

Solving Recurrence- Substitution

- 수학적 귀납법으로 증명
  - 우선 대입해보자
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$
  - $\le 2c\left(\frac{n}{2} log \frac{2}{n}\right) + n$
  - $= cn\left(log\frac{2}{n}\right) + n$
  - $= cnlogn - cnlog2 + n$
  - $= cnlogn - cn + n$
  - $\le cnlogn$
- 수학적 귀납법을 하기 위해 한계조건을 만족하는지 보자
  - $n = 1$ 일 때 $T(1) = 1$인가?
  - $T(1) \le c1log1 = 0$ 이다, 모순이다

$2c\left(\frac{n}{2}log\frac{n}{2}\right) + n$

$cnlog\frac{n}{2} + n$

$cnlogn + n - cnlog2$

$\le cnlogn$

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

$\le 2c\left(\frac{n}{2}log\frac{n}{2}\right) + n$

$= cn\left(log\frac{2}{n}\right) + n.$

$= cnlog n - cnlog2 + n.$

$\le cnlogn - cn + n.$

$\le cnlog n.$

"$T(1) \le 0$"

$T(1) \le 0$

$n=1$ 일때 "$T(1) \le 0$"

trivial!

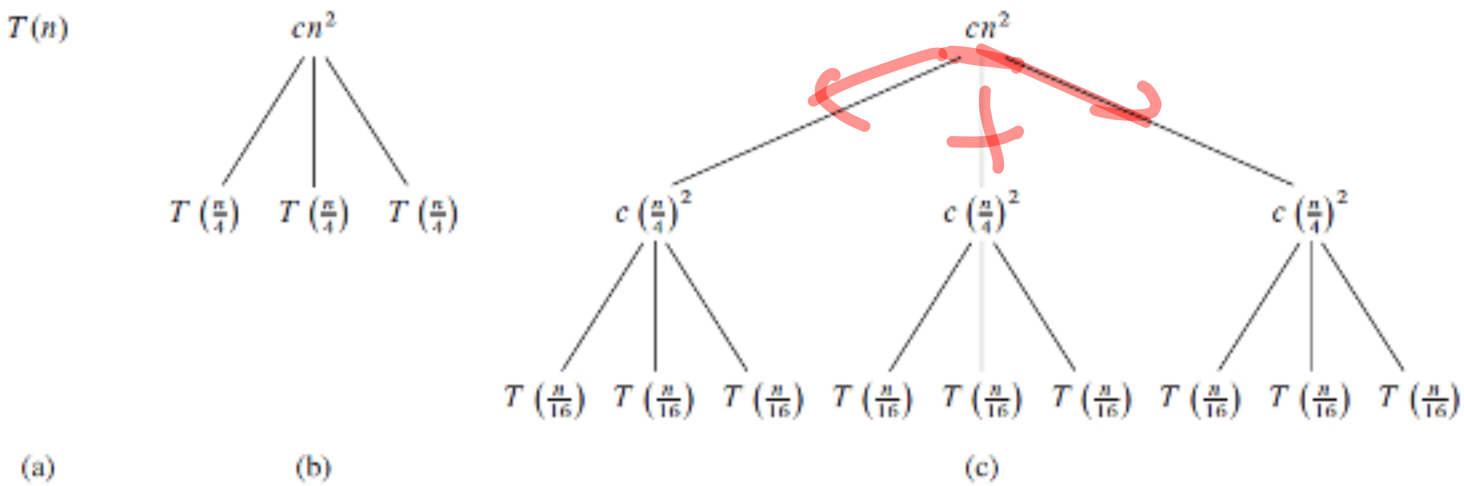
Solving Recurrence- Substitution

- 수학적 귀납법으로 증명
  - $n = 2$ 부터는? 만족한다
- 점근적 정의의 이점을 활용
  - $n \ge n_0$에 대해 $\mathrm{T}(n) \le cnlogn$을 증명하기만 하면 된다
  - 작은 n에 대해 귀납적 가정이 만족되도록 한계조건을 확장 가능
  - 항상 명시하지 않아도 된다
- 좋은 추측식 만들기
  - $T(n) = 2T\left(\frac{n}{2} + 17\right) + n$같은 경우는?
  - 뭔가 17이 추가되서 더 어려운것 같지만 직관적으로 보면 아무런 영향을 미치지 않는다

$2T(n/2)+n \le$ 가정

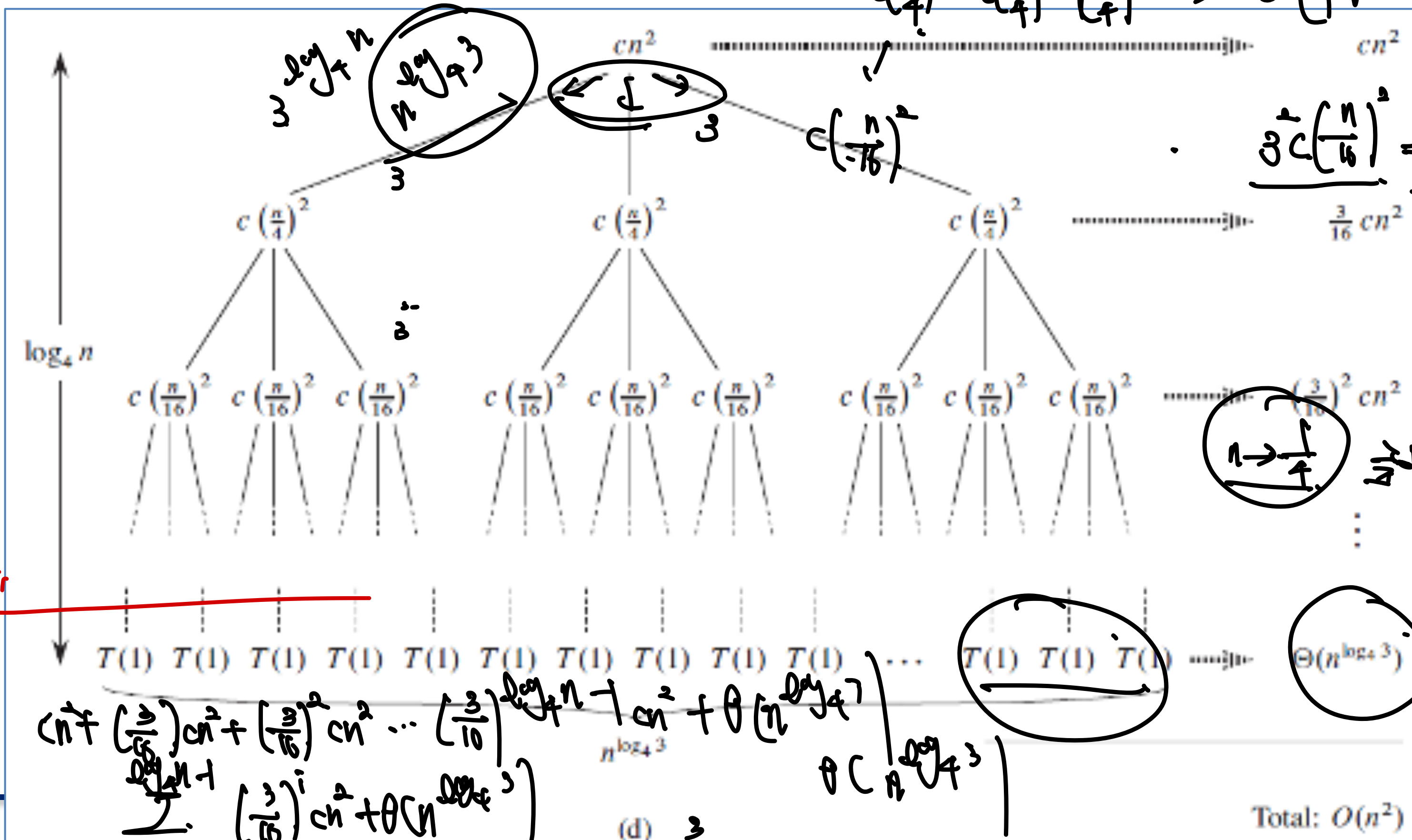
Solving Recurrence- Recursion tree

- 재귀트리 활용하기
  - 병합정렬 때처럼 부분 문제들에 대한 연산 수행 횟수를 모두 합치면 된다
- 예시) $T(n) = 3T(n/4) + cn^2$



Solving Recurrence- Recursion tree

- 재귀트리 활용하기
  - ➢ 병합정렬 때처럼 부분 문제들에 대한 연산 수행 횟수를 모두 합치면 된다
- 예시) $T(n) = 3T(n/4) + cn^2$



$cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \cdots \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$

$\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$

Solving Recurrence- Recursion tree

- **Add up the costs over all levels to determine the cost for the entire tree:**
  - 무한 등비급수를 활용하자

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \quad \text{(by equation (A.5))} .
\end{aligned}$$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}$$

Geometric series

For real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

is a *geometric* or *exponential series* and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} . \qquad \text{(A.5)}$$

When the summation is infinite and $|x| < 1$, we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} . \qquad \text{(A.6)}$$

$cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} + \Theta(n^{\log_4 3})$

$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$

$= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1}$

$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$

$\frac{1}{1 - \frac{3}{16}}$

$\frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$

$\frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\frac{3}{16} - 1} cn^2 + \Theta(n^{\log_4 3})$

$=$

$= O(n^2)$

Solving Recurrence- Master Method

- 마스터 정리
  - 점화식 (재귀 관계식)으로 표현한 알고리즘의 동작 시간을 점근적으로 계산하여 간단하게 나타내는 방법
  - 즉, 여러 경우의 점화식에 대해서, 최종적으로 닫힌 형태의 표현식을 도출 할 수 있는 일종의 공식
  - 모든 점화식을 풀 수 있는 것은 아님.
- 증명: “일반화 된” 트리 기법을 통해서 증명.

(증명 원리를 잘 외우자)

Solving Recurrence- Master Method

- 다음과 같은 형태의 점화식을 푸는 기본 기침

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

where $a \geq 1$, $b > 1$, and $f(n) > 0$.

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

$a \geq 1$, $b \geq 1$.

$f(n) > 0$

$a \geq 1$

$b > 1$

> T(n)에 대한 점근적 한계는 다음과 같다

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Solving Recurrence- Master Method

가지 시에

- 좀더 쉽게 $\mathbf{f}(\boldsymbol{n})$을 $\mathbf{O}(\boldsymbol{n}^d)$로 쓰면
  - $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$

$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$.

부분 문제들의 개수

↳ 입력 크기 감소에 대한 인자.

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

$n^{\log_b a}$

세 가지 주요 매개변수(파라미터)들:

a : 부분 문제들(subproblems)의 개수

b : 입력 크기 감소에 대한 인자 (factor)

d : 모든 부분 문제들을 생성하고 부분 문제들의 답을 조합하기 위해 n^d 횟수 만큼 연산을 수행해야 할 때, n의 지수

$O(n)$

Solving Recurrence- Master Method

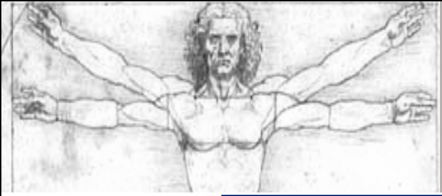
- 예제)

$5 > 2^2$ $O(n^{\log_b a}) = O(n^{\log_2 5})$

- $T(n) = 5T(n/2) + \Theta(n^2)$
  $n^{\log_2 5}$ vs. n^2
  Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use Case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$

- $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$ $\quad 2n < 3$ $\quad n^a$
  $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
  Use Case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$

- $T(n) = 5T(n/2) + \Theta(n^3)$ $\quad 5 < 2^3$ $\quad n^3$
  $n^{\log_2 5}$ vs. n^3
  Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
  Check regularity condition (don't really need to since $f(n)$ is a polynomial):
  $af(n/b) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$
  Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$



Solving Recurrence- trade off relationship

최저 사례.

문제를 나누는 행위 자체가 문제를 더 많이 (복잡하게) 만든다!
어차피 결국 트리 마지막 단계에서 기존보다 더 많은 일을 수행하게 된다

트리의 아래 단계로 갈수록 문제는 더 작아지고 작아(단순해)진다!
초기 문제의 규모가 가장 크고 또 복잡한 형태다

Solving Recurrence- trade off relationship

- 예시) $T(n) = T\left(\frac{n}{2}\right) + n$ $\left(a < b^d\right)$ → 거저 사례의 적용

  최상단

- 수행해야 할 일의 상당수는 트리의 최상단에서 이루어진다.
  - 그리고, 다른 부분 문제들의 합보다 규모가 크다.

- T(n) = O(work at top) = O(n)

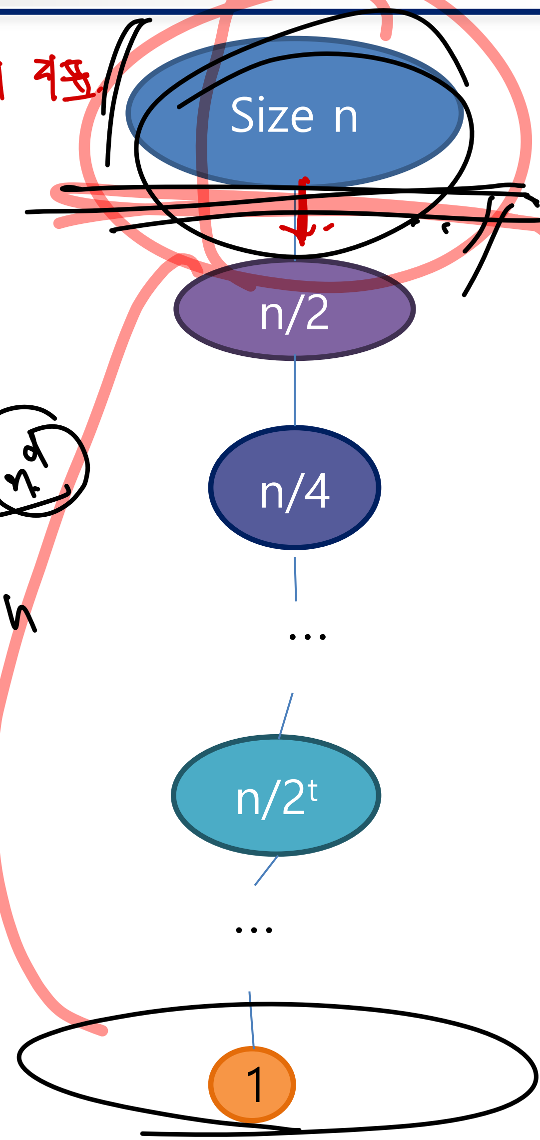
$a < b^d$

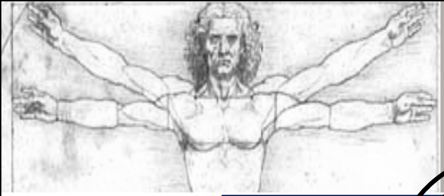
최상단

n^d

$= n$

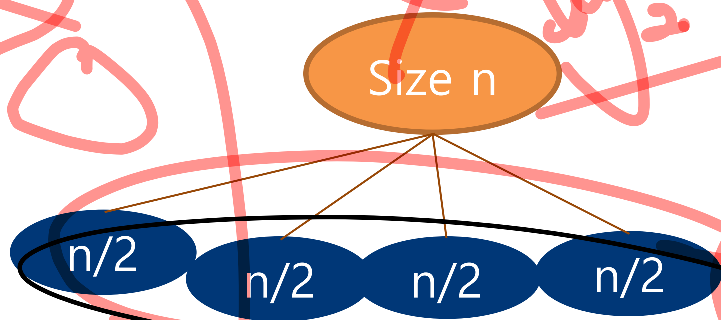
Most work at the top of the tree!





Solving Recurrence- trade off relationship

- 예시) $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \quad (a > b^d)$
- 단말 노드 (leaf node)가 많은 형태의 트리구조를 띈다.
  - 전체 작업 (연산량)의 규모로 봤을 때, 단말 노드에서 이루어지는 연산이 대부분
- $T(n) = O(\text{ work at bottom }) = O(4^{\text{depth of tree}}) = O(n^2)$



Most work at the bottom of the tree!

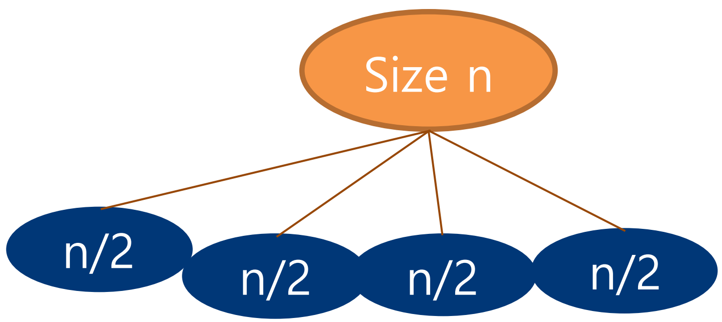
- 여러 가지 방법들 중, 문제를 푸는 상황에서 타당한 방법을 선택

Solving Recurrence- trade off relationship

- 예시) $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad \left(a > b^{d}\right)$
- 단말 노드 (leaf node)가 많은 형태의 트리구조를 띈다.
  - ➢ 전체 작업 (연산량)의 규모로 봤을 때, 단말 노드에서 이루어지는 연산이 대부분

- T(n) = O(work at bottom) = O($4^{\text{depth of tree}}$) = O(n^2)

$4^{d^{a}}, n.$



Most work at the bottom of the tree!

- 여러 가지 방법들 중, 문제를 푸는 상황에서 타당한 방법을 선택