Master Theorem

마스터 방법

INTRODUCTION TO ALGORITHMS (Thomas H. Cormen et al.)

KPSC Algorithm Study 24/09/05 Thu.

by Haru_101

Loop Invariant

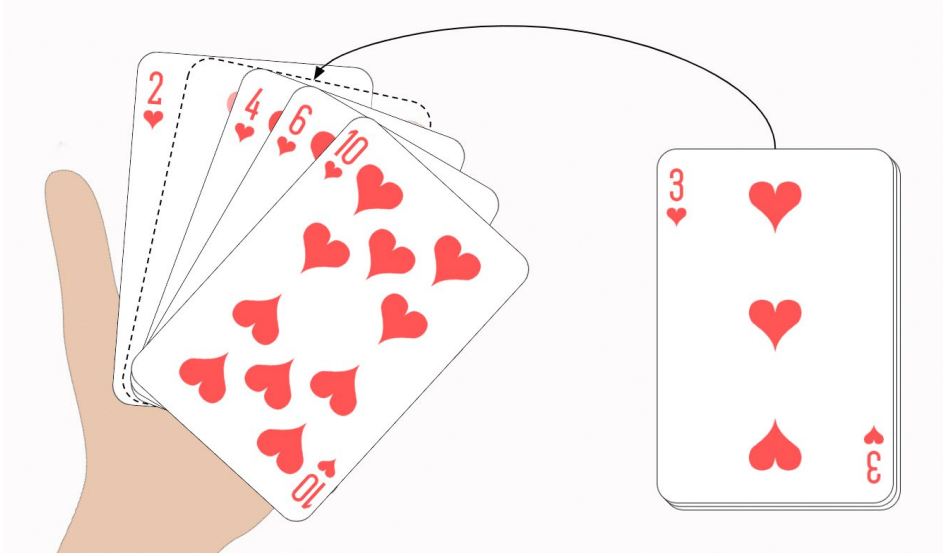
- 루프 불변성(loop invariant)이란, 알고리즘이 타당한 이유를 쉽게 이해할 수 있도록 하기 위해 사용됨.
- 초기화 : 루프가 첫 번째 반복을 시작하기 전에 참이다.
- 유지 : 루프의 반복이 시작되기 전에 참이면 다음 반복이 시작되기 전에도 유지된다.
- 종료 : 루프가 종료될 때 그 불변식이 알고리즘의 타당성을 보이는 데 유용한 특성을 가져야 한다.

Loop Invariant

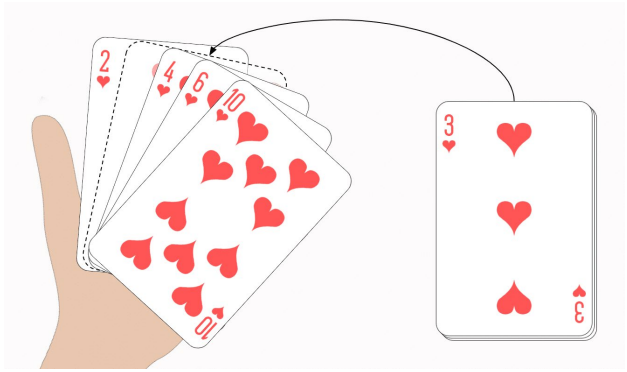
- 삽입 정렬(insertion sort)를 예시로 살펴보자.

- 동작 방식
  - 왼손에 현재 있는 카드들은 오름차순으로 정렬되어 있음.
  - 오른손은 카드 더미에서 한장을 뽑음.

  - 오른손에 있는 카드를 왼손에 있는 카드의 가장 오른쪽부터 보면서 적절한 위치에 삽입



Loop Invariant

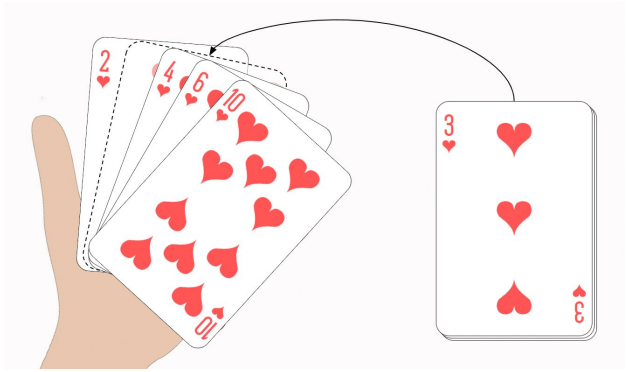


- 정렬할 카드들의 정보는 $A[1:n]$으로 나타낼 수 있음.
  - n은 정렬할 카드 수

- 첫 번째 카드를 오른손으로 뽑았을 때에는 왼손에 아무런 카드가 없으므로 왼손으로 옮기면 정렬 끝
  - $A[1:1] = \mathrm{A}[1]$은 항상 정렬되어 있음

- 이제 $i = 2$부터 n까지 카드를 뽑고 왼손에 적절한 위치에 카드를 넣으면 됨
  - 이때, $A[1:i-1]$까지는 정렬이 되어 있음. ($i-1$에서 잘 정렬 했으니)
  - 이걸 불변식이라고 함.

Loop Invariant

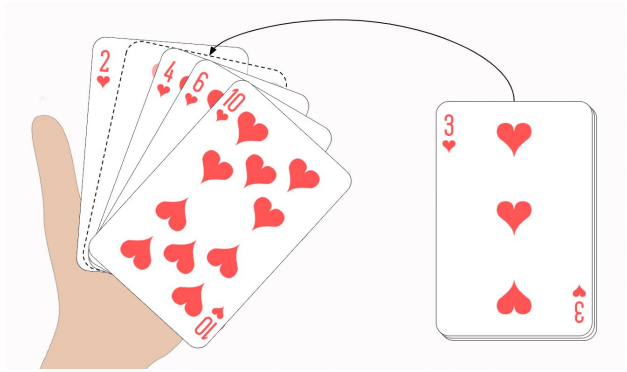


```
Insertion-Sort(A, n)

1       for i = 2 to n
2               key = A[i]
3               j = i-1
4               while j > 0 and A[j] > key
5                       A[j+1] = A[j]
6                       j = j-1
7               A[j+1] = key
```

$A[1: i-1]$는 정렬되어 있는가?

Loop Invariant



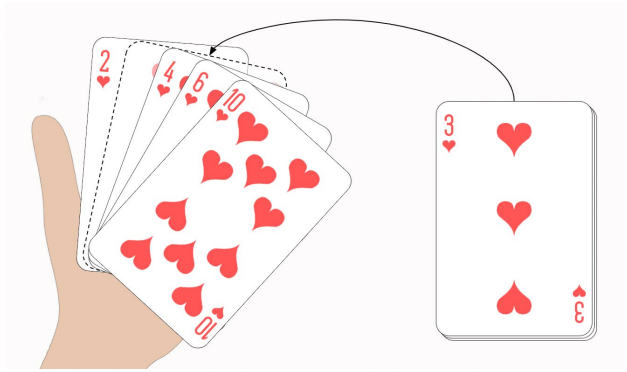
```
Insertion-Sort(A, n)

1       for i = 2 to n
2               key = A[i]
3               j = i-1
4               while j > 0 and A[j] > key
5                       A[j+1] = A[j]
6                       j = j-1
7               A[j+1] = key
```

초기화 루프가 첫 번째 반복을 시작하기 전에 참이다.

$i = 2$일때 $A[1: i-1] = A[1]$은 정렬되어 있음
([x]를 정렬하면 [x]이므로)

Loop Invariant



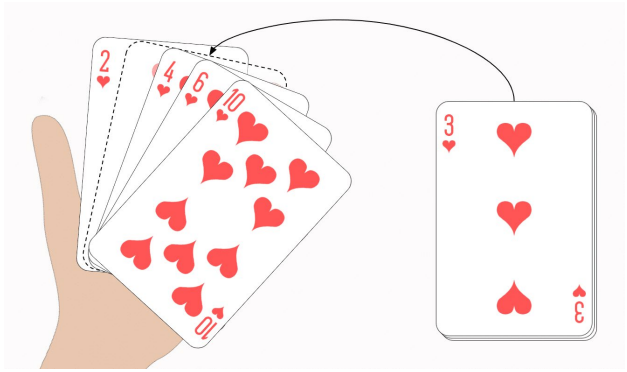
```
Insertion-Sort(A, n)

1       for i = 2 to n
2           key = A[i]
3           j = i-1
4           while j > 0 and A[j] > key
5                   A[j+1] = A[j]
6                   j = j-1
7           A[j+1] = key
```

유지 루프의 반복이 시작되기 전에 참이면 다음 반복이 시작되기 전에도 참이 유지된다

$i = k$일때 반복을 시작하기 전 $A[1:i-1] = A[1:k-1]$은 정렬되어 있다 (참).
그렇다면 $i = k+1$을 시작하기 전에도 $A[1:i-1] = A[1:k]$이 정렬되어 있는가?
$A[1:k-1]$에 $A[k]$를 적절히 삽입하면 $A[1:k]$는 정렬된다. (참)

Loop Invariant



```
Insertion-Sort(A, n)

1       for i = 2 to n
2               key = A[i]
3               j = i-1
4               while j > 0 and A[j] > key
5                       A[j+1] = A[j]
6                       j = j-1
7               A[j+1] = key
```

종료　　루프가 종료될 때 그 불변식이 알고리즘의 타당성을 보이는 데 유용한 특성을 가져야 한다.

$i = n + 1$이면 루프가 끝난다. 그렇다면 $A[1:\mathrm{i}-1]$은 정렬되는가?
$i = n + 1$이면 $A[1:\mathrm{i}-1] = \mathrm{A}[1:\mathrm{n}]$, 즉 n개의 원소가 정렬되어 있다.

Divide And Conquer

- 재귀적 구조를 가진 여러 유용한 알고리즘이 많은데, 이런 알고리즘들은 전형적으로 분할 정복 접근법을 따름.
- 전체 문제를 크기가 작은 여러 개의 문제로 분할하고, 작아진 문제를 재귀적으로 풂
- 문제가 충분히 작은 경우 재귀 없이 문제를 해결할 수 있음 (ex. [-1]을 정렬한 결과는?)
- 그렇지 않으면(재귀 케이스)에는 다음 세 가지 단계를 거치면서 재귀적으로 문제를 해결함

Divide And Conquer

- 분할(divide) : 현재의 문제를 같은 문제를 다루는 다수의 부분 문제로 분할한다.
  - 현재 문제 : 크기가 n인 배열을 정렬해야한다.
  - 부분 문제 : 크기가 n인 배열을 크기가 각각 m, k인 배열로 나눈다. $(m + k = n)$

- 정복(conquer) : 부분 문제를 재귀적으로 풀어서 정복한다. 부분 문제의 크기가 충분히 작으면 직접적인 방법으로 푼다.
  - 부분 문제 : 크기가 $m(or\ k)$인 배열을 정렬한다.

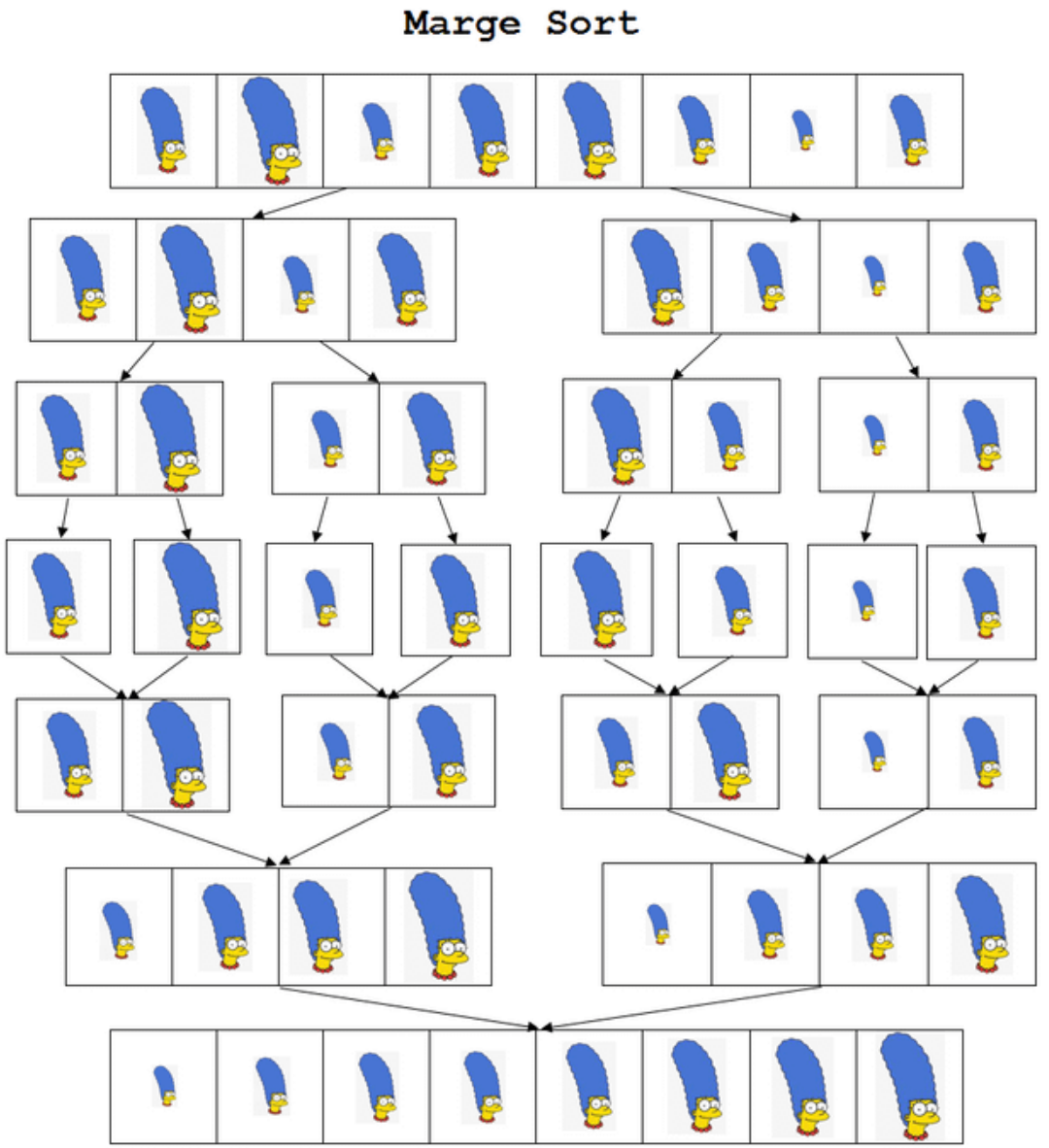
- 결합(combine) : 부분 문제의 해를 결합하여 원래 문제의 해가 되도록 만든다.
  - 정렬된 크기가 m, k 인 배열을 합쳐 정렬된 크기가 n인 배열로 만든다.

Divide And Conquer

- 만약 현재 배열의 크기가 1이면 더 쪼갤 일이 있을까?
  - 당연히 없음. [1], [101], [2147483647]은 이미 정렬된 상태.
  - 전에 삽입 정렬의 루프 불변식을 다룰때 보았음.
  - 이때는 재귀 호출이 “하한에 이른다” 라고 표현함.

- 병합 정렬(merge sort)에 대해 분할-정복-결합 구조로 풀어써보자.

Merge Sort



Merge Sort

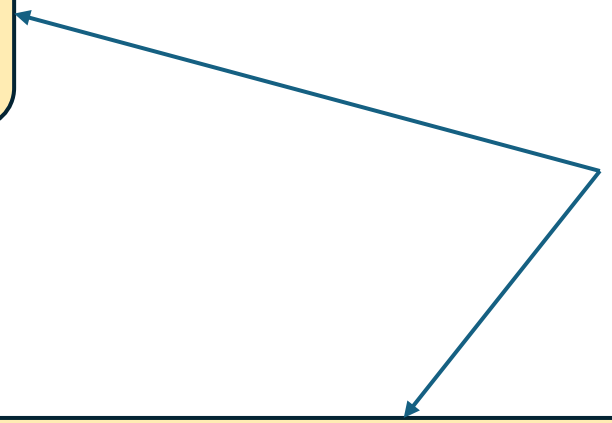
- 분할(divide) : 현재의 문제를 같은 문제를 다루는 다수의 부분 문제로 분할한다.
  - 현재 문제 : 크기가 n인 배열을 정렬해야한다. -> $A[p:r]$ $((r-p+1) = n)$
  - 부분 문제 : 크기가 n인 배열을 $A[p:q], A[q+1:r]$로 나눈다. $q = \left\lfloor \frac{p+r}{2} \right\rfloor$

- 정복(conquer) : 부분 문제를 재귀적으로 풀어서 정복한다. 부분 문제의 크기가 충분히 작으면 직접적인 방법으로 푼다.
  - 부분 문제 : $A[p:q], A[q+1:r]$를 정렬한다.

- 결합(combine) : 부분 문제의 해를 결합하여 원래 문제의 해가 되도록 만든다.
  - 정렬된 $A[p:q], A[q+1:r]$를 이용해서 $A[p:r]$로 만든다.

Merge Sort

```
Merge(A, p, q, r)
1       nL = q – p + 1 // size of A[p:q]
2       nR = r – q // size of A[q+1:r]
3       L = [0:nL-1], R = [0:nR-1]
4       for i = 0 to nL-1
5               L[i] = A[p + i]
6       for i = 0 to nR-1
7               R[i] = A[q+1 + i]
8       i = 0, j = 0
9       k = p // A[p:q]를 정렬할 때 사용할 인덱스 k = p, p+1, ..., q-1, q
```



상수 시간이 소요됨

Merge Sort

```
Merge(A, p, q, r)
1       nL = q – p + 1 // size of A[p:q]
2       nR = r – q // size of A[q+1:r]
3       L = [0:nL-1], R = [0:nR-1]
4       for i = 0 to nL-1
5               L[i] = A[p + i]
6       for i = 0 to nR-1
7               R[i] = A[q+1 + i]
8       i = 0, j = 0
9       k = p // A[p:q]를 정렬할 때 사용할 인덱스 k = p, p+1, ..., q-1, q
```

$\Theta(nL + nR) = \Theta(n)$

Merge Sort

```
10      while i < nL and j < nR
11              if L[i] <= R[j]
12                      A[k] = L[i]
13                      i = i + 1
14              else
15                      A[k] = R[j]
16                      j = j + 1
17              k = k + 1
18      while i < nL
19              A[k] = L[i]
20              i = i + 1
21              k = k + 1
```

```
22      while j < nR
23              A[k] = R[j]
24              j = j + 1
25              k = k + 1
```

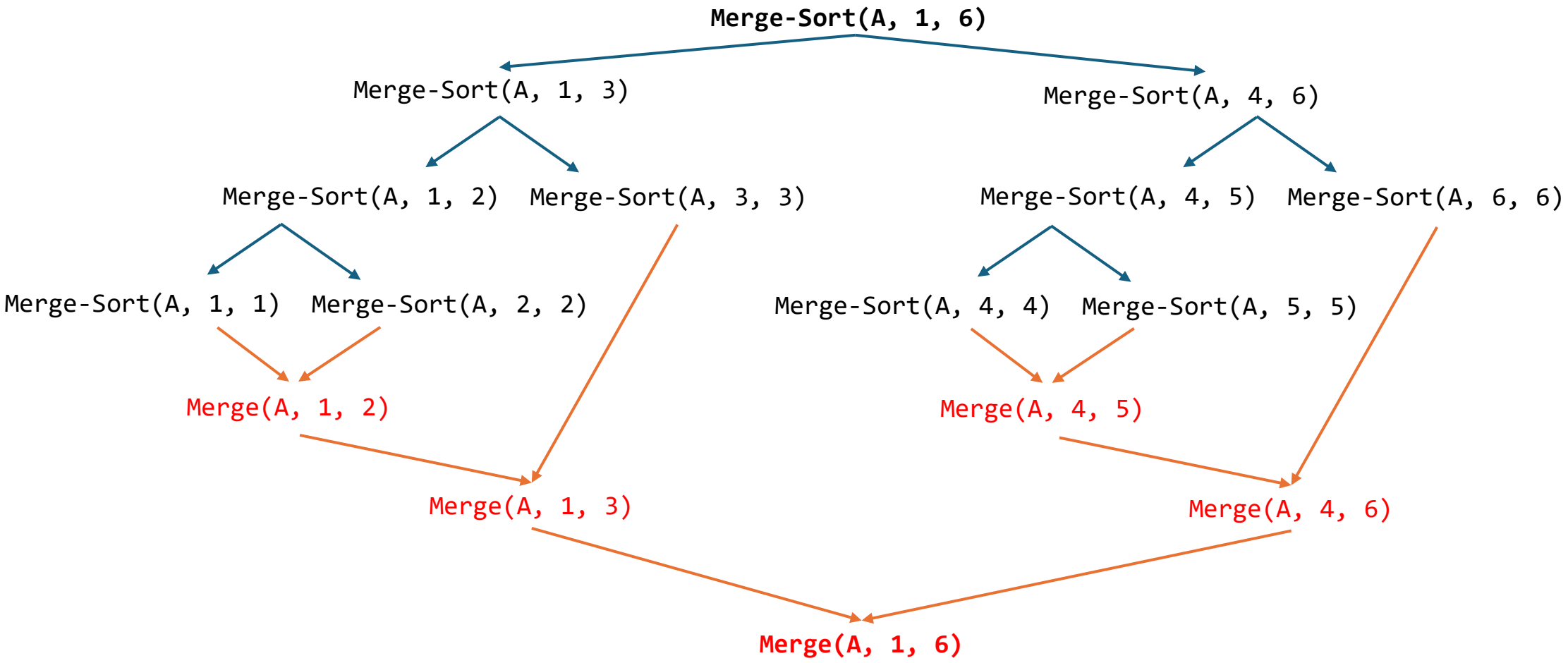
$$\Theta(nL + nR) = \Theta(n)$$

Merge Sort

```
Merge-Sort(A, p, r)
1       if p >= r // p = r이면 A[p:r] = A[p] 는 정렬된 상태. 따라서 하한
2               return
3       q = (p + r) / 2 // 소숫점 내림 3.5 -> 3
4       Merge-Sort(A, p, q) // A[p:q] 재귀적으로 정렬
5       Merge-Sort(A, q+1, r) // A[q+1:r] 재귀적으로 정렬
6       Merge(A, p, q, r) // A[p:q]와 A[q+1:r]을 합쳐 A[p:r]로 만듦
```

Merge-Sort(A, 1, 6)을 생각해보자.

Merge Sort


Merge-Sort(A, 1, 6)
Merge-Sort(A, 1, 3)
Merge-Sort(A, 4, 6)
Merge-Sort(A, 1, 2)
Merge-Sort(A, 3, 3)
Merge-Sort(A, 4, 5)
Merge-Sort(A, 6, 6)
Merge-Sort(A, 1, 1)
Merge-Sort(A, 2, 2)
Merge-Sort(A, 4, 4)
Merge-Sort(A, 5, 5)
Merge(A, 1, 2)
Merge(A, 4, 5)
Merge(A, 1, 3)
Merge(A, 4, 6)
Merge(A, 1, 6)

Recurrence

- 점화식(recurrence)는 더 작은 입력에 대한 자신의 식으로 함수를 나타내는 방정식 또는 부등식

- $T(n)$을 크기 n짜리 입력에 대한 수행 시간이라고 정의하자.
  - 문제 크기가 충분히 작은 경우 $(n < n_0, \quad n_0 > 0)$ $\Theta(n) = \Theta(1)$

  - 주어진 문제가 원래 문제의 $\frac{1}{b}$인 a개의 부분 문제로 분할되었다고 할때, a개의 부분 문제를 모두 해결하는 데에 $aT(\frac{n}{b})$의 시간이 걸리게 됨.

  - 병합 정렬에선 $a = b = 2$

  - 문제를 분할하는 데 $D(n)$ 시간이 걸리고 부분 문제를 결합하는 데에 $\mathrm{C}(n)$

Recurrence

- 문제를 분할하는 데 $D(n)$ 시간이 걸리고 부분 문제를 결합하는 데에 $C(n)$ 시간이 걸린다고 하자.

- $T(n)$을 다음과 같이 표현할 수 있음.

$$T(n) = \begin{cases} \Theta(1) & if \quad n < n_0 \\ D(n) + aT\left(\frac{n}{b}\right) + C(n) & otherwise \end{cases}$$

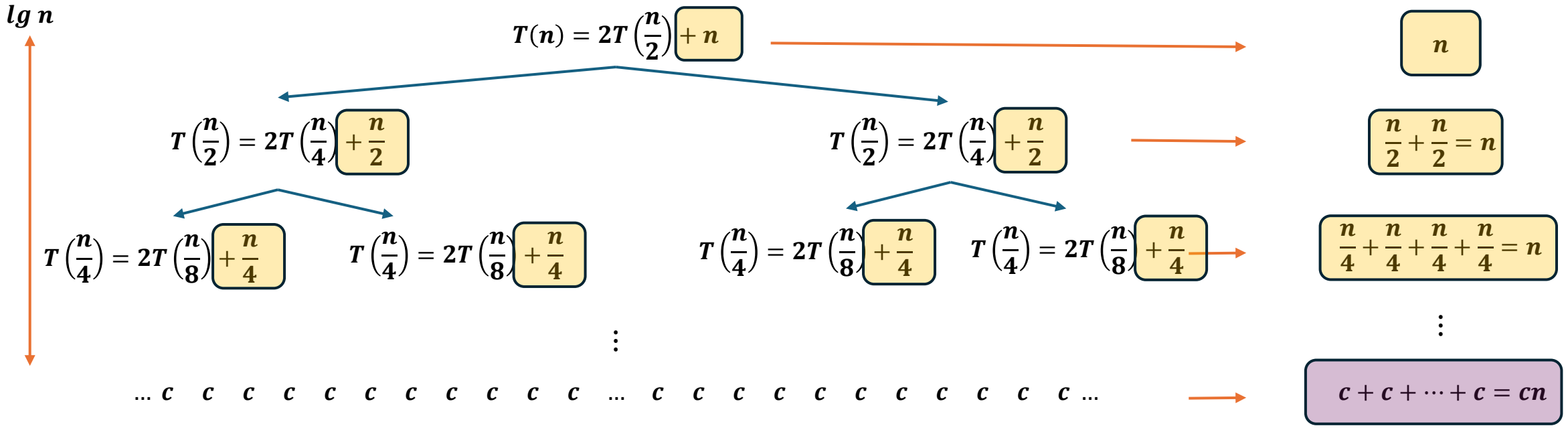
  - $D(n) = \Theta(1)$
  - $C(n) = \Theta(n)$ // Merge의 최악의 경우 수행시간

  - 따라서 $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

```
Merge-Sort(A, p, r)
1       if p >= r // p = r이면 A[p:r] = A[p] 는 정렬된 상태. 따라서 하한
2               return
3       q = (p + r) / 2 // 소숫점 내림 3.5 -> 3        D(n)
4       Merge-Sort(A, p, q) // A[p:q] 재귀적으로 정렬
5       Merge-Sort(A, q+1, r) // A[q+1:r] 재귀적으로 정렬
6       Merge(A, p, q, r) // A[p:q]와 A[q+1:r]을 합쳐 A[p:r]로 만듦
```

Recurrence

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$의 해가 $T(n) = \Theta(n \lg n)$인 이유를 마스터 정리 없이 살펴보자.
  - c는 $n = 1$일 때 문제를 풀 때 걸리는 시간, $n = 2^k (k > 0, k \in N)$꼴로 가정



Recurrence

- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \lg n + cn)$ 이므로 $T(n) = \Theta(n\ lg\ n)$

What is Master Theorem?

- 마스터 방법은 다음과 같은 형식의 점화식을 푸는 기본 지침을 제공함.

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

- 여기서 $a > 0, b > 1$는 상수, $f(n)$은 점근적으로 양인 함수.
  - $f(n)$을 구동 함수(driving function), 위 점근식을 마스터 점화식(master recurrence)이라고 함.
  - $f(n) = D(n) + C(n)$

- 마스터 점화식은 크기 n 짜리 문제를 $\frac{n}{b} < n$ 크기의 부분 문제로 나누는 분할 정복 알고리즘의 수행시간을 설명함.

What is Master Theorem?

- $f(n) = O(n^c)$라고 할 때, a, b, c를 이용한 식은 다음과 같음.

$$T(n) = \begin{cases} O\left(n^{\log_b a}\right) & (c < \log_b a) \\ O\left(n^{\log_b a} \log n\right) & (c = \log_b a) \\ O\left(f(n)\right) & (c > \log_b a) \end{cases}$$

- 수행 시간에 n^c가 영향을 더 주는지, n^c, $aT\left(\frac{n}{b}\right)$가 똑같이 영향을 주는지, $aT\left(\frac{n}{b}\right)$가 더 영향을 주는지 알 수 있게 됨.

What is Master Theorem?

- 병합 정렬에 적용해보면…
- $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
  - $a = 2, b = 2, c = 1$
- $\log_b a = \log_2 2 = 1 = \mathrm{c}$
- 따라서 $T(n) = O\left(n^{\log_b a} \log n\right) = O(n^1 \log n) = O(n \log n)$ $\quad (\log_b a = c)$
- 자세한 증명은 너무 분량이 많아 생략 (https://jjoonleo.tistory.com/28)

Exercise

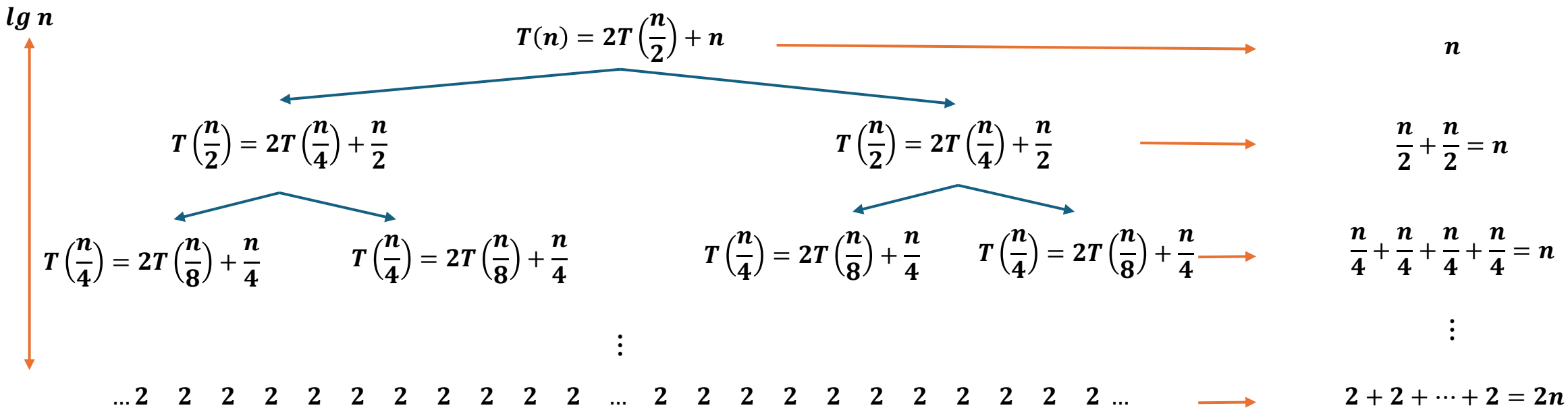
- 연습문제 2.3-4

- 수학적 귀납법을 사용해 $n \geq 2$가 2의 정확한 거듭제곱일 때 다음과 같으면 재귀의 해가 $T(n) = \Theta(n\ lg\ n)$임을 보여라.

$$T(n) = \begin{cases} 2 & if \quad n = 2 \\ 2T\left(\frac{n}{2}\right) + n & otherwise \end{cases}$$

Exercise

- 해답


lg n
$T(n) = 2T\left(\frac{n}{2}\right) + n$
n
$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$
$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$
$\frac{n}{2} + \frac{n}{2} = n$
$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$
$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$
$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$
$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$
$\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} = n$
... 2 2 2 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2 2 2 ...
$2 + 2 + \cdots + 2 = 2n$

Exercise

- $T(n) = \Theta(n \lg n + 2n)$ 이므로 $T(n) = \Theta(n\ lg\ n)$

Exercise

- 종합문제 2.1

- 병합 정렬은 최악의 경우 $\Theta(n\ lg\ n)$ 시간이 걸리고, 삽입 정렬은 $\Theta(n^2)$의 시간이 걸리지만, n이 작으면 상수 매개변수로 인해 삽입 정렬이 많은 기계에서 더 빠르다.
- 그러므로 문제 크기가 충분히 작을 때는 삽입 정렬을 적용하는 것이 유용할 수 있다.
- 즉, 병합 정렬에서 문제 크기가 충분히 작아졌을 때는 재귀호출의 단위를 “덩어리로 만들어“ 삽입 정렬을 적용하는 것이 유용할 수 있다.

- 이제 변형된 병합 정렬을 고려해보자.
- 어떤 정수 k를 정한 뒤 문제를 분할하다가 크기가 k인 n/k개의 부분 리스트가 되면 삽입 정렬을 이용해 정렬하고, 이를 다시 합치는 것은 일반적인 병합 정렬의 구조를 따르도록 한다.

Exercise

- 종합문제 2.1-a

- 삽입 정렬을 이용해 크기가 각각 k인 n/k개의 부분 배열을 최악의 경우 $\Theta(nk)$ 시간에 정렬할 수 있음을 보여라.

Exercise

- 종합문제 2.1-a, 해답
- 크기가 k인 배열을 삽입정렬로 정렬할 때 최악의 경우

$$T(k) = 0 + 1 + 2 + \cdots + k - 1 = \frac{(k-1)k}{2} = \frac{k^2 - k}{2}$$

- 이걸 n/k개의 부분 배열에 다 적용하므로 $\frac{n}{k}T(k) = \frac{nk-1}{2}$
- 따라서, $\Theta\left(\frac{n}{k}T(k)\right) = \Theta(nk)$

Exercise

- 마스터 방법을 사용해 다음 점화식에 대해 엄밀한 점근적 한계를 구하라.

- $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(1)$

- $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(\mathrm{n}^2)$

- $T(n) = 9T\left(\frac{n}{3}\right) + n$