5: Sorting (정렬)

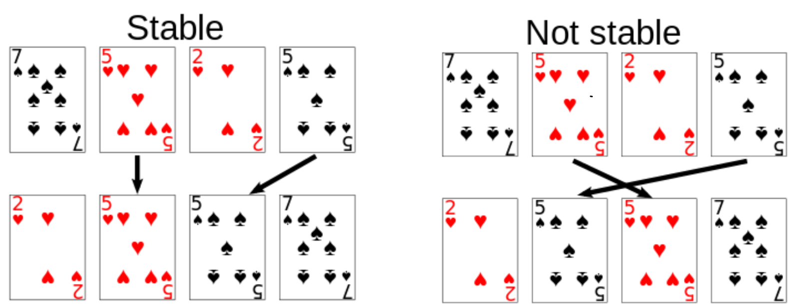
1. 리스트에 저장된 값들을 크기 순서에 따라 재배열하는 문제
    a. 되도록 비교횟수와 자리바꿈 횟수를 최소로 하는 것이 목표
    b. 매우 기본적이고 중요한 알고리즘 문제 중 하나

2. Python에서는 정렬 함수를 이미 제공하고 있다

```
a = [3, 1, 2 -5]
      a.sort()
      print(a)          # a = [-5, 1, 2, 3] 오름차순으로 정렬됨
      a.sort(reverse = True)
      print(a)          # a = [3, 2, 1, -5] 내림차순으로 정렬됨
b = sorted(a)           # a의 값의 순서는 변하지 않고, 정렬되어 b로 복사
```

3. 단순 분류;
    a. 기본 정렬 알고리즘: 느리지만 단순한 정렬 알고리즘
        i. insertion, selection, bubble 정렬 알고리즘 등
    b. 빠른 정렬 알고리즘: 빠르지만 복잡할 수도 있는 정렬 알고리즘
        i. quick, heap, merge 정렬 알고리즘 등
    c. 특별한 상황에 맞는 정렬 알고리즘
        i. count, radix, bucket 정렬 알고리즘 등

4. 정렬 알고리즘의 성질 2가지
    a. stable vs. unstable
        i. 크기가 같은 숫자인 경우, 원래 입력 순서를 정렬 후에도 유지하는 알고리즘을 stable하다고 함
        ii. 이미지 출처: https://en.wikipedia.org/wiki/Sorting_algorithm



    b. in-place vs not-in-place → 추가메모리 : O(n) 만큼 사용 (입력의 길이 만큼 사용)
        i. 상수 개의 추가 변수(메모리)만 사용한 정렬 알고리즘을 in-place하다고 함

추가메모리 양 : O(1)

5. **기본정렬 알고리즘**:

a. 느리지만 간단한 알고리즘들...

b. Insertion sort 알고리즘 [code] ⇒ 특정 데이터를 적절한 위치에 삽입한다. (특정한 데이터가 삽입될 때 그 앞의 데이터까지는 이미 정렬 되어있다고 가정)

```
def insertion_sort(A, n): # A[0] ~ A[n-1]까지 insertion sort
    for i in range(1, n):
        j = i-1
        while j >= 0 and A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
            j = j - 1
```

– 2번째 인덱스부터 시작 (첫번째 이전 데이터 ⇒ 정렬됨)

⇒ 삽입될 데이터보다 작은 데이터를 만나면 그 위치에서 멈춤

insertion: for each round: insert a value at correct position ($n-1$)

$1+2+\cdots+n-1$

$1+2+\cdots n-1$

비교 $\frac{n(n-1)}{2}$ 최악 $O(n^2)$

교환 $\frac{n(n-1)}{2}$

정렬 그럼에 있는 ($O(n)$)

c. Selection sort 알고리즘 [code] ⇒ 가장 큰 데이터 선택 맨앞 ⇒ 거기 배정

```
def selection_sort(A, n):     # A[0] ~ A[n-1]까지 selection sort
    for i in range(n-1, 0, -1):
        m = get_max_index(A, i) # A[0]~A[i] 중 최대값 인덱스 구함
        A[i],A[m] = A[m],A[i]   # A[m]이 최대값이므로 A[i]에 배치
                                # 자리바꿈(swap)이 발생
```

Selection $O(n^2)$

for each round: ($n-1$회)

– find max ⇒ $(n-1)+(n-2)\cdots+1 = \frac{n(n-1)}{2}$ 비교

– swap max and last ⇒ 1 ↳ 총 $n-1$회 교환

$\frac{n(n-1)}{n}$ +

d. Bubble sort 알고리즘 [code] ⇒ 두 개씩 비교

```
def bubble_sort(A, n):  # A[0] ~ A[n-1]까지 bubble sort
    for i in range(n):  # round i
      for j in range(1, n):
        if A[j-1] > A[j]: # A[n-1]~A[i] 인접한 수 비교 후 swap
          A[j-1], A[j] = A[j], A[j-1]
```

bubble: for each round:

– 오른쪽씩 비교 $(n-1)+(n-2)+\cdots 1$

최댓값 ⇒ 마지막 자리

$n-1+n-2+\cdots 1$

- 비교 $\frac{n(n-1)}{2}$

- 교환 $\frac{n(n-1)}{2}$

e. 아래 리스트의 값을 위의 세 가지 알고리즘으로 정렬해보자. 어떻게 값들이 재배치되는지 단계별로 알아보자

A = [12, 4, 9, 10, 21, 3, 8, 0, 7, 9, 6]

- Insertion
- Selection
- Bubble

f. 수행시간 (최악의 경우) - stable - in-place

i. insertion: $O(n^2)$ - stable - inplace

ii. selection: $O(n^2)$ - unstable - inplace

iii. bubble: $O(n^2)$ - stable - inplace

6. [**매우 중요!**] **Quick sort 알고리즘**
   a. 실제로 구현해서 실행해 본 여러 결과에 따르면 가장 빠른 정렬 알고리즘임
   b. 전형적인 divide-and-conquer 알고리즘의 예 중 하나
   c. Quick select 알고리즘과 매우 유사. 정렬 알고리즘이므로, 피봇(pivot)보다 작은 값들을 리스트의 왼쪽으로, 큰 값들을 리스트의 오른쪽으로 재배열 한 후, 양 쪽을 재귀적으로 다시 정렬하는 식으로 진행됨
   d. **Pseudo code**: `quick_sort(A, first, last) # A[first] ~ A[last]`를 퀵 정렬
      i. `A[first], ..., A[last]`까지 quick sort를 함
      ii. `first >= last`인 경우는 정렬할 값이 1개 이하이므로 그냥 리턴
      iii. `first < last`인 경우는 quick selection과 유사하게 pivot을 정해 나눔
         1. `pivot = A[first]` (pivot을 가장 왼쪽의 값으로 정함. 어떻게 정해도 상관없음)
         2. `A`의 값들을 pivot보다 작은 값은 왼쪽에 큰 값들은 오른쪽에 오도록 재배치
         3. pivot보다 작은 값들에 대해 quickSort 재귀호출 + 큰 값들에 대해 quickSort 재귀호출
   e. **구현 방법 1**: 리스트 A에서 pivot을 기준으로 나누는 방법 (in-place)

```
def quick_sort(A, first, last):
    if first >= last: return
    left, right = first+1, last
    pivot = A[first]
    while left <= right:
        while left <= last and A[left] < pivot:
            left += 1
        while right > first and A[right] > pivot:
            right -= 1
        if left <= right: # swap A[left] and A[right]
            A[left], A[right] = A[right], A[left]
            left += 1
            right -= 1
    # place pivot at the right place
    A[first], A[right] = A[right], A[first]
    quickSort(A, first, right-1)
    quickSort(A, right+1, last)
```

   f. **구현 방법 2**: 새로운 리스트를 추가로 사용하여 나누는 방법 (not-in-place) Stable

```
def quick_sort(A): # first, last가 필요없음
    if len(A) <= 1: return A
    pivot = A[0]
    S, M, L = [], [], []
    for x in A:
        if x < pivot: S.append(x)
```

$T(n) = T(|S|) + T(|L|) + cn$

$\boxed{W.C}$: $T(n) = T(n-1) + cn$
$= \boxed{O(n^2)}$

$B.C: T(n) = T(\frac{n}{2}) + (\frac{n}{2}) + cn$
$= 2T(\frac{n}{2}) + cn$
$= O(n \log_2 n)$

Average Case $= O(n \log_2 n)$

```
        elif x > pivot: L.append(x)
        else: M.append(x)
    return quick_sort(S)+M+quick_sort(L)
```

g. `A = [4, 2, 5, 8, 6, 2, 3, 7, 10]`인 경우, 위의 두 가지 구현방법에 따라 시뮬레이션 해보자

h. stable?

i. in-place?

j. **수행시간**: 전적으로 pivot보다 작은 값들의 수와 큰 값들의 수, 즉 어떤 비율로 분할되는 가에 따라 결정된다! → quick_select 알고리즘의 수행시간과 분석과 유사

Best case

i. 가장 좋은 시나리오: 대략 `n/2`개 - `n/2`개로 계속 분할되는 경우

1. T(n) = $T(\frac{n}{2}) + T(\frac{n}{2}) + cn$

$= 2 \cdot T(\frac{n}{2}) + cn$

$= O(n \log n)$

worst case

ii. 가장 나쁜 시나리오: `0`개 - `(n-1)`개 (또는 `(n-1)-0`)으로 계속 분할되는 경우

1. T(n) = $T(n-1) + cn$

$= O(n^2)$

iii. [**고급**] 평균 수행 시간(average running time)으로 계산해보면?

Average-case time analysis

1. Consider n numbers $[1, 2, \dots, n]$, and its random permutation as an input.
2. Let X_{ij} be a random variable to represent the comparison of i with j during the execution of `quick_sort`, where $X_{ij} = 1$ if they are compared, 0 otherwise.
3. $X = \sum_{1 \le i < j \le n} X_{ij}$ is the number of comparisons performed in `quick_sort`.
4. The expected number $E[X]$ of the comparisons is:

$$E[X] = \sum_{i,j} E[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr(X_{ij} = 1)$$

5. Observation: $X_{ij} = 1$ if and only if one of i and j should be the first pivot of $[i, \dots, j]$; otherwise, they would be in different groups, so they are never compared each other.
6. Since a sequence of pivots is also a random permutation of $1, \dots, n$, the probability that one of them is the first pivot is $2/(j - i + 1)$.

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \le 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \le \sum_{i=1}^{n} H_n = 2cn(\ln n + c') = O(n \log n)$$

Another proof

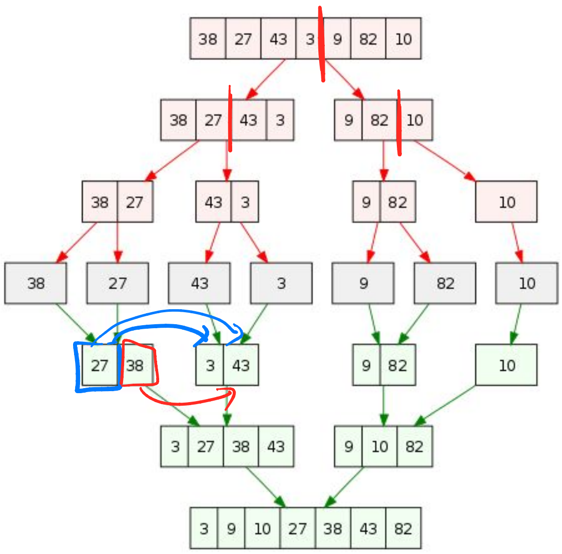
$$\begin{array}{rcl}
T(n) & = & cn + \frac{1}{n} \sum_{k=1}^{n} T(k-1) + T(n-k) \\
 & = & cn + \frac{2}{n} T(0) + \frac{2}{n} \sum_{k=1}^{n-1} T(k) \\
nT(n) & = & cn^2 + 2T(0) + 2 \sum_{k=1}^{n-1} T(k) \\
(n-1)T(n-1) & = & c(n-1)^2 + 2T(0) + 2 \sum_{k=1}^{n-2} T(k) \\
\hline
nT(n) - (n-1)T(n-1) & = & c(2n-1) + 2T(n-1) \\
nT(n) - (n+1)T(n-1) & = & c(2n-1) \ \text{ divided by } n(n+1) \\
\frac{1}{n+1} T(n) - \frac{1}{n} T(n-1) & = & c \frac{2n-1}{n(n+1)} \\
\hline
 & & \text{Let } S(n) = T(n)/(n+1). \\
S(n) - S(n-1) & \le & 2c/(n+1) \\
S(n) - S(1) & \le & 2c \left(\frac{1}{n+1} + \cdots + \frac{1}{3} \right) \\
S(n) & \le & 2c \left(\frac{1}{n+1} + \cdots + \frac{1}{2} + 1 \right) \\
 & = & 2cH_{n+1} \\
T(n) & = & 2cH_{n+1}(n+1) = O(n \log n)
\end{array}$$

7. Merge sort 알고리즘

a. 전형적인 divide-and-conquer 알고리즘 중 하나

b. 최악의 경우에도 `O(n logn)`에 동작하는 최적 정렬 알고리즘 중 하나
   - i. quick_sort의 가장 큰 문제는 분할된 크기가 비슷하지 않아 재귀를 많이 하게 되어 수행시간이 커진다는 것!
   - ii. 애초에 강제로 균등하게 분할하면 되지 않을까 하는 아이디어에서 착안

c. IDEA: merge_sort(A, first, last): # A[first] ~ A[last] merge sort 하기
   - i. `if first >= last: return` # 정렬할 원소가 하나이하면 정렬 불필요
   - ii. `merge_sort(A, first, (first+last)//2)` # 앞 부분 반 재귀적으로 정렬
   - iii. `merge_sort(A, (first+last)//2+1, last)` # 뒷 부분 반 재귀적으로 정렬
   - iv. `merge_two_sorted_lists(A, first, last)` # 현재 A는 앞 쪽 반과 뒷 쪽 # 반이 정렬되어 있으므로 두 정렬된 부분을 합병하는 함수



d. [예: 위의 그림]
   - i. **분할**: 숫자 하나씩 분할될 때까지 반씩 분할해 간다
   - ii. **정복**: 다시 분할 순서의 반대로 정렬된 두 리스트를 합병해 올라간다

e. Pseudo code [`merge_two_sorted_lists`]

```
def merge_two_sorted_list(A, first, last):

      m = (first + last)/2
      i, j = first, m+1
      B = []
      while i <= m and j <= last:
            if A[i] <= A[j]:
                  B.append(A[i])
```

```
                i += 1
            else:
                B.append(A[j])
                j += 1
        for k in range(i, m+1): # why?
            B.append(A[k])
        for k in range(j, last+1): # why?
            B.append(A[k])
        for i in range(first, last+1):      # A ← B
            A[i] = B[i - first]
```

f. stable? O

g. in-place? X

h. 수행시간 T(n)을 위한 점화식 $= (n\log n)$
   i. [❓] T(n) = $2 \cdot T(\frac{n}{2}) + cn$, T(1) = c

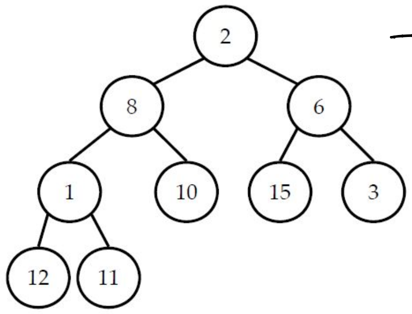
i. **[응용]** 입력 리스트의 숫자들에 대한 inversion 개수 세기 문제
   i. 예: A = [2, 4, 1, 3, 5]라면 (2, 1), (4, 1), (4, 3) 세 쌍의 숫자가 서로 역전되어 있으므로 inversion(A) = 3이 된다

   ii. **방법 1**: 가장 단순한 알고리즘은?
      1. 모든 쌍 (a, b)를 보면서 a > b인지 검사해본다
      2. [❓] 이 방법의 수행시간은?

   iii. 방법 2: 보다 빠른 알고리즘은? [**힌트**: merge 정렬의 분할정복 절차를 이용?]
      1. O(n logn) 시간 알고리즘이 존재한다!
      2. merge_sort처럼 반으로 나눠 분할한다
      3. 왼쪽 반의 inversion 개수 L을 재귀적으로 계산해 알고 있고, 오른쪽 반의 inversion 개수 R을 알고 있다고 하자. (분할해서 정렬하는 대신 inversion 개수를 세었다!)
      4. L+R이 전체 inversion의 개수인가? 아니면 어떤 inversion이 빠졌나?
      5. 결국, 왼쪽 반에 있는 수와 오른쪽 반의 있는 수의 쌍이 inversion 쌍인지 고려하지 않았다. 이 쌍의 개수를 M이라 하면,, 전체 inversion 갯수는 L+M+R이 된다
      6. 그러면, M을 어떻게 계산할까? 바로 merge_two_sorted_lists와 유사한 방식으로 두 리스트의 값을 비교하면서 inversion 갯수를 계산하면 된다. ([❓] 구체적으로 어떻게 하면 될까?)

8. **Heap sort 알고리즘**: 자료구조 수업에서 이미 배운 정렬 알고리즘

왼쪽자식노드 2i+1
오른쪽자식노드 2i+2
부모노드 (i-1)//2



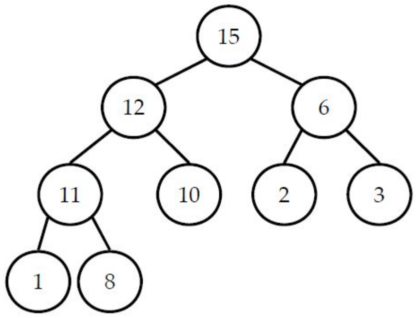
O(n log n)

A									
A	2	8	6	1	10	15	3	12	11

[힙 성질 만족 안함!]

a. **힙**(**heap**): 다음의 모양과 힙 성질을 만족하는 리스트에 저장된 값의 시퀀스
   i. (**모양 성질**) 다음과 같은 이진트리여야 한다:
      1. 마지막 레벨을 제외한 각 레벨의 노드는 모두 채워져 있어야 한다
      2. 마지막 레벨에선 노드들이 왼쪽부터 채워져야 한다
   ii. (**힙 성질**) 루트 노드를 제외한 모든 노드에 저장된 값(key)은 자신의 부모 노드의 값보다 크면 안된다
      1. 위의 그림의 이진트리는 모양성질은 만족하지만 힙 성질은 만족하지 않는다. (8의 부모가 2가 되어 성질 위배)
      2. 반면에 아래 그림은 모양과 힙 성질 모두 만족한다

부모노드의 key값 ≥ 자식노드의 key값



A	15	12	6	11	10	2	3	1	8

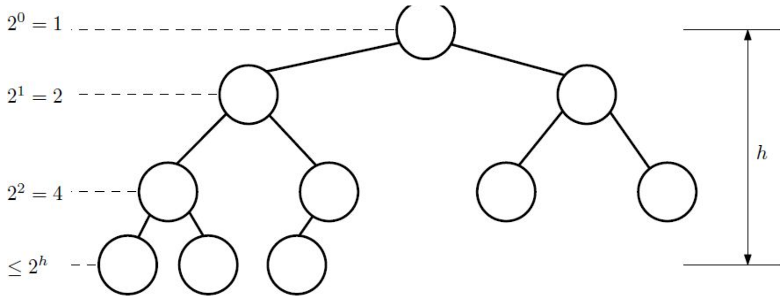
[힙 성질을 만족함!]

   iii. [**매우 중요**] 힙 성질에 따라 루트 노드에는 가장 큰 값이 저장되게 된다
   iv. 여기서 주의할 건, 실제 데이터 값은 리스트에 저장되어 있고 리스트의 값이 표현하는 (가상의) 이진트리가 모양 성질을 만족한다는 의미이다
   v. **힙의 높이** h:
      1. n개의 값으로 구성된 힙의 높이 h는 최대 어느 정도일까?
      2. 모양 성질에 따르면, 레벨 0부터 h-1에 있는 노드는 모두 채워져 있어야 하므로, 노드 개수가 총 $1 + 2 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$이다

make heap   A+heap : O(nlogn) → O(n)도 가능.

heapify : O(logn)

3. 레벨 h에는 하나 이상의 노드가 존재하므로 전체 노드 수 n >= 2^h 이 성립한다
4. 양변에 $\log_2$를 취하면 h <= log n이 성립한다. h = O(log n)이다



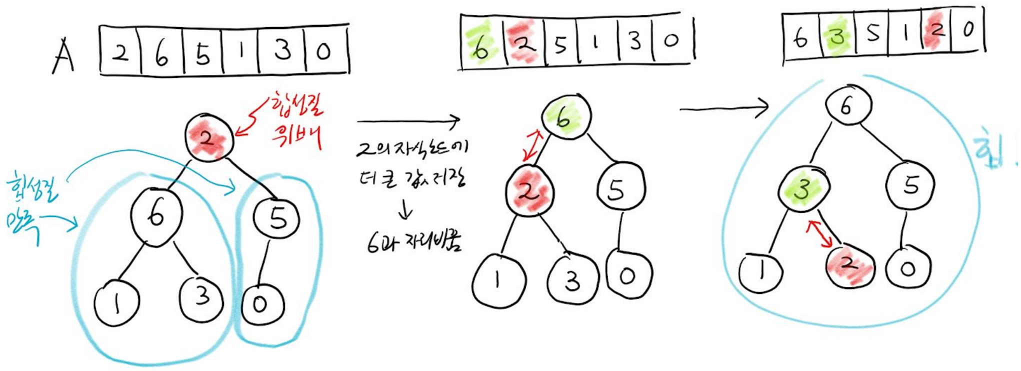
vi. Heap 클래스:

```
class Heap:
    def __init__(self, L=[]): # default: 빈 리스트
        self.A = L
        self.make_heap(A) # A의 값을 힙성질이 만족되도록
    def __str__(self):
        return str(self.A)
```

vii. 리스트에 저장된 값을 이진트리로 해석하면 **<u>자동으로</u>** 모양 성질을 만족한다. 그러나 힙 성질을 만족하지 않을 수도 있다. 위의 두 번째 경우가 그런 예이다

viii. 힙 성질을 만족하지 않으면, 값들을 재배열해서 힙을 만들 수 있다 → make_heap 함수 → 이를 위해 heapify_down 함수가 필요

ix. heapify_down(k) 함수:
   1. A[k]의 자손 노드들은 모두 힙 성질을 만족한다고 할 때, A[k] 값을 (필요하다면) 아래로 내려가면서 힙 성질을 만족하는 위치로 이동시키는 함수.



```
def heapify_down(self, k, n):
        # n = 힙의 전체 노드수 [heap_sort를 위해 필요함]
        # A[k]가 힙 성질을 위배한다면, 밑으로
        # 내려가면서 힙 성질을 만족하는 위치를 찾는다
        while 2*k+1 < n:  # [❓] 조건문이 어떤 뜻인가?
                L, R = 2*k + 1, 2*k + 2 # [❓] L, R은 어떤 값?
                if L < n and self.A[L] > self.A[k]:
                        m = L
                else:
                        m = k
                if R < n and self.A[R] > self.A[m]:
                        m = R
                # m = A[k], A[L], A[R] 중 최대값의 인덱스

                if m != k:  # A[k]가 최대값이 아니라면 힙 성질 위배
                        self.A[k], self.A[m] = self.A[m], self.A[k]
                        k = m
                else: break # [❓] 왜 break할까?
```

2. **heapify_down의 수행시간**을 알아보자
    - 수행시간은 A[k]가 내려온 레벨 수에 비례한다
    - 가장 많이 내려오려면 루트인 A[0]가 힙의 높이(가장 깊은 곳의 리프)까지 내려오는 것이다.
    - 한 레벨 내려올 때의 연산은 상수번의 비교를 하고 1번의 자리바꿈을 하는 것이므로 O(1) 시간이면 충분하다.
    - 따라서, 최악의 경우에 힙의 높이에 비례하는 시간이 필요하다. 즉, **O(log n) 시간**이 필요하다

x. make_heap: 현재 리스트의 값들을 힙 성질을 만족하도록 재배열하는 함수
1. 리스트 A의 각 값에 대해 heapify_down을 호출해 재배치한다
2. 여기서 어떤 값부터 차례로 heapify_down을 호출해야 할까?

```
def make_heap(self):
    n = len(self.A)
    for k in range(n-1, -1, -1): # A[n-1]→...→A[0]
            self.heapify_down(k, n)
```

3. range(n-1, -1, -1) → range(n//2-1, -1, -1)로 변경해도 문제 없음 (왜 그럴까? [❓])

4. make_heap의 수행시간을 분석해보자
    - **분석 1**: for 반복문은 n번 반복되고, 반복할 때마다 heapify_down이 한번씩 호출된다. heapify_down의 수행시간이 O(log n)이므로 **O(n log n)** 시간이면 충분하다

- **[고급] 분석 2**: 엄밀하게 분석하면 O(n) 시간이면 충분하다 (아래 분석 참조. 건너 띄어도 됨)

make_heap 의 수행시간

1. level i에 있는 노드 수는 (최대) 2^i 이다.
2. level i에 있는 값은 make_heap 에서 움직이는 레벨의 수가 $h-i$가 된다. 즉 $O(h-i)$ 시간이 필요하다.
3. 결국 level i에 있는 모든 노드에 대해 수행한 시간을 합하면 $2^i \times O(h-i) = O((h-i)2^i)$이다.
4. 모든 $i = 0, \ldots, h$까지 이 값을 더하면, (여기서 $c > 0$인 상수임.)

$$S = \sum_{i=0}^{h} c(h-i)2^i = c\sum_{i=1}^{h} i2^{h-i}$$

5. $2S$를 계산한 후, $2S$에서 S를 빼면 다음의 식을 얻는다. $2^h + 1 < n \le 2^{h+1}$이라는 사실을 이용하면,

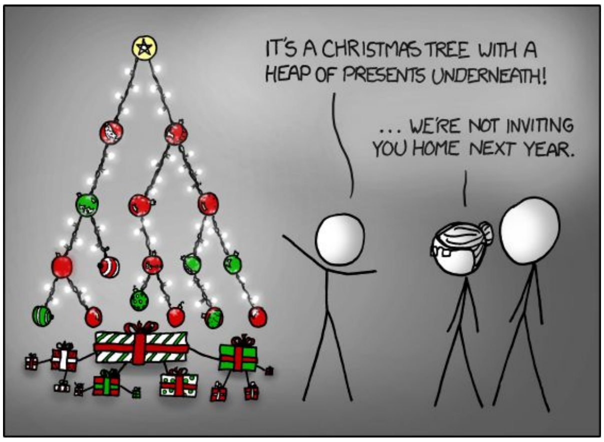
$$S = c(-2^{h-1} - h + \sum_{i=1}^{h} 2^i) = c(-2^{h-1} - h + 2(2^{h-1} - 1)) = c(2^{h-1} - h - 2) \le \frac{c}{4}n - c\log n - 2 = O(n).$$

> make_heap 의 수행시간은 $O(n)$이다.

xi. heap_sort(): make_heap과 heapify_down 함수를 반복적용하여 값들을 오름차순으로 재배치하는 함수 → 힙 정렬(heap sort) 알고리즘이라 불림

1. [**핵심**] 가장 큰 값이 힙의 루트에 저장되어 있다. 이 값은 정렬 순서에 따라 리스트의 가장 마지막에 위치해야 한다
   - A[0], A[n-1] = A[n-1], A[0] (두 값을 교환 - swap)
   - A[0]의 값은 힙 성질을 만족하지 않을 수 있으므로 heapify_down(n-1, 0)를 호출해서 성질을 만족하는 자리로 이동
   - 이제 힙은 (n-1)개의 값(A[0] … A[n-2])으로 구성되므로 n = n - 1로 변경 후, 위의 두 과정을 다시 반복

```
def heap_sort(self):
    n = len(self.A)
    for k in range(len(self.A)-1, -1, -1):
        self.A[0],self.A[k] = \
            self.A[k],self.A[0]
        n = n - 1   # A[n-1]은 정렬되었으므로
        self.heapify_down(0, n)
```



2. 춤으로 표현한 힙 동영상: https://youtu.be/Xw2D9aJRBY4

3. [💪] [해보기] 위에서 설명한 Heap 클래스 구현한 후 다음을 수행해보자

```
heap = Heap([2,8,6,1,10,15,3,12,11])
# 이미 Heap 객체 생성시 make_heap이 호출되어 힙이 됨
print(heap)        # [15, 12, 6, 11, 10, 2, 3, 1, 8]
heap.heap_sort()
print(heap)        # [1, 2, 3, 6, 8, 10, 11, 12, 15]
```

xii. 삽입 연산 insert(key): 기존의 힙에 새로운 key 값을 삽입하는 연산

1. 힙 리스트 A의 가장 오른쪽에 새로운 값 x를 저장하고, 이 값을 힙 성질이 만족하도록 위치를 재조정해야 한다
2. 이 경우엔 x가 힙의 리프에 위치하므로, 루트 노드 방향으로 올라가면서 자신의 위치를 조정하면 된다 → heapify_up 함수 구현

```
def heapify_up(self, k): # 올라가면서 A[k]를 재배치
      while k>0 and self.A[(k-1)//2] < self.A[k] :
            self.A[k], self.A[(k-1)//2] = \
                        self.A[(k-1)//2], self.A[k]
            k = (k-1)//2

def insert(self, key):
      self.A.append(key)
      self.heapify_up(len(self.A)-1)
```

3. **수행시간**: 가장 마지막 레벨의 가장 오른쪽 빈 칸에 삽입되어 루트 노드까지 올라갈 수 있으므로 heapify_up과 insert 모두 힙의 높이만큼의 시간이 필요 → **O(log n)**

xiii. 삭제 연산 delete_max(): 임의의 값을 삭제하는 것이 아닌 루트 노드에 저장된 가장 큰 값을 삭제후 리턴하는 함수

1. 루트 노드의 값을 삭제 후 리턴해야 하므로, A[0]를 A[n-1]의 값으로 대체하고, heapify_down(0, n-1)를 호출해 A[0]를 재배치한다

```
def delete_max(self):
        if len(self.A) == 0: return None
        key = self.A[0]
        self.A[0], self.A[len(self.A)-1] = \
                self.A[len(self.A)-1], self.A[0]
        self.A.pop()       # 실제로 리스트에서 delete!
        heapify_down(0, len(self.A)) # len(A) = n-1
        return key
```

2. **수행시간**: heapify_down이 1번 호출되고, 이 함수의 수행시간이 전체 시간을 결정하므로 **O(log n)** 시간에 수행

xiv. **연산 및 수행시간 정리** (n개의 값을 저장한 리스트와 힙에 대해)
1. heapify_up, heapify_down: O(log n)
2. make_heap = n times x heapify_down = O(n log n) → O(n)
3. insert = 1 x heapify_down: O(log n)
4. delete_max = 1 x heapify_down: O(log n)
5. heap sort = make_heap + n x heapify up = O(nlog n)

xv. [Python] heapq 모듈에서 heap 관련 함수를 그대로 제공한다
1. 단, 부모노드에 자식노드보다 더 크지 않은 값이 저장되는 min-heap임에 유의하자 (노트에서 다룬 것은 max-heap임!)
2. import heapq
3. 힙 자체는 리스트를 사용한다: h = []
4. 지원연산:
   - heappush(h, key): 힙 h에 key 값을 삽입 (= insert와 동일)
     i. heappush(h, (key, value))처럼 튜플 삽입 가능
   - heappop(h): 최소값을 지우고 리턴 (delete_min의 역할)
   - heapify(A): 리스트 A를 힙 성질이 만족되도록 변환
     i. make_heap()과 동일 (단, min-heap으로 변환)
   - h[0]: 힙의 최소값을 알고 싶다면

9. **반드시 필요한 최소 비교 횟수는 얼마인가? [하한, lower bound]**
    a. n개의 값을 비교해서 정렬하는 데 필요한 최소 비교횟수는 얼마일까?

    b. 이를 정렬 문제의 비교횟수에 대한 하한(lower bound)이라 부른다
        i. 어떤 정렬 알고리즘도 이 최소 비교횟수보다 더 작은 횟수로 정렬하기란 불가능함을 의미한다

    c. 예를 들어, 세 개의 서로 다른 값 a, b, c가 있다고 하자. a, b, c 값이 어떻게 주어지더라도(즉, 최악의 경우) 최소 몇 번의 비교는 해야할까?
        i. a < b이고 b < c라면 (2번의 비교 결과) a < b < c로 결론 (2번 비교 충분)
        ii. a < b이고 b > c라면 b가 가장 크다는 건 알지만, a와 c의 크기 순서는 모른다. 그래서 a ? c 비교를 한 번 더 해야 한다 (3번 비교 필요)
            1. a < c 이면, a < c < b로 결론
            2. a > c 이면, c < a < b로 결론
        iii. a > b인 경우에도 유사하게 세 번의 비교가 필요한 경우가 있다
            1. b < c < a인 경우, c < b < a인 경우, b < a < c인 경우 세 가지 중 하나의 결론에 도달
        iv. 결국, 6가지 결론(3! = 6) 중 하나에 도달하는 것이 정렬 알고리즘의 목표다
        v. 이를 위해, 최소 3번의 비교가 필요한 경우가 존재하므로, 세 개의 값을 비교 정렬하기 위해서 필요한 비교횟수의 하한은 3이다
        vi. 그림으로 표현하면 아래와 같다

    d. 이 논리를 n개의 값 `A[0] ~ A[n-1]`에 대한 최소 비교횟수를 증명하는 데 이용할 수 있을까? 답은 YES!
        i. 이러한 증명법을 **<u>(algebraic) decision tree</u>** 하한 증명법이라 부른다

    e. 두 수를 한 번 비교하면 그 결과에 따라 왼쪽과 오른쪽으로 나뉘고, 다시 같은 비교 과정을 반복하면 트리(tree) 모양이 된다.
        i. 각 노드는 비교이고, 각 에지는 비교의 결과(`True/False`)를 나타낸다
        ii. 리프 노드는 정렬의 결과(경우) 중 하나이다
    f. 이 트리의 리프 노드 개수는? [힌트: `n = 3`인 경우를 그대로 확장해서...]
        i. ____________________

    g. 어떤 비교를 통해 n개의 값을 정렬하는 알고리즘도 여러번의 비교를 통해 리프 노드에 도달하게 된다. 이 알고리즘이 수행한 비교는 루트 노드에서 리프

노드까지 방문한 노드(비교)의 개수와 같다. 가장 깊은 곳의 리프 노드까지의 경로의 길이가 트리의 높이(height)이므로 최소한 결정 트리의 높이만큼의 비교는 반드시 필요하게 된다!

h. `n!` 개의 리프 노드를 갖는 이진 트리의 최소 높이 h는 얼마일까?
   i. [❓] 왜 최소 높이를 생각할까?

   ii. [❓] h는 얼마일까?

   iii. [질문] `log(n!) >= ?`

   iv. **결론**: 비교를 통해 정렬하는 어떤 알고리즘도 최소 `cn logn`번 이상의 비교가 반드시 필요하다
      1. `merge_sort`는 더 이상 잘 할 수 없는 **최적(optimal)** 알고리즘임!!

10. **Radix sort** 알고리즘
   a. 실제 정렬하는 값의 범위가 정렬 전에 이미 알려져 있는 게 일반적이다
   b. 예를 들어, 시험 점수를 정렬한다면, 0 부터 100 사이의 정수인 경우가 많다
   c. 이렇게 정렬할 값의 범위가 정해진 경우, 이 범위 정보를 이용하면 더 빠르게 정렬할 수 있다 → radix 정렬, count 정렬 등
   d. radix 정렬 알고리즘은 두 값을 비교해서 정렬하는 알고리즘이 아니므로, `cnlog n`번의 비교가 반드시 필요한 것이 아님에 유의하자!
   e. 자리수가 최대 d개인 정수 n개를 radix 정렬한다고 가정하자
   f. 예: `A = [10, 1234, 9, 7234, 67, 9181, 733, 197, 7, 3] n = 10, d = 4`

0	1	2	3	4	5	6	7	8	9

   g. Pseudo code:

```
B = 10                          # B는 진수를 의미. 십진수이므로 10으로 정함
for i in range(d):
      slots = [[0], ..., [0]] # 리스트 slots을 B개의 빈 리스트로 구성
      for a in A:
            slots[ a%(B**i) ].append(a)
      A = []
      for i in range(B):   # slots에 있는 값을 다시 차례로 모음
            A += slots[i]
      del slots
```

   h. [❓] 수행시간을 계산해보자: dx O(n)
   i. [❓] 수행시간과 비교횟수 하한을 비교해보자

11. 정렬 알고리즘 비교횟수 정리 표 [wikipedia] (Big-O 기호는 생략함)

	Best	Average	Worst	Memory	Stable	In-place	Note
Insertion	n	n^2	n^2	1	Yes	Yes	
Selection	n^2	n^2	n^2	1	No	Yes	
Bubble	n^2	n^2	n^2	1	Yes	Yes	tiny code
Quick	nlogn	nlogn	n^2	logn ~ n	No	Yes	practically quickest
Merge	nlogn	nlogn	nlogn	n	Yes	No	easy to parallelize
Heap	nlogn	nlogn	nlogn	1	No	Yes	
Radix	-	dn	dn	n	Yes	Yes	Non-comparison