2. 정렬의 개요와 선택 정렬(Selection Sort)

```
#include <stdio.h>

int main(void) {
    int i, j, min, index, temp;
    int array[10] = {1, 10, 5, 8, 7, 6, 4, 3, 2, 9};
    for(i = 0; i < 10; i++) {
        min = 9999;
        for(j = i; j < 10; j++) {
            if(min > array[j]) {
                min = array[j];
                index = j;
            }
        }
        temp = array[i];
        array[i] = array[index];
        array[index] = temp;
    }
    for(i = 0; i < 10; i++) {
        printf("%d ", array[i]);
    }
    return 0;
}
```

함께 위 소스코드를 작성해서 실행해봅시다. 위 소스코드에서 min은 가장 작은 숫자를 일시적으로 담는 변수이고, temp는 배열에서 두 숫자의 위치를 서로 바꾸기 위해 사용하는 변수입니다.

5. 퀵 정렬(Quick Sort)

퀵 정렬은 하나의 큰 문제를 두 개의 작은 문제로 분할하는 식으로 빠르게 정렬합니다. 더 쉽게 말하자면 특정한 값을 기준으로 큰 숫자와 작은 숫자를 서로 교환한 뒤에 배열을 반으로 나눕니다.

바로 한 번 예시를 통해서 살펴보도록 합시다. 일반적으로 퀵 정렬에서는 기준 값이 있습니다. 이를 **피벗(Pivot)**이라고도 하는데, 보통 첫 번째 원소를 피벗 값으로 설정하고 사용합니다. 다음과 같이 1이라는 값이 먼저 피벗 값으로 설정이 되었다고 생각해봅시다.

(1) 10 5 8 7 6 4 3 2 9

이 경우 1보다 큰 숫자를 왼쪽부터 찾고, 1보다 작은 숫자를 오른쪽부터 찾습니다. 이 때 1보다 큰 숫자로는 바로 10을 찾을 수 있고, 1보다 작은 숫자는 찾지 못해서 결국 1까지 도달합니다. 이 때 작은 값의 인덱스가 큰 값의 인덱스보다 작으므로 피벗 값과 작은 값의 위치를 바꿉니다. 즉, 1과 1을 교환하므로 제자리 걸음입니다.

1 10 5 8 7 6 4 3 2 9

따라서 위와 같이 구성되고, 이 때 피벗 값이었던 1의 왼쪽에는 1보다 작은 값이 존재하며 오른쪽에는 1보다 큰 값이 존재합니다. 이제 이어서 왼쪽과 오른쪽에서 퀵 정렬을 순환적으로 수행하는 겁니다.

1 (10) 5 8 7 6 4 3 2 9

이 때 왼쪽은 없으므로 무시하고, 오른쪽에서는 피벗 값으로 10이 채택됩니다. 10보다 큰 값을 왼쪽부터 찾고, 10보다 작은 값을 오른쪽부터 찾습니다. 큰 값은 찾지 못하게 되며 작은 값으로는 바로 9를 찾을 수 있습니다. 이 때 작은 값의 인덱스가 큰 값의 인덱스보다 작으므로 9와 10을 교환합니다.

1 5 8 7 6 4 3 2 **10**

따라서 위와 같이 구성되고, 이 때 피벗 값이었던 10의 왼쪽에는 10보다 작은 값이 존재하며 오른쪽에는 10보다 큰 값이 존재합니다. 이제 이어서 왼쪽과 오른쪽에서 퀵 정렬을 순환적으로 수행합니다.

1 (5) 8 7 6 4 3 2 **10**

이 때 오른쪽은 없으므로 무시하고, 왼쪽에서는 피벗 값으로 5가 채택됩니다. 5보다 큰 값을 왼쪽부터 찾고, 5보다 작은 값을 오른쪽부터 찾습니다. 큰 값으로는 8이 선택되고, 작은 값으로는 2가 선택됩니다. 이 때 작은 값의 인덱스가 큰 값의 인덱스보다 크므로 8과 2를 교환합니다.

1 (5) 2 7 6 4 3 8 **10**

이어서 큰 값과 작은 값을 바꾸어 다음과 같이 바뀝니다.

1 (5) 2 3 6 4 7 8 **10**

이어서 큰 값과 작은 값을 바꾸어 다음과 같이 바뀝니다.

1 (5) 2 3 4 6 7 8 **10**

이제 큰 값과 작은 값을 선택하면 각각 6과 4인데 작은 값의 인덱스가 큰 값의 인덱스보다 작으므로 피벗 값과 작은 값을 교환합니다. 따라서 다음과 같습니다.

1 4 2 3 **5** 6 7 8 **10**

1 (4) 2 3 **5** (6) 7 8 **10**

위와 같이 퀵 정렬을 순환적으로 수행하면 반으로 쪼개 들어가면서 분할 정복식으로 정렬이 완료됩니다. 이게 왜 빠른지 이해가 안 가시는 분은 직관적으로 다음과 같이 생각해보세요. 10 * 10은 100이지만 이를 전부 1개씩 나누게 되면 1 * 1을 10번 더한 값인 10에 불과합니다.

1 2 3 4 5 6 7 8 9 10

11. 계수 정렬(Counting Sort)

```
#include <stdio.h>
```

```c
int number = 10;
int data[] = {1, 10, 5, 8, 7, 6, 4, 3, 2, 9};

void show() {
   int i;
   for(i = 0; i < number; i++) {
      printf("%d ", data[i]);
   }
}

void quickSort(int* data, int start, int end) {
   if (start >= end) { // 원소가 1개인 경우 그대로 두기
      return;
   }

   int key = start; // 키는 첫 번째 원소
   int i = start + 1, j = end, temp;

   while(i <= j) { // 엇갈릴 때까지 반복
      while(i <= end && data[i] <= data[key]) { // 키 값보다 큰 값을 만날 때까지
         i++;
      }
      while(j > start && data[j] >= data[key]) { // 키 값보다 작은 값을 만날 때까지
         j--;
      }
      if(i > j) { // 현재 엇갈린 상태면 키 값과 교체
         temp = data[j];
         data[j] = data[key];
         data[key] = temp;
      } else { // 엇갈리지 않았다면 i와 j를 교체
         temp = data[i];
         data[i] = data[j];
         data[j] = temp;
      }
   }

   quickSort(data, start, j - 1);
   quickSort(data, j + 1, end);
}

int main(void) {
   quickSort(data, 0, number - 1);
   show();
   return 0;
}
```

지금까지는 모든 데이터를 그 자체로 위치를 바꾸어가면서 정렬하는 알고리즘에 대해서 공부를 했습니다. 이번에 다룰 계수 정렬은 '크기를 기준'으로 갯수만 세주면 되기 때문에 위치를 바꿀 필요가 없습니

다. 또한 모든 데이터에 한 번씩만 접근하면 된다는 점에서 무척이나 효율적입니다. 바로 해봅시다.

0. 초기 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
0	0	0	0	0

1. 1번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
1	0	0	0	0

2. 2번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
1	0	1	0	0

3. 3번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
1	1	1	0	0

4. 4번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
1	1	1	1	0

5. 5번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5

1	1	2	1	0

6. 6번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
1	2	2	1	0

7. 7번째 상태

1 3 2 4 3 2 5 3 1 2 3 4 4 3 5 1 2 3 5 2 3 1 4 3 5 1 2 1 1 1

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
1	2	2	1	1

바로 위와 같이 해당 크기의 원소를 만나는 경우 숫자를 1씩 더해가면 됩니다. 위와 같은 방식을 반복하면 결과적으로 다음과 같이 됩니다.

크기 = 1	크기 = 2	크기 = 3	크기 = 4	크기 = 5
8	6	8	4	4

이제 크기 1부터 5까지 해당 숫자만큼 출력하면 됩니다. 즉 1을 8번 출력하고, 2를 6번 출력하고, 3을 8번 출력하고, 4를 4번 출력하고, 5를 4번 출력하면 정렬이 이루어집니다.

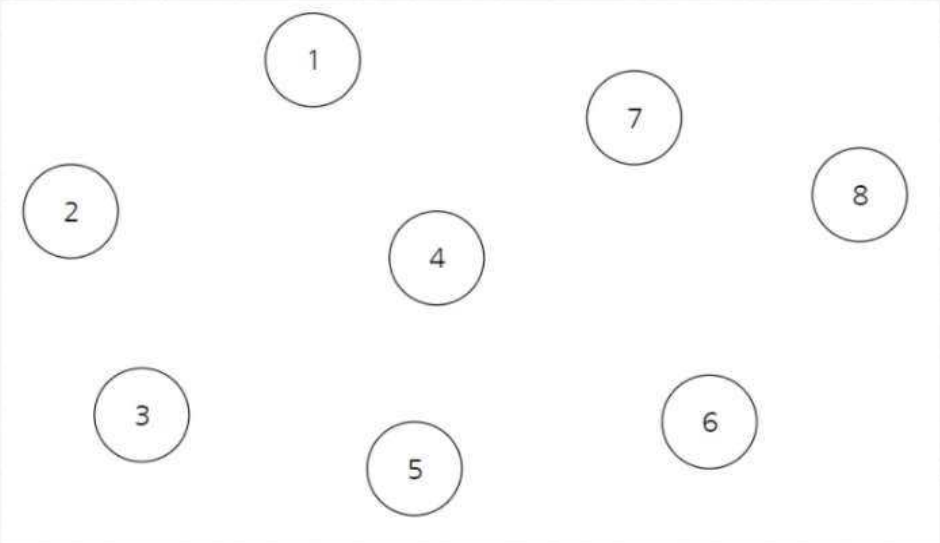
1 1 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 4 4 4 5 5 5 5

```c
#include <stdio.h>

int main(void)
{
    int temp;
    int count[6];
    int array[30] = {1, 3, 2, 4, 3, 2, 5, 3, 1, 2,
                     3, 4, 4, 3, 5, 1, 2, 3, 5, 2,
                     3, 1, 4, 3, 5, 1, 2, 1, 1, 1};
    for(int i = 1; i <= 5; i++)
    {
        count[i] = 0;
    }
    for(int i = 0; i < 30; i++)
```

```
    {
        count[array[i]]++;
    }
    for(int i = 1; i <= 5; i++)
    {
        if(count[i] != 0)
        {
            for(int j = 0; j < count[i]; j++)
                printf("%d ", i);
        }
    }
    return 0;
}
```

17. Union-Find(합집합 찾기)

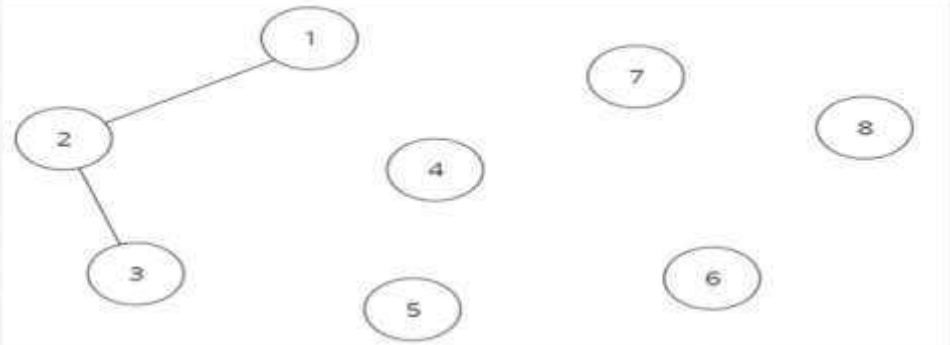
Union-Find(유니온-파인드)는 대표적인 그래프 알고리즘입니다. 바로 '합집합 찾기'라는 의미를 가진 알고리즘인데요. 서로소 집합(Disjoint-Set) 알고리즘이라고도 부릅니다. 구체적으로 여러 개의 노드가 존재할 때 두 개의 노드를 선택해서, 현재 이 두 노드가 서로 같은 그래프에 속하는지 판별하는 알고리즘입니다.



위와 같이 여러 개의 노드가 서로 자유분방하게 존재한다고 생각해봅시다. 이와 같이 모두 연결되지 않고 각자 자기 자신만을 집합의 원소로 가지고 있을 때를 다음과 같이 표현할 수 있습니다. 모든 값이 자기 자신을 가리키도록 만드는 겁니다. 아래 표에서 첫 번째 행은 '노드 번호'를 의미하고 두 번째 행은 '부모 노드 번호'를 의미합니다. 즉, 자신이 어떠한 부모에 포함되어 있는지를 의미합니다.

1	2	3	4	5	6	7	8

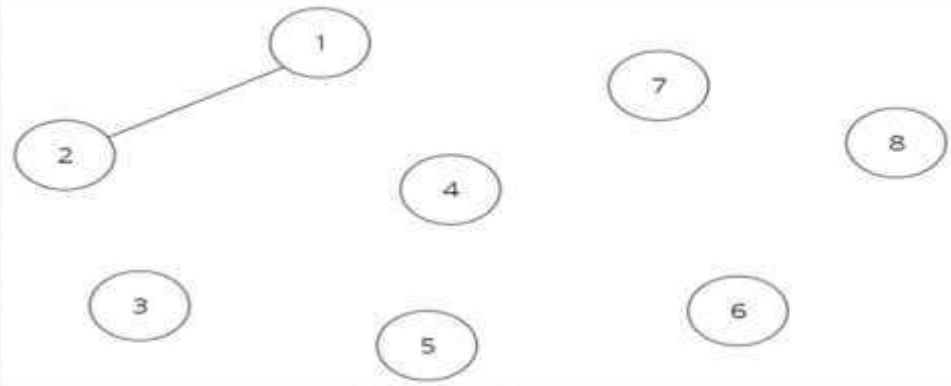
1	2	3	4	5	6	7	8



그렇다면 위와 같이 2와 3도 연결되었다면 어떻게 표현될까요?

1	2	3	4	5	6	7	8
1	1	2	4	5	6	7	8

위와 같이 표현됩니다. 다만, 여기에서 한 가지 의아한 점이 발견됩니다. 바로 '1과 3이 연결되었는지는 어떻게 파악할 수 있는지'입니다. 1과 3은 부모가 각각 1과 2로 다르기 때문에 '부모 노드'만 보고는 한 번에 파악할 수 없습니다. 그렇기 때문에 재귀 함수가 사용됩니다.



이 때 위와 같이 1과 2가 연결되었다고 해봅시다. 이러한 '연결성'을 우리가 프로그래밍 언어로 어떻게 표현할 수 있을 지에 대한 내용이 바로 Union-Find라고 생각하시면 이해가 쉽습니다.

1	2	3	4	5	6	7	8
1	1	3	4	5	6	7	8

3의 부모를 찾기 위해서는 먼저 3이 가리키고 있는 2를 찾는 겁니다. 그러면 2의 부모가 1을 가리키고 있으므로 그제서야 '결과적으로 3의 부모는 1이 되는 구나!'하고 파악할 수 있습니다. 이러한 과정은 재귀적으로 수행될 때 가장 효과적이고 직관적으로 언어를 작성할 수 있습니다. 그래서 결과적으로 합침(Union) 연산이 다 수행되고 나면 다음과 같이 표가 구성됩니다.

1	2	3	4	5	6	7	8
1	1	1	4	5	6	7	8

노드 1, 2, 3의 부모가 모두 1이기 때문에 이 세 가지 노드는 모두 같은 그래프에 속한다고 할 수 있습니다. 이것이 바로 Union-Find의 전부입니다. **Find 알고리즘**은 두 개의 노드의 부모 노드를 확인하여 현재 같은 집합에 속하는지 확인하는 알고리즘입니다.

```
int main(void) {
    int parent[11];
```

```
    for(int i = 1; i <= 10; i++) {
        parent[i] = i;
    }
    unionParent(parent, 1, 2);
    unionParent(parent, 2, 3);
    unionParent(parent, 3, 4);
    unionParent(parent, 5, 6);
```

```
C:₩Users₩나동빈₩Desktop₩실전 알고리즘 강좌₩UNION FIND.exe
1과 5는 연결되어있나요? 0
1과 5는 연결되어있나요? 1

--------------------------------
Process exited after 0.126 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

이러한 Union-Find 알고리즘은 다른 고급 그래프 알고리즘의 베이스가 된다는 점에서 반드시 익히고 들어가셔야 합니다. 다음 시간에는 이 Union-Find를 응용한 크루스칼 알고리즘(Kruskal Algorithm)에 대해 알아보도록 합시다.

```c
#include <stdio.h>

int getParent(int parent[], int x) {
    if(parent[x] == x) return x;
    return parent[x] = getParent(parent, parent[x]);
}

// 각 부모 노드를 합칩니다.
void unionParent(int parent[], int a, int b) {
    a = getParent(parent, a);
    b = getParent(parent, b);
    if(a < b) parent[b] = a;
    else parent[a] = b;
}

// 같은 부모 노드를 가지는지 확인합니다.
int findParent(int parent[], int a, int b) {
    a = getParent(parent, a);
    b = getParent(parent, b);
    if(a == b) return 1;
    else return 0;
}
int main(void) {
    int parent[11];
    for(int i = 1; i <= 10; i++) {
        parent[i] = i;
    }
    unionParent(parent, 1, 2);
    unionParent(parent, 2, 3);
    unionParent(parent, 3, 4);
    unionParent(parent, 5, 6);
    unionParent(parent, 6, 7);
    unionParent(parent, 7, 8);
```

```
    printf("1과 5는 연결되어있나요? %d\n", findParent(parent, 1, 5));
    unionParent(parent, 1, 5);
    printf("1과 5는 연결되어있나요? %d\n", findParent(parent, 1, 5));
}
```

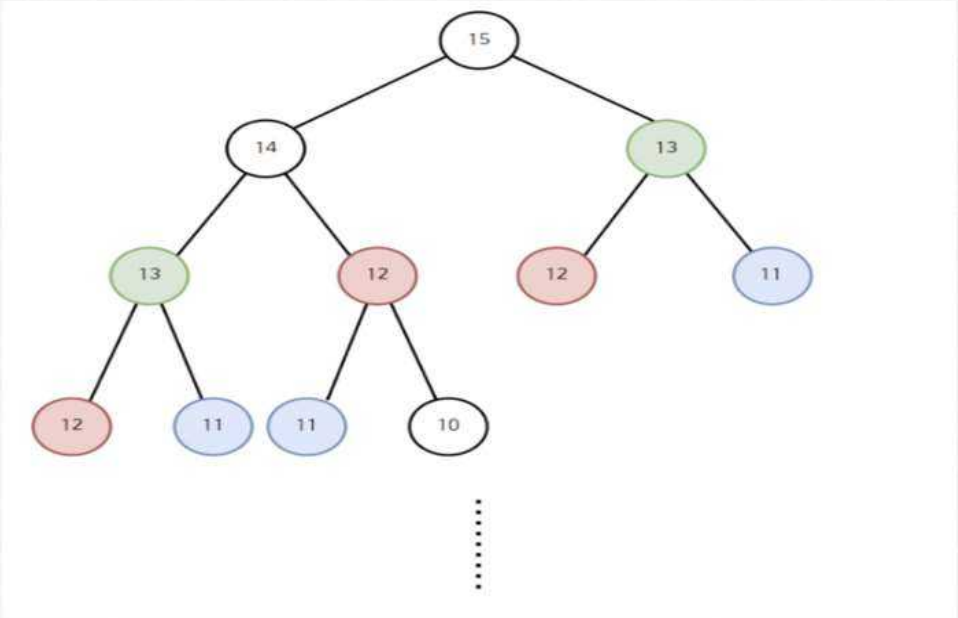
20. 다이나믹 프로그래밍(Dynamic Programming)

다이나믹 프로그래밍이란 하나의 문제를 단 한 번만 풀도록 하는 알고리즘입니다.

일반적으로 상당수 분할 정복 기법은 동일한 문제를 다시 푼다는 단점을 가지고 있습니다. (다만 분할 정복 기법은 '정렬'과 같은 몇몇 요소에 대해서는 동일한 문제를 다시 풀게 되는 단점이 없습니다. 그 예시로 퀵 정렬이나 병합 정렬은 매우 빠릅니다.) 단순 분할 정복으로 풀게 되면 심각한 비효율성을 낳는 대표적인 예시로는 피보나치 수열이 있습니다. **피보나치 수열**은 특정한 숫자를 구하기 위해 그 앞에 있는 숫자와 두 칸 앞에 있는 숫자의 합을 구해야 합니다. 바로 다음과 같습니다.

피보나치 수열의 점화식: D[i] = D[i - 1] + D[i - 2]

위 공식에 따라서 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... 와 같이 나아갈 수 있는 것입니다. 만약에 단순하게 분할 정복 기법을 이용해 15번째 피보나치 수열을 구한다고 가정해봅시다. 그러면 다음과 같이 그래프의 형태로 진행 상황을 확인할 수 있습니다. D[15]를 구하려면 D[14]와 D[13]을 알아야 합니다. 다만 D[14]를 알려면 D[13]과 D[12]를 알아야하네요. 이런 식으로 진행되는 방식입니다.



따라서 이런 경우에는 병합 정렬을 하실 때처럼 단순한 분할 정복 기법을 사용하시면 안 됩니다. 왜냐하면 이미 해결한 문제를 다시 반복적으로 해결하여 비효율적이기 때문입니다. 위 예시를 보시면 이 짧은 순간에 **D[12]가 벌써 3번이나 반복적으로 계산**되었기 때문입니다. (발간색으로 색상 처리를 했습니다.) 따라서 이럴 때는 대신에 동적 프로그래밍 기법을 사용해야 합니다.

다이나믹 프로그래밍은 다음의 가정 하에 사용할 수 있답니다.

1번 가정. 큰 문제를 작은 문제로 나눌 수 있습니다.
2번 가정. 작은 문제에서 구한 정답은 그것을 포함하는 큰 문제에서도 동일합니다.

즉, 쉽게 말해 크고 어려운 문제가 있으면 그것을 먼저 잘게 나누어서 해결한 뒤에 처리하여 나중에 전

체의 답을 구하는 것입니다. 다만 이 과정에서 '**메모이제이션(Memoization)**'이 사용된다는 점에 분할 정복과 다릅니다. 이미 계산한 결과는 배열에 저장함으로써 나중에 동일한 계산을 해야 할 때는 저장된 값을 단순히 반환하기만 하면 되는 것입니다. 바로 피보나치 수열을 예로 들어 문제를 확인해봅시다.

```
#include <stdio.h>

int d(int x) {
    if(x == 1) return 1;
    if(x == 2) return 1;
    return d(x - 1) + d(x - 2);
}

int main(void) {
    printf("%d", d(10));
}
```

위 경우는 답이 55로 잘 출력됩니다. 하지만 다음과 같이 50번째 피보나치 수열을 구하려고 한다면 어떻게 되는지 확인해봅시다.

```
#include <stdio.h>

int d(int x) {
    if(x == 1) return 1;
    if(x == 2) return 1;
    return d(x - 1) + d(x - 2);
}

int main(void) {
    printf("%d", d(50));
}
```

실행을 해보시면 아래와 같이 거의 우주가 멸망할 때까지 실행이 되지 않고 계속 CPU가 돌아가는 것을 볼 수 있습니다. 이는 사실 당연한 결과입니다. 피보나치 수열이 1개만 늘어나도 계산 량은 2배로 늘어나기 때문입니다. 간단하게 2의 50제곱이라고 보시면 됩니다. 1,000,000,000,000,000개가 넘는 계산량을 컴퓨터가 처리할 수 있을까요?

```
#include <stdio.h>

int d(int x) {
    if(x == 1) return 1;
    if(x == 2) return 1;
    return d(x - 1) + d(x - 2);
}
```

그래서 이를 해결하기 위해 메모이제이션 기법을 활용하시면 됩니다.

```
#include <stdio.h>

int d[100];

int fibonacci(int x) {
    if(x == 1) return 1;
    if(x == 2) return 1;
    if(d[x] != 0) return d[x];
    return d[x] = fibonacci(x - 1) + fibonacci(x - 2);
}

int main(void) {
    printf("%d", fibonacci(30));
}
```

위와 같이 해주시면 이미 계산된 결과는 배열 d에 저장되기 때문에 한 번 구한 값을 다시 구하는 일은 존재하지 않습니다. 위와 같이 작업하면 순식간에 계산된 결과가 출력되는 것을 확인할 수 있습니다. 다이나믹 프로그래밍의 개념을 이해하신 뒤에는 많은 문제를 접하여 감을 잡으시는 것이 중습니다.

21. 다이나믹 프로그래밍 타일링 문제 풀어보기

22. 에라토스테네스의 체

에라토스테네스의 체는 가장 대표적인 **소수(Prime Number) 판별 알고리즘**입니다. 소수란 '약의 약수를 두 개만 가지는 자연수'를 의미하며 2, 3, 5, 7, 11, ... 등이 존재합니다. 이러한 소수를 대량으로 빠르고 정확하게 구하는 방법이 에라토스 테네스의 체라고 할 수 있습니다. 일단 에라토스테네스의 체를 공부하기 전에 간단하게 소수를 판별하는 알고리즘을 작성해보도록 합시다.

```
#include <stdio.h>

bool isPrimeNumber(int x) {
    for(int i = 2; i < x; i++) {
        if(x % i == 0) return false;
    }
    return true;
}

int main(void)
{
    printf("%d", isPrimeNumber(97));
    return 0;
}
```

위와 같이 알고리즘을 작성하는 경우 소수 판별 알고리즘의 시간 복잡도는 **O(N)**입니다. 모든 경우의 수를 다 돌며 약수 여부를 확인한다는 점에서 몹시 비효율적입니다. 사실은 **O(N^(1/2))**로 손쉽게 계산할 수 있습니다. 왜냐하면 예를 들어 8의 경우 2 * 4 = 4 * 2와 같은 식으로 대칭을 이루기 때문입니다. 그러므로 특정한 숫자의 제곱근까지만 약수의 여부를 검증하시면 됩니다.

```
#include <stdio.h>
#include <math.h>

bool isPrimeNumber(int x) {
    int end = (int) sqrt(x);
    for(int i = 2; i <= end; i++) {
        if(x % i == 0) return false;
    }
    return true;
}

int main(void)
{
    printf("%d", isPrimeNumber(97));
    return 0;
}
```

다만 이렇게 한 두 개의 소수를 판별하는 것이 아니라 대량의 소수를 한꺼번에 판별하고자 할 때 사용하는 것이 바로 에라토스테네스의 체입니다. **에라토스테네스의 체**는 가장 먼저 소수를 판별할 범위만큼 배열을 할당에 그 인덱스에 해당하는 값을 넣어줍니다. 예를 들어 1부터 25까지 판별한다고 해봅시다.

① 이차원 배열을 생성하여 값을 초기화합니다.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

② 2부터 시작해서 특정 숫자의 배수에 해당하는 숫자들을 모두 지웁니다.

먼저 2의 배수를 지워봅시다. (자기 자신은 지우지 않습니다.)

1	2	3	~~4~~	5
~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15
~~16~~	17	~~18~~	19	~~20~~

21	~~22~~	23	~~24~~	25

이제 3의 배수를 지워봅시다. (자기 자신은 지우지 않습니다.)

1	2	3	~~4~~	5
~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~
~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25

③ 이미 지워진 숫자의 경우 건너뜁니다.

4는 이미 지워져있으므로 지우지 않고 건너 뜁니다. 이제 이러한 과정을 반복합시다.

1	2	3	~~4~~	5
~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~
~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~

결과적으로 위와 같이 모두 지워졌습니다.

④ 2부터 시작하여 남아있는 숫자들을 출력합니다.

1	2	3	~~4~~	5
~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~
~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~

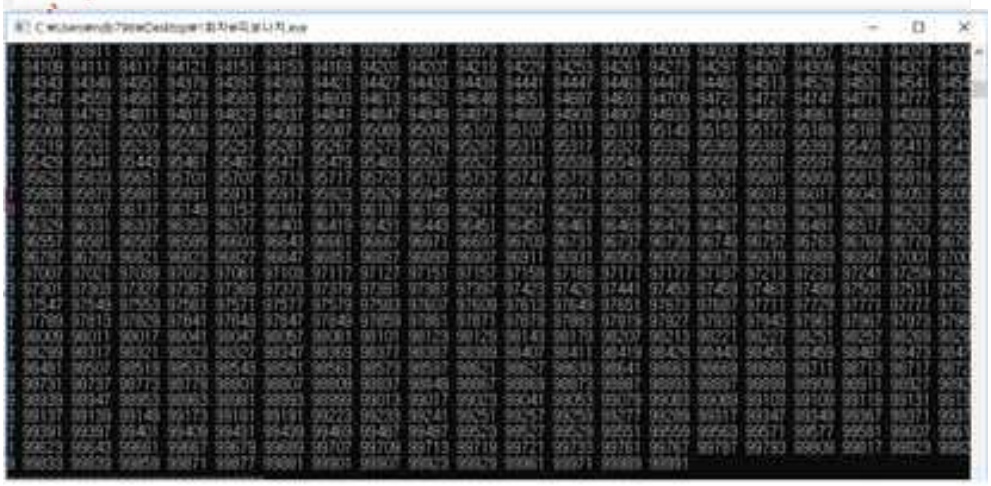
출력: 2 3 7 11 13 17 19 23

따라서 성공적으로 소수를 판별하게 되었습니다. 이를 이제 C언어 소스코드로 옮겨봅시다.

```
#include <stdio.h>

int number = 100000;
```

```
int a[100001];

void primeNumberSieve() {
    for(int i = 2; i <= number; i++) {
        a[i] = i;
    }
    for(int i = 2; i <= number; i++) {
        if(a[i] == 0) continue;
        for(int j = i + i; j <= number; j += i) {
            a[j] = 0;
        }
    }
    for(int i = 2; i <= number; i++) {
        if(a[i] != 0) printf("%d ", a[i]);
    }
}

int main(void) {
    primeNumberSieve();
}
```

```
#include <stdio.h>

int number = 100000;
int a[100001];

void primeNumberSieve() {
    for(int i = 2; i <= number; i++) {
        a[i] = i;
    }
    for(int i = 2; i <= number; i++) {
        if(a[i] == 0) continue;
        for(int j = i + i; j <= number; j += i) {
            a[j] = 0;
        }
    }
```



3. 버블정렬

내가 자주 사용하는 정렬방법 (걸리는시간 n^2)

```c
#include <stdio.h>

int main(void) {
    int i, j, temp;
    int array[10] = {1, 10, 5, 8, 7, 6, 4, 3, 2, 9};
    for(i = 0; i < 10; i++) {
        for(j = 0; j < 9 - i; j++) {
            if(array[j] > array[j + 1]) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    for(i = 0; i < 10; i++) {
        printf("%d ", array[i]);
    }
    return 0;
}
```

40. 이분정렬

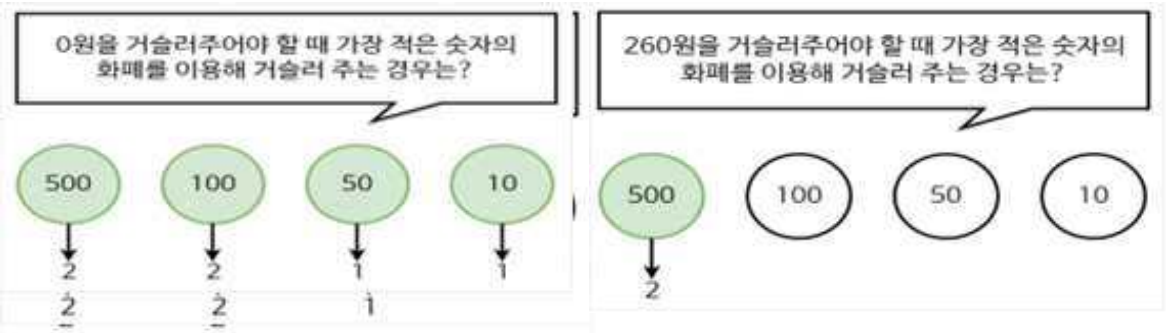
선택한 번호가 몇 번째에 있는지 찾는 정렬방법

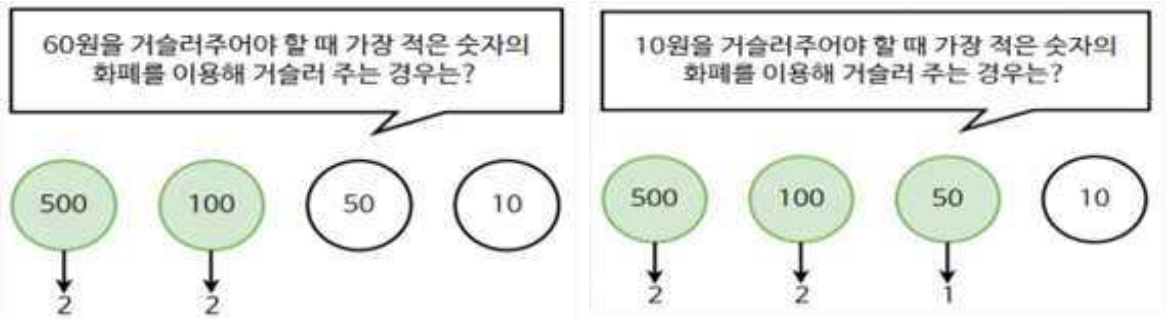
```c
#include <iostream>
#define NUMBER 10

using namespace std;

int a[] = {8, 3, 4, 5, 1, 9, 6, 7, 2, 0};
int data = 7;

int main(void) {
    for(int i = 0; i < NUMBER; i++) {
        if(a[i] == data) {
            cout << i + 1 << "번째에서 찾았습니다.";
        }
    }
}
```

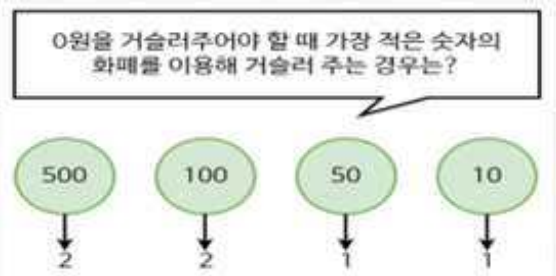
34. 그리디 알고리즘

돈을 거슬러줄 때 사용하는 알고리즘

```cpp
#include <iostream>

using namespace std;

int main(void) {
    int n, result = 0;
    cin >> n;
    result += n / 500;
    n %= 500;
    result += n / 100;
    n %= 100;
    result += n / 50;
    n %= 50;
    result += n / 10;
    cout << result;
    return 0;
}
```







22. 에라토스테네스의 체

소수 판별 알고리즘

```c
#include <stdio.h>

bool isPrimeNumber(int x) {
    for(int i = 2; i < x; i++) {
        if(x % i == 0) return false;
    }
    return true;
}

int main(void)
{
    printf("%d", isPrimeNumber(97));
    return 0;
}
```

```c
#include <stdio.h>
#include <math.h>

bool isPrimeNumber(int x) {
    int end = (int) sqrt(x);
    for(int i = 2; i <= end; i++) {
        if(x % i == 0) return false;
    }
    return true;
}

int main(void)
{
    printf("%d", isPrimeNumber(97));
    return 0;
}
```

스택

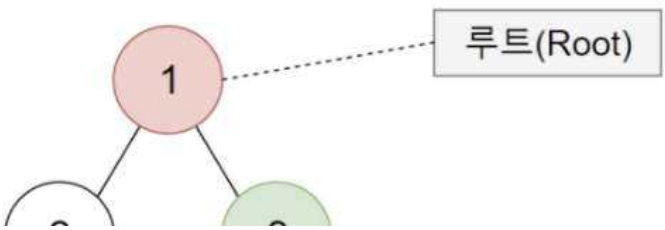
〈 명령어 〉
삽입(7)
삽입(5)
삽입(4)
삭제()
삽입(6)
삭제()

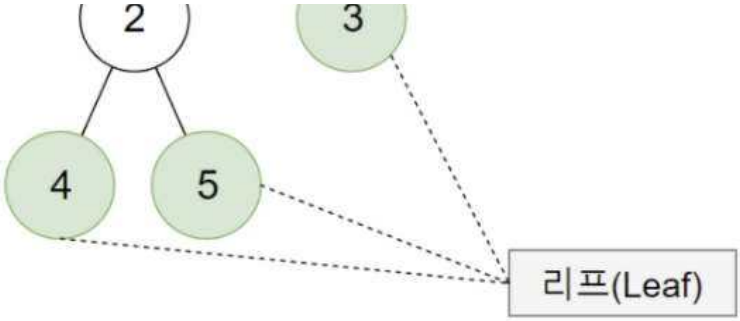
```
#include <iostream>
#include <stack>

using namespace std;

int main(void) {
    stack<int> s;
    s.push(7);
    s.push(5);
    s.push(4);
    s.pop();
    s.push(6);
    s.pop();
    while(!s.empty()) {
        cout << s.top() << ' ';
        s.pop();
    }
    return 0;
```

10. 힙 정렬(Heap Sort)

힙 정렬(Heap Sort)은 병합 정렬(Merge Sort)와 퀵 정렬(Quick Sort)만큼 빠른 정렬 알고리즘입니다. 또한 실제로 고급 프로그래밍 기법으로 갈 수록 힙(Heap)의 개념이 자주 등장하기 때문에 반드시 알고 넘어가야 할 정렬 알고리즘이기도 합니다. **힙 정렬**은 힙 트리 구조(Heap Tree Structure)를 이용하는 정렬 방법입니다. 즉 정렬의 기초 아이디어는 다음과 같습니다.

이진 트리: 모든 노드의 자식 노드가 2개 이하인 트리 구조



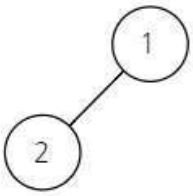


흔히 위와 같은 구조를 이진 트리라고 합니다. 여기서 **트리**(Tree)라는 것은 말 그대로 가지를 뻗어나가는 것처럼 데이터가 서로 연결되어있다는 것입니다. 트리는 그 형태에 따라서 종류가 굉장히 다양한데 우리는 일단 자식 노드가 2개 이하인 이진 트리에 대해서만 알면 됩니다. 위와 같이 이진 트리가 어떻게 생겼는지 알아보았으니 이제 **완전 이진 트리**(Complete Binary Tree)에 대해 알아보도록 합시다.

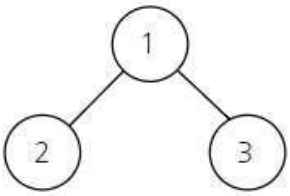
완전 이진 트리는 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차근차근 들어가는 구조의 이진 트리입니다. 즉 완전 이진 트리는 이진 트리의 노드가 중간에 비어있지 않고 빽빽히 가득 찬 구조입니다. 예를 들어 데이터가 1부터 7까지 총 7개라면 이것을 트리에 삽입하면 다음과 같은 순서로 들어갑니다. 항상 왼쪽 자식 노드부터 데이터가 들어갑니다.

1

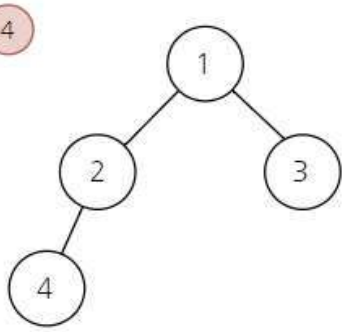
2



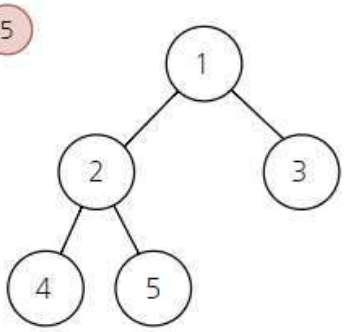
3



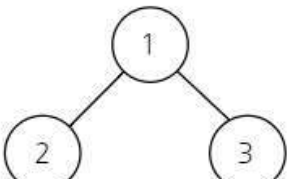
4



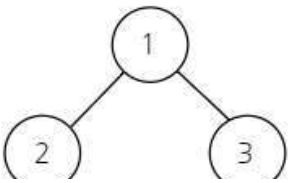
5



6

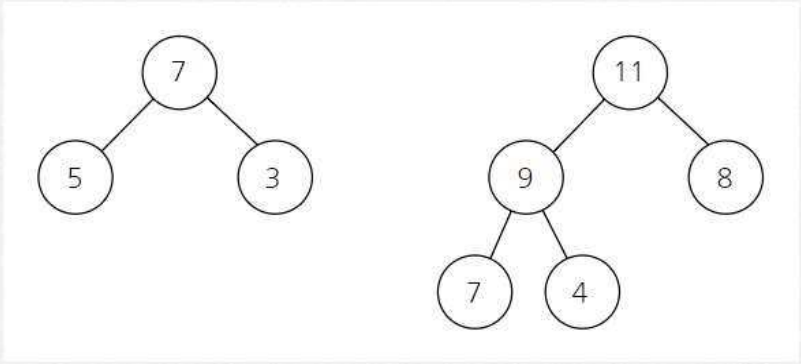


7

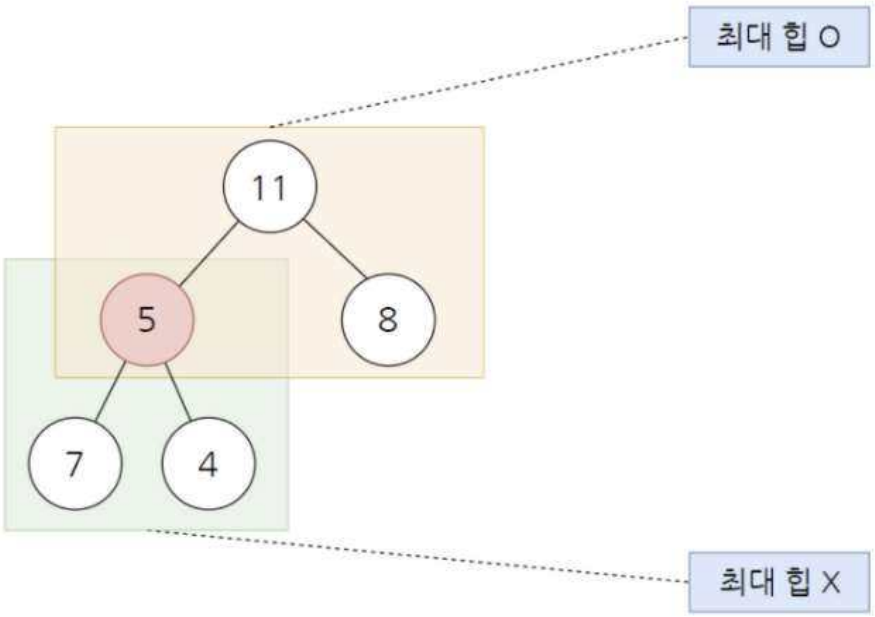


이제 완전 이진 트리에 대해서도 알아보았으므로 **힙(Heap)**에 대해 알아보도록 합시다. 힙은 최솟값이나 최댓값을 빠르게 찾아내기 위해 완전 이진 트리를 기반으로 하는 트리입니다. 힙에는 최대 힙과 최소 힙이 존재하는데 **최대 힙**은 '부모 노드'가 '자식 노드'보다 큰 힙이라고 할 수 있습니다. 일단 힙 정렬을 하기 위해서는 정해진 데이터를 힙 구조를 가지도록 만들어야 합니다.

힙이 어떻게 생겼는지 알기 위해서 힙의 예제들을 살펴봅시다. 최대 힙은 부모 노드의 값이 자식 노드보다 커야 합니다. 아래와 같은 것들이 전부 최대 힙이라고 할 수 있습니다.

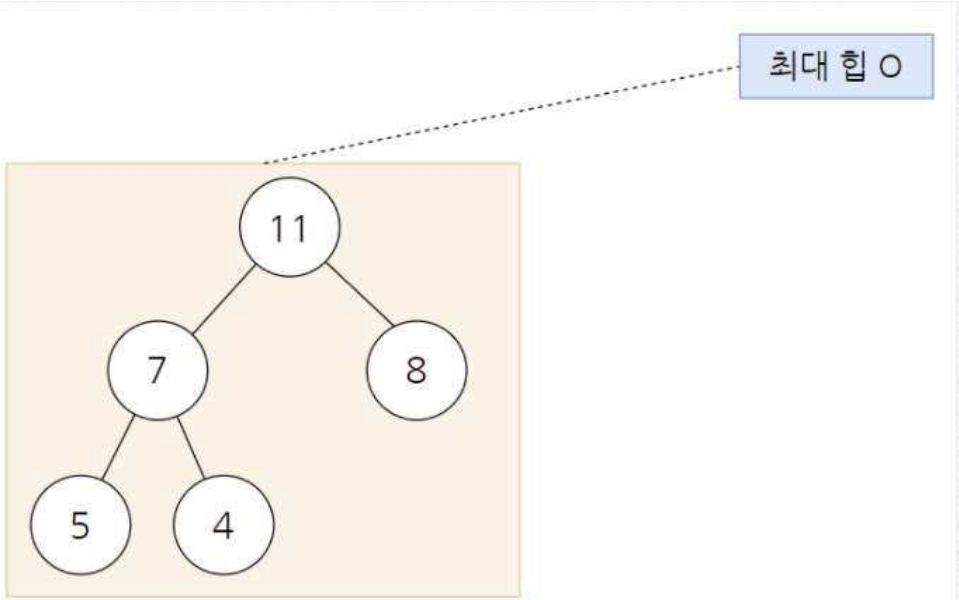


다만 트리(Tree) 안에서 특정 노드 때문에 최대 힙이 붕괴되는 경우가 있습니다. 아래와 같이요. 전체로 보면 중간에 있는 데이터 5를 가지는 노드 때문에 최대 힙이 아니지만 그 위 쪽으로 보았을 때는 최대 힙이 형성되고, 아래쪽으로 보았을 때는 최대 힙이 형성되지 않는 경우가 만들어질 수 있습니다.



힙 정렬을 수행하기 위해서는 **힙 생성 알고리즘(Heapify Algorithm)**을 사용합니다. 힙 생성 알고리즘은 특정한 '하나의 노드'에 대해서 수행하는 것입니다. 또한 해당 '하나의 노드를 제외하고는 최대 힙이 구성되어 있는 상태'라고 가정을 한다는 특징이 있습니다. 위의 그림이 정확히 해당 가정에 부합합니다. 위 트리에서 5만 최대 힙 정렬을 수행해주면 전체 트리가 최대 힙 구조로 형성되는 상태입니다.

힙 생성 알고리즘(Heapify Algorithm)은 특정한 노드의 두 자식 중에서 더 큰 자식과 자신의 위치를 바꾸는 알고리즘입니다. 또한 위치를 바꾼 뒤에도 여전히 자식이 존재하는 경우 또 자식 중에서 더 큰 자식과 자신의 위치를 바꾸어야 합니다. 자식이 더이상 존재하지 않을 때 까지요. 즉 위 예시에서는 5의 두 자식인 7과 4 중에서 더 큰 자식인 7과 5의 위치를 바꾸어주면 됩니다. 바꾼 결과는 아래와 같습니다.



위와 같이 힙 생성 알고리즘은 전체 트리를 힙 구조를 가지도록 만든다는 점에서 굉장히 중요한 알고리즘입니다. 이러한 힙 생성 알고리즘의 시간 복잡도는 몇 일까요? 한 번 자식 노드로 내려갈 때마다 노드의 갯수가 2배씩 증가한다는 점에서 O(log N)입니다. 예를 들어 데이터의 갯수가 1024개라면 대략 10번 정도만 내려가도 된다는 거에요. 이제 한 번 실제 힙 정렬 과정을 수행해보도록 합시다.

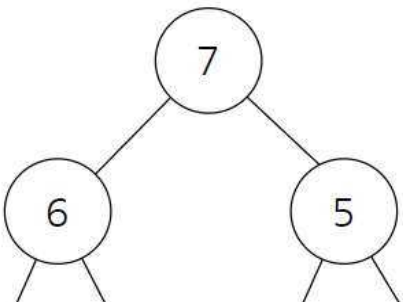
다음의 데이터를 오름차순 정렬하세요.

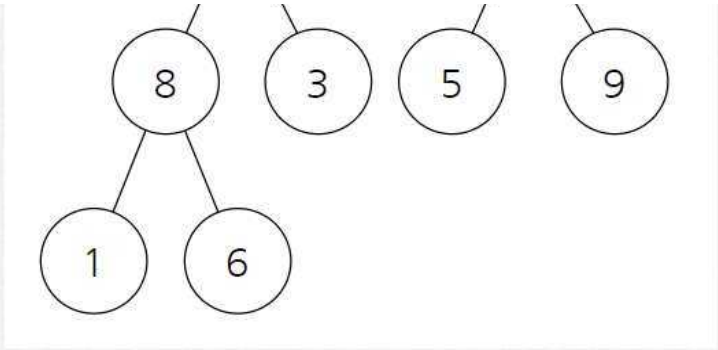
7 6 5 8 3 5 9 1 6

기본적으로 완전 이진 트리를 표현하는 가장 쉬운 방법은 배열에 그대로 삽입하는 겁니다. 현재 정렬할 데이터의 갯수가 9개이기 때문에 인덱스 0부터 8까지 차례대로 담아주는 것입니다.

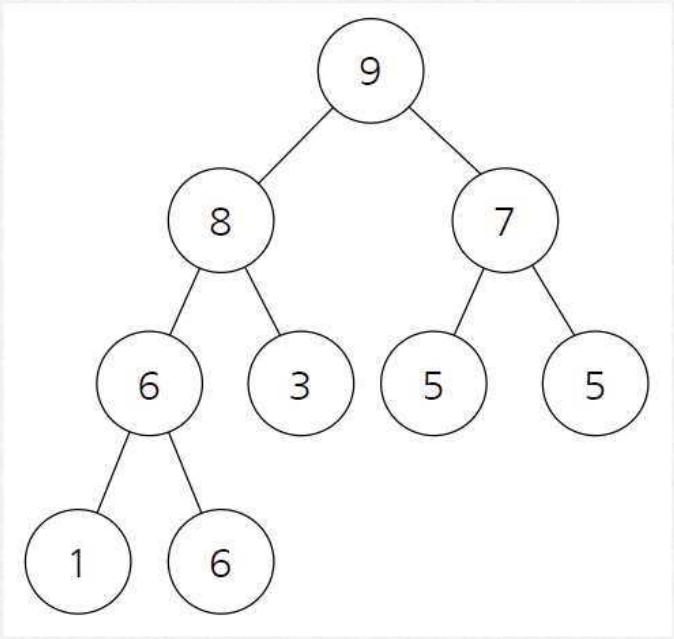
0	1	2	3	4	5	6	7	8
7	6	5	8	3	5	9	1	6

다시 말해서 완전 이진 트리에 삽입이 되는 순서대로 인덱스를 붙여주는 겁니다. 위 배열을 완전 이진 트리 형태로 출력하면 다음과 같습니다.

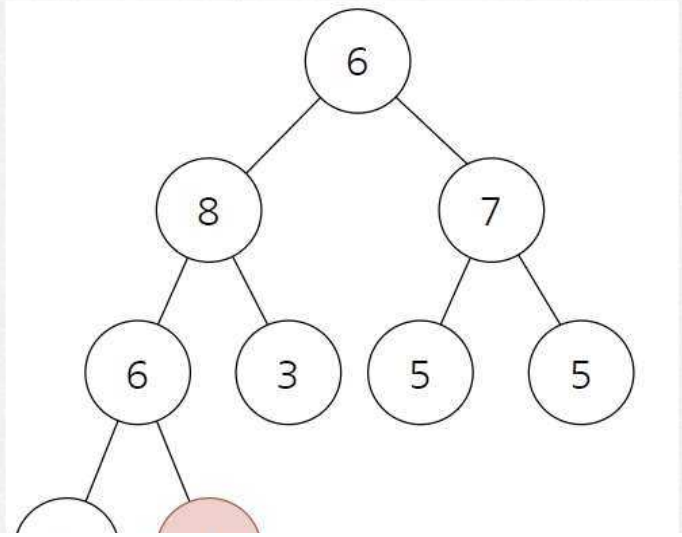




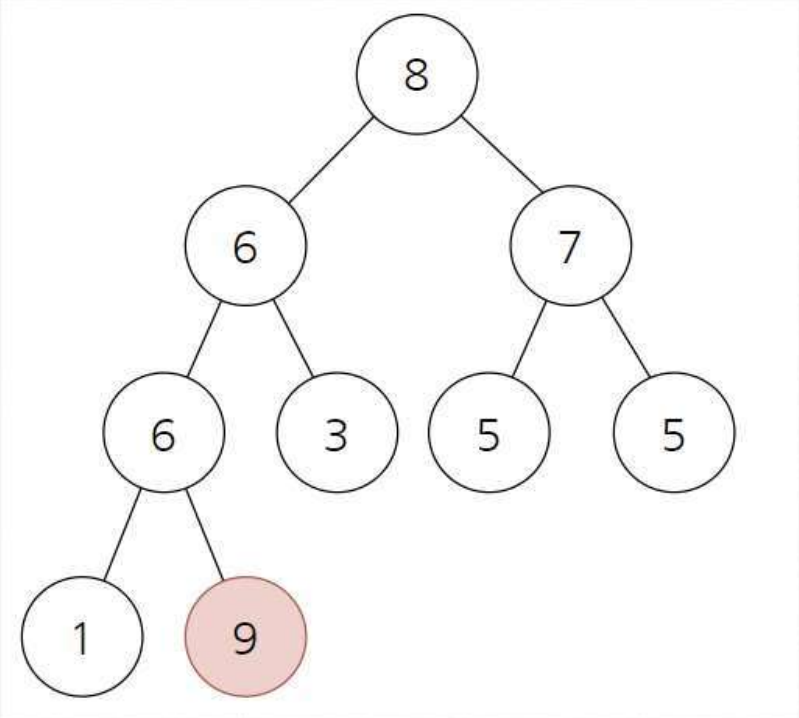
말 그대로 배열에 있는 인덱스가 그대로 차례대로 트리로 표현된 것입니다. 완전 이진 트리를 배열로 표현하고, 배열을 다시 완전 이진 트리로 표현할 수 있으셔야 합니다. 위와 같은 상황에서 모든 원소를 기준으로 힙 생성 알고리즘을 적용하셔서 전체 트리를 **힙 구조**로 만들어주시면 됩니다. 이 때 데이터의 갯수가 N개 이므로 전체 트리를 힙 구조로 만드는 복잡도는 O(N * log N)입니다. (사실상 모든 데이터를 기준으로 힙 생성 알고리즘을 쓰지 않아도 되기 때문에 **O(N)**에 가까운 속도를 낼 수 있습니다.)



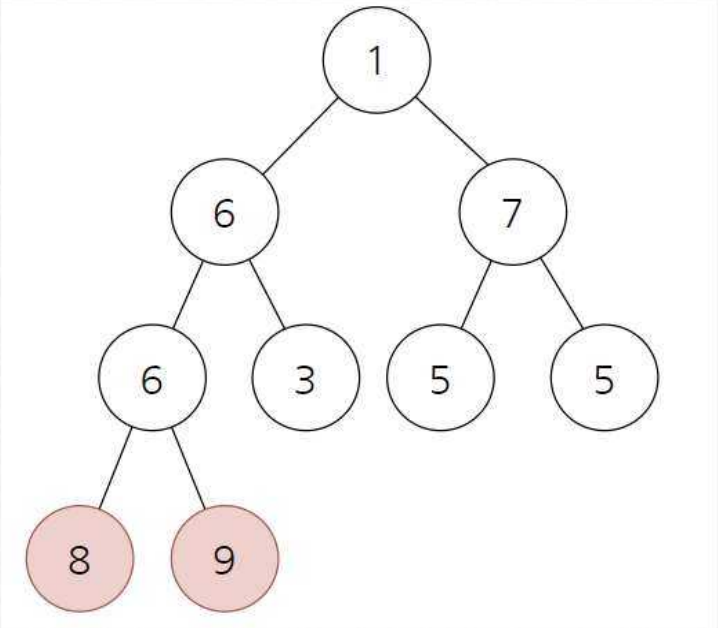
그래서 결과적으로는 위와 같이 최대 힙이 구성됩니다. 고작 **O(N * logN)**으로 위와 같이 만들 수 있는 겁니다. 이제부터 실제로 우리가 원하던 정렬을 직관적으로 수행할 수 있습니다. 루트(Root)에 있는 값을 가장 뒤쪽으로 보내면서 힙 트리의 크기를 1씩 빼주는 겁니다.



즉 위와 같이 9와 6을 바꾼 뒤에 9는 정렬이 완료된 것이므로 빨간색으로 처리합니다. 이제 9를 제외하고 나머지 8개 원소를 기준으로 또 힙 생성 알고리즘(Heapify)를 수행합니다. 결과는 다음과 같습니다.



이제 다시 가장 큰 숫자인 8이 루트에 존재합니다. 이것을 가장 뒤 쪽의 원소와 서로 바꿉니다.



그럼 위와 같이 8과 9가 가장 뒤에 배열되어 정렬이 이루어졌습니다. 이제 이 과정을 반복하시면 됩니다. 힙 생성 알고리즘의 시간 복잡도는 O(log N)이고 전체 데이터의 갯수가 N개이므로 시간 복잡도는 O(N * log N)이라고 할 수 있습니다. 또한 아까전에 계산했던 맨 처음에 힙 구조를 만드는 복잡도는 O(N * log N)이었습니다. 한 마디로 전체 **힙 정렬의 전체 시간 복잡도**는 O(N * log N)입니다.

```c
#include <stdio.h>

int number = 9;
int heap[9] = {7, 6, 5, 8, 3, 5, 9, 1, 6};

int main(void) {
    // 힙을 구성
    for(int i = 1; i < number; i++) {
        int c = i;
        do {
            int root = (c - 1) / 2;
            if(heap[root] < heap[c]) {
                int temp = heap[root];
                heap[root] = heap[c];
                heap[c] = temp;
            }
            c = root;
        } while (c != 0);
    }
    // 크기를 줄여가며 반복적으로 힙을 구성
    for (int i = number - 1; i >= 0; i--) {
        int temp = heap[0];
        heap[0] = heap[i];
        heap[i] = temp;
        int root = 0;
        int c = 1;
         do {
            c = 2 * root + 1;
            // 자식 중에 더 큰 값을 찾기
            if(c < i - 1 && heap[c] < heap[c + 1]) {
                c++;
            }
            // 루트보다 자식이 크다면 교환
            if(c < i && heap[root] < heap[c]) {
                temp = heap[root];
                heap[root] = heap[c];
                heap[c] = temp;
            }
            root = c;
         } while (c < i);
     }
     // 결과 출력
     for(int i = 0; i < number; i++) {
         printf("%d ", heap[i]);
     }
 }
```

보충 자료

최대 힙을 활용한 오름차순 정렬에서 힙 생성 함수(Heapify)는 특정한 노드를 기준으로 위쪽으로 올라가는 **상향식 구현 방식**과 아래쪽으로 내려가는 **하향식 구현 방식**이 있습니다. 두 방식 모두 시간 복잡도는 동일하다는 특징이 있습니다. 위 본문에서는 상향식 구현 방식을 보여주었는데 하향식 구현 방식도 스스로 고민해서 작성해보는 연습을 해보시는 것을 추천합니다.

〈 예시 입력 〉

10
10 26 5 37 1 61 11 59 15 48 19

위와 같이 10개의 원소가 들어왔을 때 다음과 같은 정렬 과정이 출력되도록 하는 프로그램을 작성하세요. 이때 힙 생성을 함에 있어서 **하향식**으로 구현을 하셔야 정상적으로 답이 출력됩니다.

〈 예시 출력 〉

61 48 59 26 19 11 37 15 1 5
59 48 37 26 19 11 5 15 1 61
48 26 37 15 19 11 5 1 59 61
37 26 11 15 19 1 5 48 59 61
26 19 11 15 5 1 37 48 59 61
19 15 11 1 5 26 37 48 59 61
15 5 11 1 19 26 37 48 59 61
11 5 1 15 19 26 37 48 59 61
5 1 11 15 19 26 37 48 59 61
1 5 11 15 19 26 37 48 59 61

〈 정답 소스코드 〉

```
#include <iostream>
#include <vector>

using namespace std;

int number;
int heap[1000001];

void heapify(int i) {
    // 왼쪽 자식 노드를 가리킵니다.
    int c = 2 * i + 1;
    // 오른쪽 자식 노드가 있고, 왼쪽 자식노드보다 크다면
    if(c < number && heap[c] < heap[c + 1]) {
        c++;
    }
    // 부모보다 자식이 더 크다면 위치를 교환합니다.
    if(heap[i] < heap[c]) {
        int temp = heap[i];
        heap[i] = heap[c];
```

```
        heap[c] = temp;
    }
    // number / 2까지만 수행하면 됩니다.
    if(c <= number / 2) heapify(c);
}
```