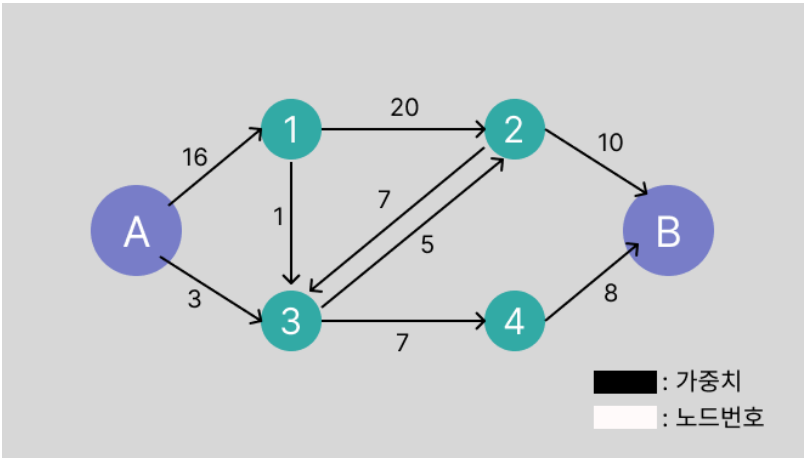
#algorithms #study

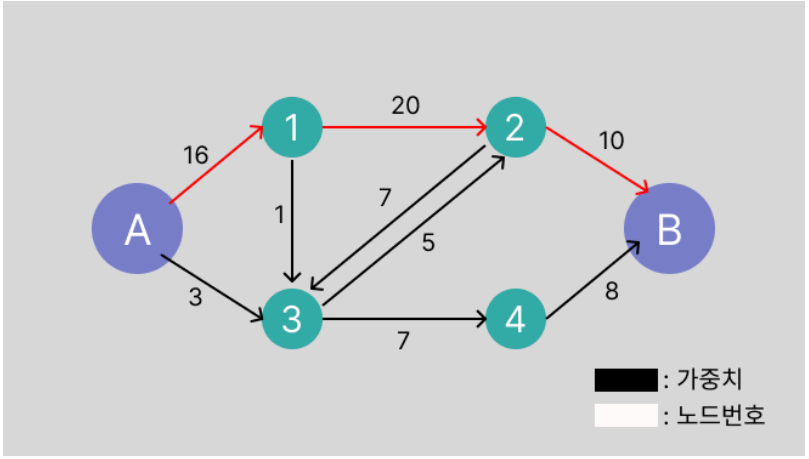
네트워크 유량

네트워크 유량이란?

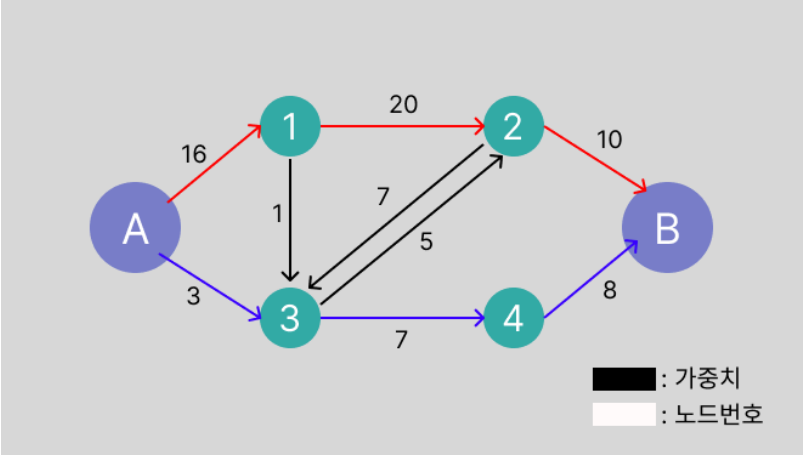
- 문제 하나를 생각해 보자

> 철수는 교통조사관이다. 이번에 서기관으로 괴팍한 상사가 한 명 부임하더니 철수에게 A 지점에서 B 지점까지 시간 당 가장 많은 차량 이동이 가능하게끔 도로 통제 계획을 짜라는 지시가 내려왔다. 철수는 까라면 까라는 상관의 지시에 한숨을 쉬고 본인이 맡은 도로 지도를 보았다.

- 

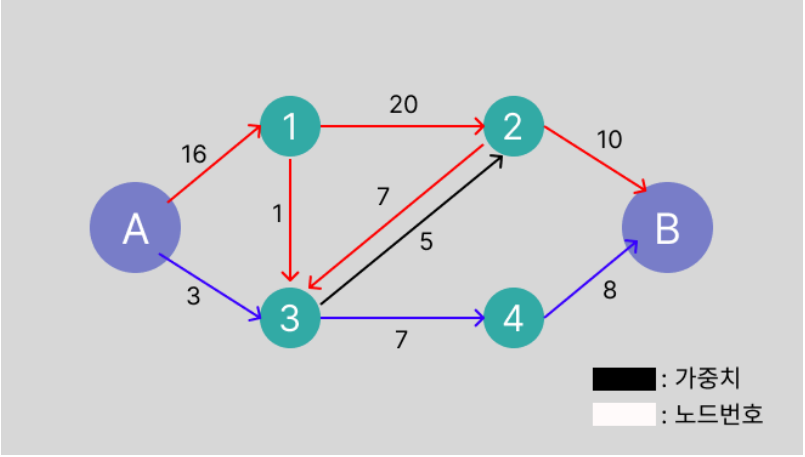
- 가중치는 각 도로를 지나갈 수 있는 차량 대수이며 A->B까지 걸리는 시간은 경로를 뭘 하든 모두 동일하다. 이럴 때, 만약 철수가 A -> 1 -> 2 -> B 경로만 택한다고 가정하자
  - 

  - A -> 1로 갈 때는 16대의 차량이 지나갈 수 있고 1 -> 2로 갈 때는 16대의 차량이 그대로 이동할 것이다. 하지만 2 -> B로 갈 때 10대의 차량 밖에 지나가지 못하니 결국 A -> B로 한 번 이동할 때 도착하는 차량은 10대다.
- 어차피 A -> B로 보냈을 때 도착하는 대수가 많으면 그만큼 교통 분배를 성공적으로 한 것이다. 철수는 여기서 아이디어를 얻어 여러 경로로 동시에 보내는 걸 고려했다.



- A -> 1 -> 2 -> B 경로는 여전히 10대가 통과한다. 그리고 추가로 설정한 A -> 3 -> 4 -> B는 마지막 경로만 보면 8대가 더 통과할 수 있지만 처음에 통과한 3대 말고 추가 유입이 없으므로 결국 3대가 통과한다. 즉, 10 + 3 = 13 대다.

- 철수는 A -> 3 -> 4 -> B 경로가 한산하니 이쪽으로 좀 더 진입하는 길목으로 유도하면 좀 더 많은 교통량이 목적지까지 도착할 수 있지 않을까 생각했다.



- A -> 1 -> 2 -> B 경로에서 각각 1에 도착했을 때 3으로 1대 보내고 2에 도착했을 때 3대를 보내면 3 -> 4로 이동했을 때 최대 7대를 보낼 수 있다. 그렇게 하면 4 -> B로 갈 때, 7대가 이동하므로 총 10 + 7 = 17대를 보낼 수 있게 되었다.

- 위의 문제와 같이 시작 지점에서 끝 지점으로 내용물을 보낼 때, 끝 지점에 최대로 보낼 수 있는 용량을 구하는 알고리즘이 네트워크 유량 알고리즘이다.

네트워크 유량의 개념과 용어

- 네트워크 유량 : 각 간선에 용량이라는 추가 속성이 존재하는 방향 그래프
- 용어
  - source : 시작점
  - sink : 도착점
  - 유량 : 용량이 각 간선에서 통과할 수 있는 최대치라면 유량은 현재 그 간선에서 통과하고 있는 크기를 말한다.
  - c(u, v) : 정점 u에서 정점 v로 가는 간선의 용량
  - f(u, v) : 정점 u에서 정점 v로 이동하는 실제 용량
  - r(u, v) : c(u, v) - f(u, v)
- 네트워크 유량의 속성

- 용량 제한 속성 : 각 간선의 유량은 해당 간선의 용량을 초과할 수 없다

$$f(u, v) <= c(u, v)$$

- 유량의 대칭성 : u에서 v로 유량이 흘러올 경우, v에서는 유량이 음수로 들어오는 것과 같다

$$f(u, v) = -f(v, u)$$

- 유량의 보존 : 각 정점에 들어오는 유량과 나가는 유량의 양이 정확히 같아야 한다

$$\sum_{v \in V} f(u, v) = 0$$

포드-풀커슨 알고리즘

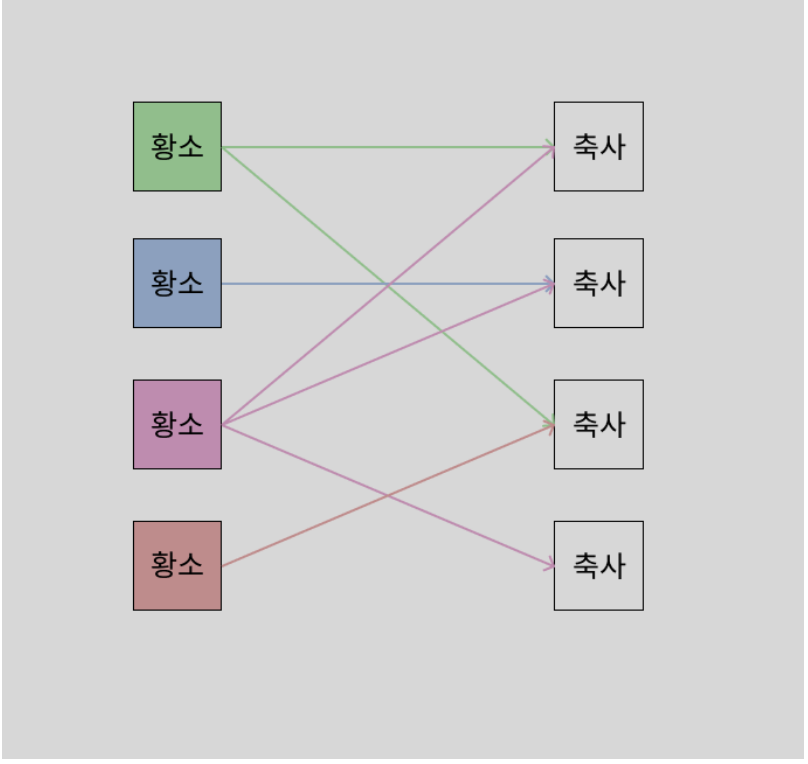
- 네트워크 유량의 모든 유량을 0으로 만들고 소스에서 싱크로 유량을 더 보낼 수 있는 경로를 찾아 유량을 보내는 걸 반복한다. 그렇게 유량을 보내는 경로를 찾고 찾아 누적하여 가장 많이 보낼 수 있는 유량 값을 반환한다.
- 근데, 내가 선점한 경로 때문에 최대 유량 값을 찾지 못하면? 이를 해결하기 위해서 유량의 대칭성이 필요하다. 즉 역방향은 음수로 두어 해당 간선의 유량을 상쇄시키는 것이다.

```
// MAX_V는 정점의 개수로 이미 변수로 저장되어 있다고 가정, INF는 int형 최대값이다.
// capacity는 각 간선의 용량, flow는 유량이다
int capacity[MAX_V][MAX_V];
int flow[MAX_V][MAX_V];
int V = MAX_V;

// flow를 계산하고 총 유량을 반환한다
int networkFlow(int source, int sink) {
    int totalFlow = 0;
    while (true) {
        // 경로 탐색용
        vector<int> parent(MAX_V, -1);
        queue<int> q;
        // 시작점
        parent[source] = source;
        q.push(source);
        // 증가 경로를 만든다. 즉, source -> sink로 현재 유량을 보낼 수 있는 경로를 찾는
것
        while (!q.empty() && parent[sink] == -1) {
            int here = q.front();
            q.pop();
            for (int there = 0; there < V; there++) {
                if (capacity[here][there] - flow[here][there] > 0 &&
parent[there] == -1)
                    q.push(there);
                    parent[there] = here;
            }
        }
```

```
        // 증가 경로가 없으면 종료
        if (parent[sink] == -1)
            break;
        // 증가 경로를 통해 유량을 얼마나 보낼 지 결정
        int amount = INF;
        for (int p = sink; p != source; p = parent[p])
            amount = min(capacity[parent[p]][p] - flow[parent[p]][p],
amount);
        // 지나가는 경로에 전부 유량이 늘어나므로 반영한다
        for (int p = sink; p != source; p = parent[p]) {
            flow[parent[p]][p] += amount;
            flow[p][parent[p]] -= amount;
        }
        // 결정되어 sink까지 간 유량을 총합계 변수에 더해준다
        totalFlow += amount;
    }
    // 총합계 변수를 반환한다
    return totalFlow;
}
```

이분 매칭

- 이분 매칭은 A그룹과 B그룹이 있을 때, A 그룹을 최대한 B그룹과 연결시키는 것이다.
- 문제를 보면서 이해하자.
  - 

  - 마음씨 좋은 주인이 황소에게 축사를 고를 권리를 주었다. 그러자 각 황소들은 자신들이 선호하는 축사를 그림과 같이 화살표로 표시했다. 이 때, 축사는 황소 1마리 밖에 못 들어갈 때, 축사에 못 들어가는 황소의 수가 가장 적을 때 축사에 들어간 황소의 수를 구하시오.
  - 당연하지만 지금 상황을 보면 직관적으로 봐도 모든 황소가 전부 축사에 들어갈 수 있다. 하지만 축사가 하나 줄어들거나 선호도 문제로 축사에 아무도 못 들어가거나 하는 상황이 발생하면 결국

최대로 들어가는 황소 수를 알고리즘을 이용하여 구하는 수 밖에 없다.

- 이게 네트워크 유량과 무슨 상관인가? 가만히 생각해보자. 황소는 A그룹이고 축사는 B그룹이다. 즉, A-> B로 가는 방향 그래프이며 그룹이 두 부류로 나뉘어서 하나의 그룹으로 향하니 이분 매칭이다. 그럼 가상의 출발점이 출발점 -> 황소로 가고 가상의 도착점이 축사 -> 도착점으로 가면 어떻게 될까? 거기에 더해 황소의 선호도로 그은 간선이 황소 1마리 밖에 못 들어가는 용량이 1인 간선이라 하면? 네트워크 유량 문제가 된다.

풀어보자

- 최대 유량, https://www.acmicpc.net/problem/6086
- 축사 배정, https://www.acmicpc.net/problem/2188

게임 이론

게임 이론이란?

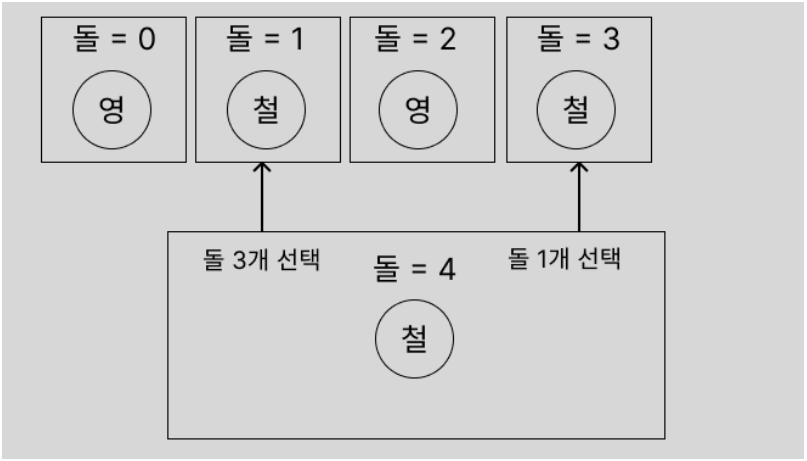
- 게임 이론은 상호 의존적이고 이상적인 의사결정을 다룬 수학적 이론이다. 알고리즘에서는 보통 게임의 참가자가 각자 최선의 선택을 하는 상황에서 나오는 결말을 묻는 경우가 많다.
- 알고리즘에서 게임 이론 문제는 보통 참가자 모두가 자신에게 있어 승리할 최선의 선택을 한다. 모두가 최선의 선택을 하는 상황에서 조건을 주고 승패 여부를 판정한다.
- 문제 하나를 생각해보자

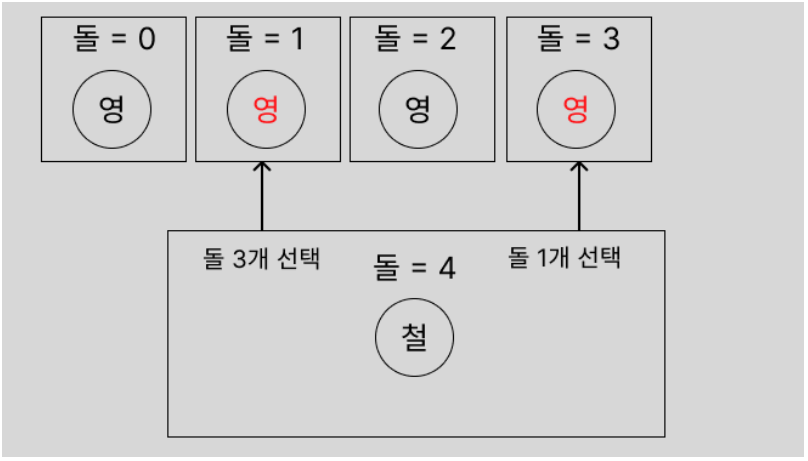
> 철수와 영희는 돌 가져오기 게임을 한다. 초기에 무작위로 돌무더기가 주어지고 그 돌무더기에서 차례를 번갈아 가며 돌을 1개 또는 3개 가져갈 수 있다. 마지막에 손에 쥘 수 있는 돌이 없는 자가 패배하며, 철수와 영희가 모두 자신이 이기기 위한 최선의 선택을 한다. 철수가 먼저 돌을 가져가고 영희가 그 다음에 돌을 가져가는 차례가 보장될 때, 돌무더기에 있는 돌 개수가 주어진 경우, 철수와 영희 중 누구가 이기는 지 판단하시오.

  - 철수든 영희든 이기려면 결국 상대 차례가 왔을 때, 돌을 가져가게 두면 안 된다. 이것을 전제로 패턴을 찾아보자.(철수 승리 0, 영희 승리 1)
  - 돌이 0개면 첫 차례인 철수가 돌을 가져갈 수 없으니 영희가 이긴다(dp[0] = 1)
  - 돌이 1개면 첫 차례인 철수가 돌 1개를 가져가므로 영희가 돌을 주어갈 수 없으므로 철수가 이긴다(dp[1] = 0)
  - 돌이 2개면 첫 차례인 철수가 돌 1개를 가져갈 수 밖에 없고 영희가 나머지 돌을 전부 가져가므로 영희가 이긴다(dp[2] = 1)
  - 돌이 3개면 1 + 1 + 1인 경우 철수의 승리, 3도 철수의 승리. 그러므로 철수의 승리(dp[3] = 0)
  - 돌이 4개면 1 + 3 또는 3 + 1이므로 철수가 1개를 가져가든 3개를 가져가든 영희의 승리(dp[4] = 1)
  - 돌이 5개면 3 + 1 + 1로 철수가 나머지 돌을 가져가므로 철수의 승리(dp[5] = 0)
  - 자, 규칙이 좀 보이는가? 철수가 1개를 가져가든 3개를 가져가든 남는 돌에서 영희가 최선의 선택을 할 것이다. 그렇다면 현재 돌의 개수가 n이라면 영희에게 주어지는 선택지는 !dp[n - 1] 또는 !dp[n - 3]이라는 결과다(철수 기준이니까 영희 기준으로 보려면 당연히 결과를 반전시켜야

한다). 그렇다면 철수가 선택할 수 있는 건 !dp[n - 1]과 !dp[n - 3] 중에서 자신이 승리하는 결과가 나오는 수일 것이다.

- 돌이 4개일 때, 철수가 1개 또는 3개를 선택할 수 있으며, 남는 돌일 때는 영희가 기준이 되어 판단한다.





미니맥스 알고리즘

- AI는 어떻게 바둑에서 최선의 수를 둘 수 있는가? 최근에는 알파고를 보면 알 수 있듯이 데이터 기반으로 파악을 하지만 과거 알고리즘 중심으로 발전하던 시기에는 게임 이론을 접목한 AI 알고리즘을 사용했다.
- 미니맥스 알고리즘은 위의 돌 게임과 비슷하다. 현재 내가 둘 수의 경우의 수를 고려하면서 내가 최대한의 이익을 가져가면서 상대에게 최소한의 이익을 요구하는 수를 계산하는 것이다.
- 즉, 내가 수를 둘 차례에 둘 수 있는 수를 전부 뽑아낸 다음에 그 수에서 시작해 게임이 끝날 때까지 나올 경우의 수를 전부 구해서 그 중에서 내가 이기는 걸 +, 상대가 이기는 걸 -해서 합산이 가장 높은 수를 선택하는 것.

알파-베타 가지치기

- 미니맥스 알고리즘은 한 수를 둘 때마다 경우의 수를 전부 구해서 점수를 구해야 한다. 굉장히 비효율적이다. 그렇기에 백트래킹 기법을 추가하여 상대에게 이득이 되는 수가 발견되면 상대는 그 수를 택할 것이므로 나머지 수를 탐색하지 않는 방식으로 가지치기를 한다.
- 자, 말로만 하면 머리가 터질 터이니 수도 코드를 보면서 마음을 다스리자

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
```

```
        return the heuristic value of node
    if maximizingPlayer
        v := -∞
        for each child of node
            v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
            α := max(α, v)
            if β ≤ α
                break (* β cut-off *)
        return v
    else
        v := ∞
        for each child of node
            v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
            β := min(β, v)
            if β ≤ α
                break (* α cut-off *)
        return v
```

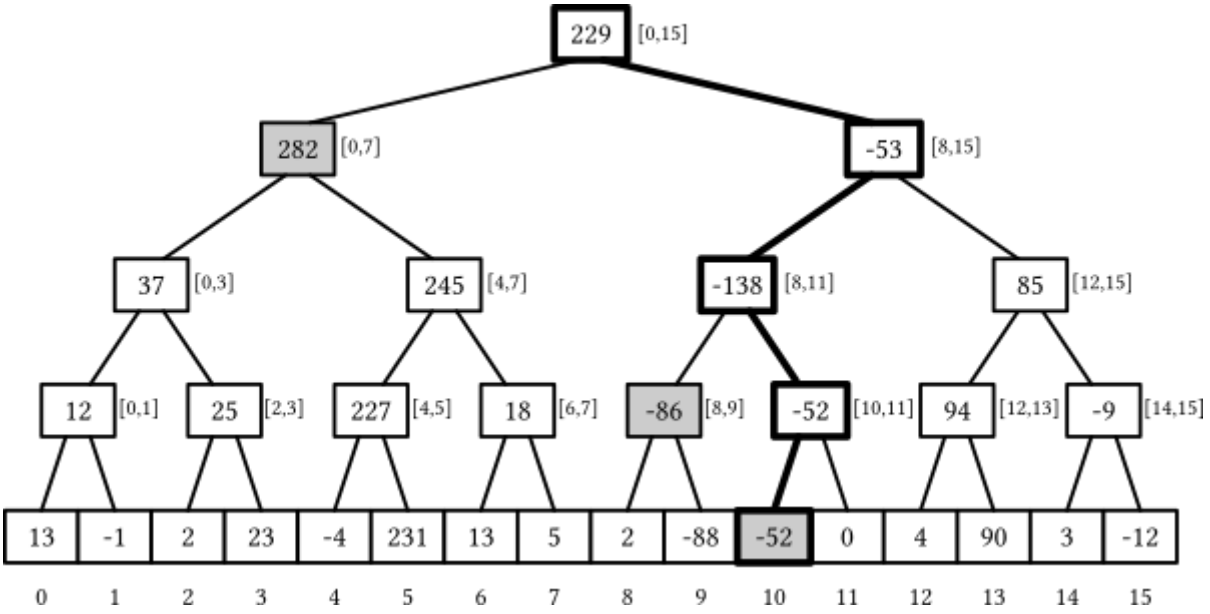
풀어보자

- 돌 게임 5, https://www.acmicpc.net/problem/9659
- 알파 틱택토, https://www.acmicpc.net/problem/16571

세그먼트 트리

세그먼트 트리란?

- 배열이 있을 때, 배열의 구간 합을 트리로 나타낸 것이 세그먼트 트리다.



- 리프 노드는 각 배열의 원소값이고 그 위 부모 노드는 자식 노드의 합이다. 그렇기에 각 부모 노드는 배열의 일정 구간의 합인 셈이다.
- 세그먼트 트리 조성

```
void makeTree(int[] arr, int node, int start, int end)
{
    if (start == end)
        tree[node] = arr[start];
    else {
        int mid = (start + end) / 2;
        makeTree(arr, node * 2, start, mid);
        makeTree(arr, node * 2 + 1, mid + 1, end);
        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }
}
```

세그먼트 트리 사용 예시

- 만약, 배열 원소 3 ~ 6까지의 구간 합을 구하고 싶다고 가정하자.
- 세그먼트 트리에서는 4가지 탐색 기준이 생긴다.
    - 내가 찾고 있는 구간(3~6)이 현재 탐색하는 노드의 구간 범위를 벗어났을 때
    - 내가 찾고 있는 구간(3~6)이 현재 탐색하는 노드의 구간 범위에 쏙 들어올 때
    - 내가 찾고 있는 구간(3~6)이 현재 탐색하는 노드의 구간 범위에 일부 속할 때
- 노드의 구간 범위를 벗어난 경우 당연하게도 찾을 필요가 없다. 즉, 현재 노드와 그 노드에 포함된 자식 노드들도 볼 필요 없다.
- 노드의 구간 범위에 쏙 들어오면 더 아래로 내려간다. 만약, 범위가 동일하면 밑으로 들어갈 필요 없이 그냥 답으로 치면 된다.
- 일부 구간이 겹치면 구하려는 범위를 분할하여 겹치는 쪽은 탐색하러 내려가고 겹치지 않는 쪽은 다른 노드를 찾는다.

풀어보자

- 구간 합, https://www.acmicpc.net/problem/2042