|  |  |  |
| --- | --- | --- |
| 종류 | 알고리즘 | 정리내용 |
| **알고리즘** | | |
| 최단경로 | 다익스트라 | **가. 목적.** dist[] 배열을 사용해서 현재 모든 경로 중 최단경로를 가지는 목적지를 먼저 방문하여, 한 지점에서 나머지 모든 지점까지의 최단 경로를 구하는 알고리즘.  **나. 시간복잡도**  배열 시간복잡도 : O(|V|^2+|E|)  우선순위 큐 시간복잡도 : O(|E||log|E|) (O(|E||log|V|))  **다. 구현 종류 및 방법**  계속해서 dist배열을 갱신하면서 최단 경로를 검색한 후 방문하는 배열 중심의 구현 방법, 우선순위 큐를 사용해서 자동으로 최단 경로를 찾아줘 큐에 들어있는 위치를 먼저 방문하는 우선순위 큐 중심의 구현 방법이 있다.  (노드가 적고 간선이 많을 경우 배열 방법을 사용할 수 있다.)  1. 배열 구현 방법  dist배열을 모두 INF로 초기화 한 뒤, 시작점의 값만 0으로 만들어준다. 이후 마지막 노드를 제외한 n – 1 번 동안 방문하지 않은 노드 중, 최단경로를 가지고 있는 노드를 시작으로 연결되어 있는 노드들의 최단경로를 비교 후 갱신해준다. 갱신하는 노드 또한 한 번 방문한 노드라면 더 이상 갱신해주지 않는다.  다른 노드가 갱신되면서 이전에 방문한 노드의 최단거리가 갱신되는 경우가 생길 수 있지 않을까 의아해할 수 있는데, 그럴 경우가 전혀 없다. 시작점에서 한 경로씩 뻗어나가는 형태이기 때문에, 더 나아갈수록 경로의 값이 계속 더해지게 되어있고, 다익스트라에서 해결하지 못하는 음의 가중치가 없고서야 방문할 때의 dist 값보다 더 작아져 갱신되는 일은 없다.  방문하지 않은 노드 중 최소 dist 값의 노드를 방문하는 이유는, 위에서 말했던 대로 최단경로를 가진 노드의 값은 더 이상 남은 노드들에 의해서 갱신 될 수 없기 때문이다. 반대로 나머지 노드들은 최단 경로를 가진 노드에 의해 갱신될 수 있기 때문에 먼저 방문해서는 안된다.  2. 우선순위 큐 구현 방법  최단경로를 가진 노드를 빠르게 찾기 위해 배열과 선형탐색 대신 우선순위 큐를 사용한 것만으로 생각할 수 있지만, visited 배열을 사용하지 않는다는 중요한 차이도 존재한다.  dist가 갱신되면서 큐에 거리와 도착 위치가 전달되는 형식이라, 나중에 다시 갱신되었을 때 의미없는 값이 큐에 남아있을 수 있고  (2개의 노드를 지나온 경로와 아직 1개의 노드를 지난 경로 등 다양한 경로들이 큐에 남아있다. 이때 1개의 경로를 지난 지점의 위치를 dist 배열과 대조해보았을 때, 일치하지 않는다면 그 경로는 그 지점까지의 최단경로가 될 수 없으므로 continue하게 되는 것이다.)  배열에서 visited가 필요없는 연산을 줄이기 위해서 사용되는 것처럼 이 의미없는 dist값들을 건너뛰기 위해서 현재 위치에서의 dist 값과 일치하지 않으면 건너뛰는 if(dist[now] != heapdist) continue; 라는 문장을 사용하는 것이다.  배열에서의 visited(여기서의 visited는 새로운 노드의 값을 갱신해 줄 때 사용하는 것을 의미한다.) 와 위 문장 모두 사용하지 않고 같은 결과를 낼 수 있지만, 말했듯이 필요없는 연산을 진행하면서 효율성을 엄청나게 떨어트리므로, 작성해주는 것이 올바르다.  **라. 의문을 가졌던 점들.**  1. 우선순위 큐 구현을 보면 heapdist != dist[now] 문을 찾을 수 있는데, 이 문장의 기능.  우선순위 큐를 사용하여 최단 경로만 방문하다 보니 이전의 더 오래걸리는 경로들이 큐에 남아있게 된다. 이때 나온 값들이 현재 now 위치에서의 dist 값보다 크면 갱신의 필요가 없으므로 굳이 아무 의미없는 검사를 피하기 위해 continue 문으로 그 단계는 넘어가 준다.  > 연산자를 사용해도 되지만, 큐에 들어가있는 값들은 모두 dist를 갱신하면서 동시에 넣은 값이고, 현재 dist 값과 같지 않다는 것은 이후 갱신이 적어도 한 번 있었음을(현재 값보다 새 경로가 더 작을 때만 갱신을 하기 때문에) 말하는 것으로, != 연산자를 써도 무방하다.  (또한, 알고리즘에 대해 정확한 이해를 했는지 스스로 확인하기 위한 용도도 있다.)  2. 왜 현재 dist값들 중 최단 값만 이동하는가?(최단값 우선으로 이동하는 이유?)  우선순위 큐에서는 최단값 우선으로 이동하는 것이 시간단축에 도움이 된다. 결국에는 큐에 들어있는 값을 꺼내서 확인하게 되는데, 최단값 우선 이동하게 되면 현재위치에서의 dist 값도 또한 최단값일 가능성이 높아 이후 최단값을 갖고 있지 않은 dist 값들을 모두 continue 할 수 있기 때문이다.(내 생각)  3. 2에 이어서, 최단값을 우선 방문하는 것이 속도 때문이라면, 항상 먼저 방문한 루트가 최단 경로가 아닌 경우도 존재하는가?  당연히 그러지 않을까? 이 알고리즘은 약간 bfs와도 닮아있다. 조건에 부합하면 큐에 넣고 해당 큐 값을 방문해서 다시 거기서부터 검사를 진행해나가는 방식인데, bfs는 가중치를 따지지 않아 항상 먼저 방문한 것이 최단경로라서 visited배열을 주로 활용하는 반면, 다익스트라는 먼저 방문한 것과는 관련이 없지만 먼저 최단거리를 dist에 저장할수록 불필요한 조건 연산의 양을 줄일 수 있기에 우선순위 큐를 사용한다고 생각할 수 있겠다.(불필요한 조건 연산의 양을 줄인다는게 의미가 있으려면 dist[now] != heapdist : continue 의 문장은 필수적이다.)  **마. 관련문제**  특정한 최단 경로, 최단경로, 최소비용 구하기, 최소비용 구하기 2, KCM Travel. |
| 최단경로 | bfs\_dijk | **가. 목적.** 다익스트라 기반으로 쓴 코드인데 dist 배열을 아예 사용하지 않고 우선순위 큐로 매번 최단경로만 이동하도록 구현, visited배열을 사용해서 먼저 방문한 곳은 또 방문하지 않는 bfs의 느낌을 첨가한 방법이다. 프림 알고리즘과 모양이 매우 유사하다. 하지만 경로의 우선순위를 결정하는 방법이 다르니 주의.  **다. 구현 종류 및 방법**  visited로 방문한 위치는 넘어가는 문장의 위치가 매우 중요(if(visited[now]) continue; visited[now] = 1;  일반 bfs처럼 구현했다가는 틀린 답을 도출해낸다.(bfs는 모든 가중치가 같기 때문에 양자 상관이 없다.(코드에 따라 계산하는 과정조차 똑같음)  하지만 다익스트라는 우선순위 큐가 결정한 순서에 따라 움직이는 것이기 때문에 visited를 먼저 큐에 넣은 경로라고 해서 방문했다고 친다면 그것은 우선순위 큐를 사용하는 이유를 없애는 매우 멍청한 짓이다. 따라서 가능한 모든 경로를 입력받은 후, 우선순위 큐가 결정한 순서에 따라 이동하면서 방문을 체크해주어야 한다.(인접한 경로는 겹치는 경우가 없으므로 visited를 당장 해줘도 될 것 같다는 착각이 들 수 있다. 하지만, 이후에도 다시 그 지점과 연결된 경로가 있다는 것을 생각한다면, 항상 먼저 검사하는 것이 최적의 경로는 아님으로(먼저 방문하는 것이 최적의 경로이지) visited를 내부에서 해줘서는 안된다는 것을 알 수 있다.)  하지만 일반 다익스트라에서는 dist배열에 바로 바로 값을 저장해준다.  이는 왜 그래도 되는가?  dist값을 갱신해주는 것은 visited와는 다르다. dist 값 비교문은 더 방문했던 방문하지 않았던 더 빠른 경로가 존재한다면 값을 갱신하고 큐에 대입하여 나중에 방문하도록 하는 형태이고, visited는 먼저 방문하는 것이 최적의 경로라는 사실에 의존하여 방문했던 곳을 검사하는 용도로만 사용하는 것이다.(없어도 되지만 효율을 떨어지게 되는 문장)  따라서 dist 배열의 형태에도 효율을 높이기 위해서라면 visited를 써도 된다. |
| 최단경로 | 플로이드 와샬 | **가. 목적.** 그래프의 모든 노드에서 모든 노드까지의 최단거리를 한 번에 구하는 방법으로, 시간을 비교적 오래 걸리지만 구현이 단순하다.  **라. 의문을 가졌던 점들.**  1. (웜홀문제 : 가중치가 음수인 그래프) 자기 자신으로 오는 경로를 처음에 INF대신 0으로 초기화해주는데, 이 문제에서는 그러지 않는다. 그랬을 때 다른 경로의 최솟값에도 영향을 주지 않을까?  않는다. 왜냐하면 다른 경로에 영향을 주는 경우는 거쳐가는 노드를 자기 자신으로 가지는 경우 즉, a[i][j] = min(a[i][j], a[i][i] + a[i][j]); 뿐인데, 따라서 모든 간선의 가중치가 양수인 경우 a[i][i]의 값이 0이 아니건 0이건 a[i][j]의 값에는 변화가 없다.(최솟값만 저장하기 때문) 웜홀 문제처럼 간선 가중치 중 음수가 존재하는 경우에는 나머지 최단경로 뿐 아니라 a[i][i] 의 값조차도 차이가 없다. 이는 우선 a[i][j]의 경우 위의 식에서 a[i][i] 가 음수인, 음의 무한루프가 생기는 경우에만 최솟값이 변하게 되는데, 이때 음의 루프가 생긴 것은 a[i][i]의 값을 0 또는 INF로 초기화시키는 것과 별개의 문제이기 때문이다.  -> 그렇다면 음의 루프가 생기지 않는 음의 가중치 문제에서는 a[i][i]의 값을 꼭 0으로 초기화 해줘야 할까?  (달리 쓰자면, 정확히 음의 루프가 플로이드 와샬 알고리즘에서 어떻게 영향을 미치는가?)  **마. 관련문제**  플로이드, 웜홀 |
| 최단경로 | 벨만포드 | **가. 목적.** 다익스트라와 목적하는 바는 같다. 한 노드에서 다른 모든 노드들까지의 최단경로를 구하는 알고리즘이다. 하지만 다익스트라의 작동 방식 특성상 가중치가 음인 간선은 계산하지 못하므로, 가능한 모든 이동경로를 돌아서 그 중 최솟값을 dist배열에 저장하는 조금 원시적인 알고리즘이 벨만 포드이다.  가능한 최단경로는 음의 사이클이 없는 이상 같은 노드를 2번 이상 지나지 않을 것이라는 개념을 중심으로 동작하는 알고리즘이라 봐도 무방하다.  **나. 시간복잡도**  O(|V||E|)  **다. 구현 방법 및 종류**  최단거리는, 음의 사이클을 형성하지 않는 경우 같은 노드를 2번 지나지 않는다. 따라서 어떤 한 노드에서 다른 모든 노드에 대한 최단거리를 생각해 보았을 때 그 거리들이 지나가는 노드의 개수는 1개에서 v – 1개 사이이다. 가능한 모든 경로를 검사하므로 이 개수만큼 반복하는 반복문을 돌려준다.  이 반복문 안의 현재 dist 배열에 저장되어있는 값은 최대 i(첫 반복문의 반복인자가 i) 개의 노드를 지나온 경로까지 고려했을 때 시작점부터 그 노드까지의 최단경로의 크기이다.  그래프의 출발이 가능한 노드 1에서 v까지 중에서 그 dist값이 아직 INF인 값을 제외하고 모두 arr에 저장된 경로를 통해서 가능한 지점으로 이동하고, 그 경로가 최단경로라면 dist의 값을 갱신해준다. (반복문 2개 사용)  dist 값이 INF 인 노드를 건너뛰는 이유는 출발점에서 도달할 수 있는 노드인지 아직 확인하지 못했을 뿐 아니라, 음의 가중치를 갖는 간선 때문에 걸러주지 않으면 이동하려는 next 노드의 dist값도 INF 인 상황에서 INF + cost(cost < 0) 으로 값이 갱신되는 오류가 생기기 때문이다.  음의 사이클을 찾는 문제라면 dist의 초기값, 시작점으로부터 도달이 가능한 노드인지에 대해서는 신경쓰지 말아야 한다. arr로 입력받는 간선들의 가중치에 의해 형성되는 것이고, 어떤 값으로 초기화하던지 간에 가장 마지막에 한 번 더 반복시켰을 시 최단거리 갱신이 일어났다면 사이클이 존재하는 단순한 형태이기 때문이다. 시작점으로부터 도달이 가능한지 신경쓰게 되면 도달이 불가능한 노드끼리 음의 사이클을 형성한 경우에 대해서 검사하지 못하기 때문에, 무시하고 서로 연결이 가능한 노드들끼리 계속해서 갱신해주도록 하면 된다.(따라서 시작점의 dist값도 아무런 관련이 없다.)  **라. 의문을 가졌던 점들**  1. 연결되지 않은 노드들끼리 음의 사이클을 형성할 경우 INF값을 걸러주지 않음으로써 ac를 받았는데, 이는 왜그런가?  이는 웜홀 문제에 특정해서 생각해봐야 할 듯 싶다. 음의 사이클이 존재하는지만 묻는 문제임으로, n – 1번 반복문을 돌린 후 마지막으로 한 번 더 돌렸을 때 dist 값의 갱신이 있다면 음의 사이클이 존재한다는 뜻이기에 초기 dist 값을 어떤 값으로 초기화하던지, 간선이 연결되어있던지 안되어있던지에 상관 없이 일어나는 일이다.  정리하자면, 모든 간선이 연결되어 있다고 하더라도 음의 사이클이 존재하지 않는다면 그 최소 경로는 최대 n - 1개의 노드만 거칠 것이고, 따라서 마지막으로 한 번 더 돌렸을 때 갱신이 있다면 음의 사이클이 있는 것으로 생각할 수 있는 것이다. 초기 dist 값을 어떤 값으로 초기화하던지 간에 음의 사이클은 arr로 입력받은 간선들의 가중치에 의존하는 것이기에 초기화 값과 상관없이 일어난다.  1+. 아래 문장을 지운다는 것이 시작 노드와 모든 노드를 연결하는 것이라고 했는데, 그렇게 연결하게되면 없던 음의 사이클을 만들어 낼 수 있는 것 아닌가?  (정확하게 하자면 여기서 시작 노드란 경로에서의 출발점을 의미하는 것이 아니라, 알고리즘을 돌리기 위해 필요한 하나의 점화점 같은 느낌이다. 어느 지점에서 시작했는지와 별개로 답이 구해지며, 점화점에서 닿지 않는 노드끼리 음의 사이클을 형성하는 경우를 방지하기 위해 아래의 문장을 지우거나, 모든 노드들을 점화점으로 돌려보면서 가능한 모든 음의 사이클을 찾는 방법도 있다.)  그렇지는 않을 것이다. 왜냐하면 느낌상 경로가 생긴 것을 뿐 실제 arr 벡터에 경로가 추가된 것이 아니기에 새로운 경로가 생긴다던가 음의 사이클이 생기는 것은 아니다. 다만 굳이 표현해보자면 마치 자동차에 시동을 걸어주기 위해 배터리를 사용하는 것처럼, 시작점에서 도달하지 못하는 지점들끼리의 음의 가중치가 있는지 확인하기 위해 갱신을 강제로 시작시키는 방법이다.  -> 실질적으로 기존 방법과 다른 최단거리 값을 출력하는 노드들은 시작점에서 도달하지 못하는 점들뿐일 가능성이 크다(INF가 충분히 크다면). 나머지는 n – 1 번의 경로를 반복하면서  INF + cost(cost가 음수) 로 업데이트 될 가능성이 있지만(또한 이래서 기존 방법에서는 INF 일 때 건너뛰는 이유이고) 나중에라도 이어진 경로를 찾고 그 경로가 INF + cost 이하라면 정상적인 최단거리로 갱신 될 것이다.  if(dist[j] == INF) continue; 문의 역할 -> 출발점에서 아직 도착하지 못한 지점 걸러줌.  이 문장을 지운다는건? -> 아직 도달하지 못한 지점, 점화점에서 절대 도달하지 못할 지점들에서도 주변 경로를 통해 값을 갱신하게 해줌.  (위에서도 말했듯이 점화점에서 결국 도달할 수 있는 지점이라면 제대로 된 값으로 갱신될 것이고, 도달할 수 없는 지점이라면 INF + cost(cost < 0) 등으로 잘못된 값이 저장되어 있겠지만 음의 사이클을 찾는 것만이 목적이기 때문에 이 문제에 한정해서는 오류가 없다.)  2. 음의 사이클은 정확히 언제 생기는 것인가?  for(auto &cur : arr[j]){  int next = cur.first, cost = cur.second;  if(dist[next] > dist[j] + cost){  dist[next] = dist[j] + cost;  }  }  여러 개의 노드가 사이클처럼 연결되어있을 때 그 사이클을 돌았을 때 경로 가중치의 합이 음수면 발생한다. 가중치가 모두 양수일 때와는 정반대로 사이클을 돌면 돌수록 최단거리가 짧아지기 때문에 계속해서 관련된 노드들의 최단거리가 갱신된다.  3. 반복문에서 노드를 1부터 v까지 순차적으로 방문하는데, 이렇게 되면 나중에 갱신되는 dist 값에 의해 먼저 갱신할 수 있었던 상황을 못하게 되지 않을까?  그런 경우에는 다음 노드에서 갱신된 값으로 다시 검사하기에 문제 될게 없다. 또한 최단경로는 지나치는 노드가 n – 1 개 이상인 경우가 없으므로(음의 사이클을 제외하고) 그 반복문 내부에서 모두 갱신되게 되어있다.  **마. 관련문제**  웜홀, 타임머신 |
|  | 유니온  파인드 | **union과 find의 조합.(disjoint set, 분리 집합 등 다양한 이름 존재)**  1. 서로 같은 집합에 속해 있는지 아닌지를 판단하는 함수 find.  (트리에서는 루트의 값을 집합의 이름으로 하여 구현할 수도 있다.)  2. 집합에 요소를 더하거나 집합끼리 합칠 때 사용하는 함수 union.  **기본 작동(사용) 방식**  상당히 간단하다. 그래프에 대한 분리 집합을 예로 들면, 배열을 사용해서 각 노드의 집합을 각자의 이름으로 초기화 해준다. (아마도 노드의 이름이 1부터 N까지의 숫자일 것이고 이름은 겹치지 않을 것이기에) 사용하려는 목적에 따라 구현이 달리겠지만, 주로 사용은 find 함수를 사용해서 해당 노드들의 최상위 노드, 즉 집합의 대표를 비교함으로써 같은 집합에 포함되어있는지를 판단하고, 경우에 따라 union 함수를 사용해 두 노드를 같은 집합에 포함해준다.  **구현 방식**  find 함수는 재귀적으로 구성하면 된다. 초기화 과정에서 각 노드의 소속 집합을 각자의 이름을 초기화 해주었고, union 함수는 두 노드를 같은 집합으로 묶되 그 집합의 이름은 대표 노드로 설정해준다. 즉, 특정 노드의 집합을 찾아내려면 집합 이름이 저장되어 있는 배열의 인덱스와 요소값이 일치 할 때까지 재귀적으로 찾아주면 된다는 것.  union 함수는(공용체가 아니라 이름은 달리 선언해야 하지만) 서로 다른 집합을 합칠 때 사용하는 함수이다. 두 집합에 각각 소속되어 있는 노드를 find 함수를 통해 그 집합의 이름을 찾고, 둘 중에 하나의 이름을 바꿈으로써 대표 노드를 다른 집합에 소속된 노드로 만들어준다.  **최적화**  find에서 재귀적 호출을 할 때 리턴되는 값을 현재 인덱스에 저장해주면 다음 탐색에서 더 짧은 재귀 안으로 결과를 찾을 수 있다.  return find(root[x]); => return root[x] = find(root[x]);  문제 해결 방법과 관련이 없다면, union에서 두 집합을 더할 때 숫자가 더 작은 쪽을 부모집합으로 해준다던지, 특정 규칙을 정해놓으면 쏠림 현상 없이 고르게 분포된 트리를 형성할 수 있다. |
|  | 최소  스패닝  트리 1  (크루스칼) | **유니온 파인드를 사용해 집합개념으로 푸는 방법**  주어진 모든 경로를 가중치 오름차순으로 정렬한 뒤 작은 것부터 방문하면서 현재 집합에 포함되어 있는 경로인지 아닌지 확인한다. 포함되지 않은 경로라면 당연히 사이클을 구성하지 않으므로, 지나가는 경로로 하고 현재 집합에 포함시켜 준다.  그렇게 모든 간선에 대해 검사를 마치거나 모든 노드를 집합에 포함시켰다면 반복을 중지하고 지나온 경로들이 최소 스패닝 트리이다.  **이해가 안갔던 부분**  1. 단순히 visited 배열로 구현할 순 없었나?(내가 처음 생각한 방법) 유니언 파인드를 사용하지 않고?  할 수 없다. 예로 시작과 끝 두 노드 모두 방문한 노드일 경우 경로를 잇지 않는다는 룰을 정했을 때, 두 노드 모두 방문한 상태이지만 이을 수 있는 경우가 존재한다. 그 룰 없애자니 사이클을 생성하지 않게 하는 법도 딱히 존재하지 않는다. 유니온 파인드를 쓰는걸로. |
|  | 최소  스패닝  트리 2  (프림) | **최소 스패닝 트리는 최단 경로를 구하는 문제가 아님을 명확하게 하고 프림 알고리즘을 이해하려고 하자.**  **완성된 최소 신장 트리에서 하나씩 분해한다고 생각하면 이해가 되려나?**  **n – 2개의 간선이 연결된 그래프를 최소 신장트리라고 하면, 마지막 하나 연결되지 않은 노드를 연결할 때 가능한 경로 중 최소 경로를 연결하면 최소 신장트리가 완성된다. 그 방법대로 거꾸로 간다고 생각하면 이 동작방식이 이해가 갈 것이다.**  다익스트라와 유사한 모양을 가지고 있어서 헷갈리지만, 최단 거리 위주의 방문을 하게 되면 방문 순서가 완전히 달라진다. 주의. 모양이 비슷한 것에 대해서는 마지막 부분에 써놓았다.  최단 경로는 모든 점을 되도록 지나지 않으려 할 것이고 최소 스패닝 트리는 모든 점을 이어야 한다는 점을 생각하면 그리 어렵지 않다.  작동 방법을 생각해보면, 현재 이동할 수 있는 노드들 중에서 (방문하지 않은 것들만) 그 경로의 가중치가 가장 작은 노드로 이동한다.  모든 노드를 방문해야 하므로 가중치가 낮은 경로를 먼저 따라 방문하는 것인데, 크루스칼과의 차이점을 찾자면 현재 지점으로부터 연결되어 있는 경로만 우선순위 큐에 넣어 검사한다는 것이다. 크루스칼에서는 유니온 파인드로 사이클이 생기는 경로는 뛰어넘는 반면에, 프림에서는 연결되어 있는 것 중 최소 경로를 우선 방문하고 방문한 노드는 visited처리 해줌으로써 사이클이 생기는 것을 방지한다.  프림을 이용해 방문하는 노드끼리의 경로를 더하면 최소 스패닝 트리의 가중치가 구해진다.  이해가 안갔던 부분  1. 매번 헷갈리는 것 중 하난데, visited 값 대입의 위치.  다음 경로를 우선순위 큐에 저장하는 부분에서 visited 체크를 해주면 안되는 이유는, (현재 위치 now에서 바로 연결된 노드 중 서로 같은 노드는 존재하지 않는다. 따라서 거기서 연결된 노드에 visited 체크를 해주어도 그 반복문에서 또 다른 연결 노드를 검사하는 것에는 영향이 없다. 하지만 그러지 말아야 되는 이유는,) 우선순위 큐를 사용하면서 다음 경로를 큐에 넣음과 동시에 방문처리를 하는 것은 정말 멍청한 짓이기 때문이다. 그렇게 방문 처리를 할 것이면 우선순위 큐를 사용해 값을 크기별로 정렬하여 차례대로 방문할 이유가 전혀 없다. visited만 생각한다면 우선순위 큐가 결정한 순서대로 방문하는 것이 아닌, 먼저 이어진 경로대로 방문하는 가중치 없는 bfs가 돼 버린다.(따라서 이는 bfs에서는 잘 통한다.)  따라서 우선순위 큐가 결정한 값을 top을 통해 빼서 방문하고, 방문처리를 해주는 것이 당연하다.  하지만 아직 if(visited[now]) continue; 문의 역할이 헷갈릴 수도 있겠다.  이 문장의 과연 의미가 있을까라는 의문은 아마도 방문하는 지점이 이미 visited처리가 되어 있을까라는 의문으로부터 시작할 것이다. 큐에 경로를 대입할 때 먼저 if(!visited[next])로 방문한 지점에 대해서 걸러주기 때문에 말이다.  기본 다익스트라 알고리즘의 if(dist[now] != heapdist) continue; 문장을 기억할 것이다.  이 문장과 위의 문장과의 역할은 동일하다고 보면 될 것 같다. 그것도 완벽하게.  큐에 경로를 저장할 때 방문하지 않은 점들에 대해서 경로를 저장해주지만, 이후 다른 경로를 통해 그 지점을 앞서 방문했을 경우가 존재한다. 그러면 그 방문 이전에 경로가 저장된 것들은 그대로 우선순위 큐에 남아있게 되고, 더 작은 값으로 먼저 방문했던 곳이기에 검사의 의미가 없다.  (다만 다익스트라에서는 도착 지점에 도달하거나 dist값이 최소값으로 이미 되어있기에 검사를 하더라도 시간만 더 걸리지 답은 변함이 없지만, 프림은 방문하는 모든 지점 가중치를 더하기 때문에 답에 변화를 준다는 차이점은 있다.)  + continue되는 경로는 사이사이에 존재한다. 마지막에 몰리지 않는다.  처음에는 다익스트라와 모양은 비슷하나 다른 방법이라고 써두었다. 방문순서가 다르기에 다른 방법임이 확실하다. 하지만 왜 모양이 이렇게 비슷할까?  - 이어서 계속 –  **증명**  [https://ko.wikipedia.org/wiki/%ED%94%84%EB%A6%BC\_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98#%EC%A0%95%ED%99%95%EC%84%B1\_%EC%A6%9D%EB%AA%85](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A6%BC_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98%23%EC%A0%95%ED%99%95%EC%84%B1_%EC%A6%9D%EB%AA%85) |
| 그래프 | BFS | **너비 우선 탐색**  가중치가 없는 그래프에서 노드간 최단거리를 구하기 위해 주로 사용되는 알고리즘이다. (다익스트라는 가중치가 있는 그래프에서 사용) \_ 따라서 문제에서는 주로 그래프를 주기보단 어떠한 행위를 하는데 걸리는 시간, 가중치가 모두 일정할 때 목표를 이룰 수 있는 최소 시간을 묻곤 한다.  (가끔 가중치가 존재하더라도 그 종류가 2, 1로 제한되는 등의 경우라면 덱을 우선순위 큐 대신 사용하여 bfs 처럼 풀 수도 있다.)  **큐를 사용해서 구현한다.** (이런 면에서 어느정도는 위상정렬과 비슷)  dfs와 사촌간이기 때문에 이 또한 백문이 불여일견 그림을 보면 작동 방식이 이해가 쉽게 간다.  루트 노드부터, 바로 아래 깊이의 모든 노드를 먼저 방문한다. 즉, 깊이가 1인 모든 노드를 방문하고 나면 그제서야 그들의 자식 노드를을 차례로 방문하고, 그다음 그들의 자식노드들을 방문하고,,, 하는 알고리즘이다.  + 메모리 초과 방지하는 스킬  다익스트라에서는 다음 방문 지역을 큐에 추가하는 과정에서 visited를 체크해주면 안된다는 것을 알 수 있었다. (우선순위 큐가 결정할 일은 visited가 나서는 것이기 때문.)  하지만 bfs에서는 다음 방문지역을 넣어주면서 visited를 미리 체크해주는 것은 메모리 절약에 있어서 필수적이라는 사실을 알게 되었다.  큐에 들어있는 3가지 경로 모두 다음 노드를 가르키는 상황에서, 방문 후 visited를 체크해주는 것으로 구현되어있으면 3가지 경로가 존재하는 현재 단계에서는 다음 노드가 걸러질 수 없으므로 3번 모두 큐에 들어가게 된다. 불필요한 메모리 사용, 불필요한 연산으로 이어진다. 따라서 조건문을 통해 다음으로 이동할 수 있는 노드를 큐에 넣음과 동시 visited 처리를 해주어야 한다. |
| 그래프  DP | DFS | **깊이 우선 탐색**  트리에서 연결된 노드에서 자식 노드 우선으로 (깊이 중심) 방문하는 알고리즘.  백문이 불여일견이라고 그림으로 보면 바로 어떻게 동작하는지 이해가 간다.  그래프에서의 이동, 주어진 데이터를 움직이는 보드 삼아 4방향 이동, 백트래킹 알고리즘 등 다양한 곳에서 응용해 사용할 수 있는 알고리즘이다.  재귀적으로 방문하기 때문에, 이전에 방문했던 노드는 방문 체크를 해줌으로써 무한 루프를 방지하고, 다른 경로로의 재방문을 허용할 때는 백트래킹으로 방문 체크를 켜주고 꺼주면 된다.  메모이제이션과 함께 사용하면 탑다운 dp를 구현하는, 엄청난 사용범위를 보이는 녀석이다.  (굳이 알고리즘이라고 하기도 그럼) |
|  | 백트래킹 | dfs의 재귀적 이동에서 탈출조건에 의해 방문했던 순서대로 돌아오면서 반환값을 사용한다던지 방문 표시를 제거해준다던지 등 되돌아 오고 나서의 구현을 백트래킹 방법이라고 할 수 있겠다.  꼭 재귀적 탐색이 아닌, 이동해온 방향의 반대로 거슬러 올라가면서 처리해주는 구현도 백트래킹이라고 할 수 있는 것으로 알고 있다.  -- 비슷한 류  trace back 이라고 dfs, bfs, 다익스트라 등 그래프 이동에서 방문한 순서를 저장해주기 위한 메소드로, 다음 지점의 값을 인덱스 삼아 현재 위치를 저장하면 도착 지점에서 거꾸로 출발 지점까지 trace back 할 수 있다. |
| 그래프 | 위상정렬 |  |
| DP | 냅색 | knapsack 문제. 정형화된 웰논 dp 중 하나.  백준: 앱, 평범한 가방 문제 |
| DP | LCS | 테이블 만들어서 같다면 왼쪽 상단 +1, 다르다면 위 아래 중 큰 값 저장.  저장되는 값 중 최댓값이 lcs, 실제 lcs 를 구하고자 할 때는 최댓값부터 1씩 줄여나가면서 nlogn 짜리 LIS 백트래킹처럼 구하면 된다.  마. 관련문제  LCS |
|  | LIS | 1. 완전탐색  2. dp(재귀와 메모이제이션 사용 or 그냥 dp)  3. nlogn  - 방법에 대한 이해  특정 인덱스에 한 원소가 위치할 수 있었던 이유는, 가장 처음 추가하는 상황에서 앞에서 그보다 작은 수가 있었기 때문에 뒤에 붙을 수 있었던 것이다. 즉, 그 수가 입력되는 상황에서 그 수의 위치까지는 lis가 성립된다.  그리고 lowerbound에 있는 값을 교체해주는 이유는, 뒤에 나오는 수들이 참조를 못하는 경우가 생기지 않게 해주기 위함이다. (뒤에 나오는 값을 위한 준비? 정도라고 생각하면 편하다. 실제 lis 값을 의미하지도 않고(실제 lis는 벡터의 마지막에 추가되는 것들만 모으면 된다.), lis 길이에 영향을 주지도 않는다. 나중에 나오는 원소들이 추가되면서 올바른 길이를 구할 수 있도록 한다.)  -> (여기서 lowerbound에 수를 덮어씌우고 이후 마지막에 수를 추가할 때는 순서적으로 올바른데 왜 실제 lis가 아닌지, 또 수를 덮어씌움으로써 lis가 안되는데도, 마지막에 수를 추가하는 것이 어떻게 lis의 길이를 의미할 수 있는지 의문이 들 수 있다.) -> 마지막 의문점에서 설명.  확실하게 알아둬야 할 것은, 가장 마지막에 새롭게 추가하던 lo위치에 바꾸던, 그 상태에 들어있는 원소들은 모두 “추가하려는 원소”보다 순서가 앞서 등장한 것들이다. (크기에 따라 정렬된 것과 헷갈리지 말 것) 따라서 크기비교를 통해 들어간 자리(인덱스)가 곧 그 원소가 마지막인 lis의 길이인 것이다.(그 원소까지 참조했을때)  4. 백트래킹을 사용해 nlogn 방법에서 실제 lis를 알아내는 법.  참조하는 값이 그 값을 대입하려는 인덱스보다 이전에 위치한 값을 가르키게 하면 된다. 정리했던 것처럼 덮어씌우던, 추가하던 처음부터 대입하려는 인덱스까지는 lis가 맞기 때문이다. 하나씩 가르키게 하고 가장 마지막에 위치한 값부터 거꾸로 참조해나가다 보면, lis를 구성하는 요소들만 가져올 수 있다.  4+. 백트래킹을 사용하지 않고 nlogn 방법에서 실제 lis를 알아내는 방법.  제공된 모든 수열(전깃줄 문제에서는 b 전봇대 값) 에 대해서 참조하는 값이 lis 길이를 구하는 벡터 어느 위치에 저장되는지와 left값을 모두 저장해둔다. lis길이를 구한 이후, 가장 마지막에 저장한 수열부터 제일 처음 저장한 수열까지 반복문을 통해, 벡터 저장 위치가 num *(num = result – 1, lis에서 저장된 위치를 검사하기 위해 사용되는 변수)* 과 일치하는지 검사하고, 일치한다면 num -= 1을 해주면서 진행한다. 그러면 이후에 새로 갱신된 값을 무시하고 이전에 저장되었으며, 이전 인덱스에 위치하는 값들만 뽑아낼 수 있다. 선형시간 안에 가능.  (한가지 주의 할 점은 백트래킹에서는 track이 이전 인덱스의 pole값을 저장하게 하였으나 이 방법에서는 당장의 저장하는 pole의 left값을 저장한다.)  4.5 ++++++  이의 작동방식과 어떻게 이 방식이 lis를 가져다주는지 좀 더 정확하게 명시할 필요가 있다.  사례를 들어 생각한다면 이해는 쉽다.  예) 1 4 5 7 3 4 인 수열  -> lis : 1 4 5 7  -> nlogn방식의 마지막 벡터 : 1 3 4 7  -> 마지막 7부터 시작해서 lowbound 또는 마지막에 추가했던, 모든 수열에 대해서 당시 추가할 때의 위치를 저장한 배열과 현재 lis의 크기를 비교, 일치하면 lis 크기를 하나 줄이고 그 수가 최종 ㅣis의 일부가 된다.  5. 인덱스 트리를 이용한 방법  **라. 의문을 가졌던 점들(nlogn)**  1. 왜 실제 lis가 될 수 없는가?  lis는 순서를 지켜야 한다. 이 방법대로라면 이후에 나온 값이 lowerbound에 덮어씌워지게 되면, 그 인덱스까지는 lis를 구성할 수 있지만, 이후는 이전에 벡터에 추가된 값들이기 때문에 순서에 어긋난다. 그래서 모든 요소가 다 대입된 벡터는 (모든 요소가 추가된 상황이 아닌 이상) lis가 아니다.  2. lowerbound에 수를 덮어씌우고 나면 전체 길이에서 lis가 더 이상 아니게 되는데, 왜 마지막에 노드를 추가하는 것은 올바른 lis 길이를 의미하는가?  우선 현재 검사하는 값은 벡터에 들어있는 모든 요소보다 이후에 나온 것이다. 마지막에 추가되는 경우, lis로써 순서상의 문제는 없어보인다.  하지만 lis는 될 수 없다(lowerbound 덮어씌인것이 있을 경우). lowerbound에 요소가 덮어지면서 그 뒤에 위치한 요소들과 순서가 어긋낫기 때문이다(뒤에가 존재하지 않은 경우 뒤에 설명). 그럼에도 뒤에 추가할 수 있는 이유는 가장 마지막 요소는 lowerbound가 덮어씌워지기 전부터 존재했으며, 그 마지막 요소가 추가되었을 당시에는 lis를 이루고 있었기 때문이다. 그렇다면 마지막 요소가 lowerbound된 경우를 생각해 볼 수 있는데, 이 또한 뒤에 위치한 요소들이 없으므로 lis에 해당한다. 그래서 lowerbound로 일부 값이 바뀌면서 더 많은 값이 마지막에 추가될 수 있게 하더라도 lis의 길이에는 전혀 문제가 없는 것이다. (마지막에 추가되는 경우 lis가 되는 것이고, 중간에 추가되는 경우 이미 존재하던 lis에 숫자만 바꾼 것이므로)  (이게 모든 경우이다. lowerbound를 마지막에 아닌 곳에 덮어쓰기, 마지막에 덮어쓰기, 새로운 요소를 추가하기. 뒤에서 2개는 모두 당시 lis를 이루는 상태이다.)  ++(lowerbound의 연쇄적 영향)  가장 마지막 값을 바꿔주면 그 다음 값들 중 더 많은 것이 추가될 수 있다는 것은 직관적으로 이해가 간다. 그러나 중간, 마지막과 거리가 있는 인덱스에 값을 덮어준다고 이후 추가되는 값의 위치에 영향을 줄 수 있을까?  그런듯 하다. 왜냐하면 1 4 5 7 에서 2, 3, 5과 5, 3, 2를 각각 덮어주면, 전자는 1 2 3 5, 후자는 1 2 5 7이 나온다. 이전의 lowerbound가 이후 lowerbound의 위치를 키움으로써, 결과적으로 더 작은 값이 마지막 값으로 들어올 수 있게 해준다.  이러한 직관상, lis의 길이를 구하려고 할 때 순서에 따라 참조하면서 lowerbound의 위치에 값을 덮어주는 것은 합리적으로 보인다.  마. 관련문제  전깃줄, 가장 긴 증가하는 부분 수열, 가장 긴 바이토닉 부분 수열 |
|  | LCA | Lowest Common Ancestor / 최소 공통 조상. |
| 트리 | 세그먼  트  트리 | 세그먼트 트리, aka 세그트리는 배열에서 특정 구간(subsequence) 의 구간합을 빈번하게 구해야 할 때 구현하는 자료구조이다. (구간합 뿐 아니라 구간 최소, 최댓값 등 구간에 해당하는 대푯값을 특정할 수 있으면 모두 가능) |
| 문자열 | KMP |  |
| DP  비트마스킹 | TSP  외판원순회 | Travling salesmen problem. |
| 메소드 | 비트마스킹 |  |
|  |  |  |
| **자료구조** | | |
| 자료구조 | 기본 | **배열 & 리스트**  컴퓨터 프로그램을 통해 데이터를 연속적으로 저장하고 활용하는 데 있어서 가장 기본적인 단위.  배열은 c에서 기본적으로 구현이 되어 있으며, 리스트는 c++ stl 을 사용해야 하기 때문에 직접 포인터를 활용해 구현해야 한다.  두 구조의 가장 근본적인 차이라고 한다면 연속된 데이터들의 실제 저장 위치라고 할 수 있겠다. 배열은 선언하게 되면 해당 크기만큼의 공간이 일렬로 배열을 위해 할당된다. 즉, 데이터의 실제 저장 위치가 모두 이웃한다.  하지만 리스트의 경우 동적 할당으로 원하는 크기만큼의 공간을 여러 번 선언해주기 때문에, 특정값의 주소를 알아낸다고 하더라도 다음 값이 어디에 저장되어있는지 알 수가 없다. (삽입시 매번 선언하는건 시간 효율이 떨어지므로 실제 활용에서는 2의 제곱승 형태로 크기를 선언해 주곤 한다.)  따라서 포인터를 활용해 다음 노드를 생성과 동시 그 주소를 이전 노드의 포인터에 저장해주어, 이웃한 데이터간 이동을 가능하게 한다. (즉, 데이터의 저장 순서를 형성할 수 있음)  배열의 장점은 선언된 공간에 해당하는 위치는 인덱스로 상수시간 O(1) 에 참조할 수 있다는 것이고, 리스트의 장점은 중간 특정 데이터를 삭제 하더라도 배열처럼 뒤에 이어지는 모든 데이터를 움직이지 않고 이웃한 노드의 포인터만 이어주면 된다는 것이다.  각각의 장점에 따라 더 구체적인 자료구조: 힙, 이진탐색트리, 해쉬 맵등의 구현에 데이터의 기본 형테로 사용될 수 있다.  **스택, 큐, 덱**  **이진트리** |
| 자료구조 | 힙  (우선순위큐) | 옛날에는 C로 힙 다 구현해서 썼는데 stl 쓰다가 구현 다까먹었네,,,,  우선순위 기준에 따라 데이터를 정렬해서 저장하고, 정렬 기준 중 우선순위가 가장 높은 데이터를 삭제, 참조할 수 있는 자료구조.  알고리즘 문제에서는 다익스트라, 프림 등의 알고리즘을 구현하기 위해서 필수적으로 사용된다.(우선순위 큐라는 이름으로)  힙은 배열을 기반으로, 이진 트리의 모습으로 구현된다. 쉽게 생각하면 이진 트리이나, 데이터의 저장 규칙이 사용자가 정한 우선순위 규칙에 따르는 자료구조이다.  시간복잡도 : O(logN) / 저장, 삭제, 탐색 모두 일정한 logN이다.  동작 방식은 아래와 같다.  1. 삽입  배열을 활용해 현재 데이터 갯수를 인덱스로 우선 저장해준다. 즉, 트리의 가장 마지막 부분에 저장.  이후 부모 노드와 우선순위 비교를 통해, 자리바꾸기를 진행한다. (사실 매번 자리바꾸기는 낭비이다. 따라서 비교 후 우선순위가 더 낮은 노드만 아래로 값을 복사해주고, 삽입한 값이 대입될 위치가 어딘지 찾았을 때 그 인덱스에 값만 대입해준면 된다.)  + 배열을 활용한 데이터 저장은 시작 인덱스를 1로 한 후 왼쪽 자식 노드는 현재 값의 \* 2, 오른쪽 자식 노드는 \* 2 + 1 해주면 된다. 부모노드는 / 2.  2. 삭제 |
| 정렬 | 패배자들 | 버블정렬, 선택정렬, 삽입정렬 : O(n^2)  구현 과정만 이해한다면 절대로 다시 볼 일 없는 구데기 알고리즘.  힙정렬 : O(NlogN)  자료구조 힙을 구현해 현재 데이터를 힙에 모두 넣은 후 정렬된 데이터를 꺼내면서 저장함으로써 정렬한다.  병합정렬 : O(NlogN)  분할 정복 개념을 기반으로 정렬하는 알고리즘.  병합정렬 구현 방식 |
| 정렬 | 퀵정렬 | 의미있게 사용되는 정렬 알고리즘. O(NlogN)의 시간복잡도를 가진 알고리즘(힙 정렬, 병합 정렬 등) 들 중에서 데이터의 이동량이 적고 메모리를 적게 사용한다는 점에서 가장 빠른 알고리즘으로 알려져 있다.  대부분 알고리즘 문제들에서도 정렬이 필요한 상황이면 퀵소트로 배열을 정렬하고 필요시 이진탐색등을 사용하지, 쿼리가 많은 특정 상황이 아니고서야 이진 탐색 트리(stl map, set) 등을 사용해서 데이터를 관리하지 않는다.  구현방식 |
| 탐색 | 이분탐색 | 너도 알고 나도 알고 옆집 개도 아는 이분탐색. 더 이상의 설명은 생략.  여지껏 푼 문제는 몇 안되지만 그 중 제일 괜찮은 이분탐색 문제:  <https://codeforces.com/contest/1480/problem/C> |
| 탐색 | 보간탐색 | 이분탐색의 효율성을 증가시킨 모델.  이분탐색에서처럼 단순히 인덱스 값으로만 이동하는 것이 아니라, 배열의 데이터를 참조해 배열의 데이터가 1의 고른 간격으로 분포되어 있을 때 찾으려고 하는 값의 위치가 어느 인덱스에 있을지를 비례식을 통해 대략 구하는 것. 값을 이용하다보니 타겟 값이 배열에 없는 경우 무한반복을 하게 되는 버그가 생기는데, 매 루프에서 타겟값이 arr[first]와 arr[last]사이에 있는지를 검사해줌으로써 해결할 수 있다. |
| 정렬  탐색 | 이진탐색  트리 | 이진트리 + 데이터 저장규칙 (수시로 삭제, 삽입, 탐색을 위한 자료구조 끝판왕)  레드블랙트리(stl map 기반), avl 트리로 개선하여 사용한다.  부모 노드의 데이터보다 크기가 작다면 왼쪽 자식 노드로 연결, 더 크다면 오른쪽 자식 노드로 연결해서 데이터를 저장해둠으로써, 저장된 데이터를 탐색하는 과정을 빠르게 한다.  편향된 데이터를 저장하다 보면 왼쪽과 오른쪽 서브트리의 높이의 차가 많이 벌어지는 경우가 생기는데, 이러면 탐색하는데 비효율적이기에 트리의 높이를 조절해주는 avl 이진탐색트리를 구현한다.  시간복잡도 : log2N (데이터 삽입, 삭제, 탐색 모두 일정)  **라. 의문을 가졌던 점들**  트리 노드를 구조체로 만들고 구조체를 중심으로 함수들을 쌓아올리다보니, 헷갈리는 것들이 많아졌다.  1. 새 노드를 추가할 때 구조체 포인터를 만든 후 함수를 사용해서 동적 할당 하는 것.  이런 구현에서 동적 할당은 필수인듯 싶다. 그저 구조체를 정적 선언한다면 수시로 자료를 추가하고 삭제하는 과정에서 더 이상 필요없어진 노드들의 메모리를 관리할 수 없기에, 메모리 누수가 발생할 것이다.  동적 할당한 값을 포인터로 반환하는 이유는 리스트를 이해한다면 쉽게 이해가 될 것이다. 동적 할당으로 생성된 메모리의 위치를 가르키는 포인터가 있어야 노드끼리의 연결이 가능하다.  이 모든 기능을 함수로 묶은 이유는 아래의 포인터의 초기화 과정까지 깔끔하게 한번에 해주기 위해서일듯.  2. 동적 할당으로 포인터에 새 노드의 위치를 넣어주면서 트리노드의 2개의 포인터를 null 값으로 초기화 해주는 이유.(null은 굳이 초기화 안해도 있던 것 아니었나?)  int형의 기본이 0인 것처럼 포인터의 기본값은 null이다. 하지만 이것은 구조체가 전역에서 선언되었을 때의 경우이고, 일반 변수와 마찬가지로 지역에서 선언한다면 모두 쓰레기 값이 저장된다. 함수에서 동적 할당을 통해 선언되는 트리 노드의 경우 양 포인터에 모두 쓰레기 값이 저장되기에 추후 함수의 구현에서의 편리를 위해 미리 null값으로 초기화해준다.  3. 함수에 따라 이중 포인터와 포인터로 나누어 인자를 갖는 이유.  아마 다른 곳에서도 이런 이야기를 적어 놓았겠지만, 이중 포인터의 역할은 함수 밖 원 포인터가 **다른 값을 가르키도록 포인터의 화살표를 바꾸는 것이고,** 포인터의 역할은 **포인터가 가르키는 대상의 값을 변경하는 것이다.** 이것만 기억하자. |
| 자료구조 | 해시  맵 |  |
| 메소드 | 분할정복 |  |
|  |  |  |
| **C 기본** | | |
| C  기본 | 함수와  포인터 | 1. 포인터의 가장 기본적인 개념은 어떤 변수의 저장위치를 값으로 저장한다는 것이다.  이는 일반 변수가 할 수 없는 일이고, 포인터 변수만이 해당하는 타입에 일정한 메모리공간을 할당, 변수의 위치를 값으로 저장할 수 있게 된다.  2. 함수에서 한 인자를 포인터나 이중 포인터로 설정했다면, 그 의미는 다른 것이 아닌 함수에 집어넣는 매개변수의 값을 [복사]를 통해 이루어지는 ‘값에 의한 호출’이 아닌 ‘참조에 의한 호출’ 을 하기 위함이다. 즉, 함수 바깥에 존재하는 변수의 값을 함수 내부에서도 변경하기 위하여 그렇게 작성한 것이라고 생각하면 된다.  예로 한 포인터의 값을 함수안에서도 변경하고 싶다면, 함수의 인자를 이중 포인터로 해주는 것이다.  이는 메모리 관리와도 큰 관련이 있다. 함수 내에서 선언한 선언한 (정적) 변수는 함수가 종료되면 자동 해제되기 때문에 매개변수로 따로 저장할, 함수 밖에서 할당된 변수의 주소를 가지고 와서 값을 저장하면 계산한 값을 가져갈 수 있다. (메모리 주소 등의 값을 동적 할당 없이 반환하려고 할 때) |
|  | 동적정적메모리해제 | 변수의 메모리 해제 위치(지역변수에 대해서)  종료된 함수 내부에 있는 정적 메모리는 자동 해제된다.  동적 메모리는 해당이 되지 않는가?  아마도 그런 것 같다. 그래서 트리 노드를 생성할 때도 노드 포인터에 동적으로 메모리를 할당하는 함수를 따로 구현하는 것이 가능했다.  BTreeNode \* MakeBTreeNode(void)  {  BTreeNode \* nd = (BTreeNode\*)malloc(sizeof(BTreeNode));  nd->left = NULL;  nd->right = NULL;  return nd;  }  이렇게 말이다. 이렇게 한다면 나중에 원하는 시기에 메모리를 해제할 수 있어 자료구조 같은 데이터 저장을 구현할 때에는 상당히 용이하다.  실행 중에 종료될 함수에 정적 메모리를 사용하게 된다면 미리 메인 함수 등에서 값을 저장하기를 원하는 변수의 주소를 매개변수로 받아와야 한다. 그리고 그 변수에 값을 복사하고 함수를 종료해야 해제된 메모리를 참조하는 에러가 일어나지 않는다.  (트리 구조에서 2중 포인터를 매개변수로 둔 것도 같은 이유) |
|  | NULL값 | 포인터 상수 NULL은 주소의 0값을 의미한다. 즉, 포인터를 초기화할 때 0을 대입한것과 같은 값이라는 뜻.  문자열에서의 null값도 아스키코드에서 0번에 해당한다. |
|  | 함수포인터 | 함수를 다른 함수의 인자로 넣게 될 때 함수포인터를 사용한다. 특히 사용자가 특정 함수를 만들어 기존 함수의 인자로 집어넣어 원하는 기능을 구현하려고 할 때 주로 사용하는 방법이다.  함수 포인터를 보기에 앞서 typedef 문을 봐보자.  typedef (자료형) Function(int);  또는  typedef (자료형) (\*Function)(int);  ~~두 코드 모두 같은 의미이지만, 전자는 후자에 비해 조금 더 코드에 자유도가 높은 편이다.~~  ~~무슨 말이냐 하면, 전자의 경우 구조체의 멤버나 함수의 인자 등을 생성할 때~~  ~~Function action, Function \* action 모두 사용이 가능하다. 두 표현 모두 컴파일러가 action을 함수 포인터로 인식하여 구조체의 멤버로는 \*를 붙여 선언했으나 함수의 인자로는 없이 받아 구조체의 함수포인터에 \* 없이 받은 인자를 저장할 수도 있다. 둘 다 같은 의미라는 것.~~  완전히 잘못된 내용이다.  **왜 헷갈리는지 정확하게 알려주도록 하겠다.**  보통 함수는, 변수와 다르게 선언시 자료형이 없는 것처럼 보인다. 변수와 구분되게 배우고 사용해왔으니 그럴 만 하다. 하지만 함수 포인터를 공부하면서 느껴야 할 부분은, **함수도 자료형이 존재한다는 것이다.(내 생각)**  반환값과 매개변수의 자료형, 매개변수의 갯수 등이 하나의 함수 자료형을 결정한다고 생각하면 될 듯 하다.  따라서 위의 두 코드 중 첫번째 코드는 정수형 변수를 예로 들면 int의 별명을 만들어 준 것이고, 아래의 코드는 int \* 의 별명을 만들어 준 것이다. 즉, 서로 다른 코드다.  그렇다면 이 코드는 어디에 사용될 수 있을까?  사실 그냥 변수를 선언하듯이 함수의 기본 선언 형식을 대체해도 된다. 하지만 그렇게 쓰는 경우는 매우 드물고(함수는 각기 다른 기능을 가지고 있어서 그 자료형을 공유하는 경우가 드뭄) 함수를 다른 함수에 매개변수로 사용해야 할 때 위의 typedef 문을 사용한다.  변수를 함수의 매개변수로 전달하는 방법은 2가지가 있었다. 값에 의한 전달, 참조에 의한 전달이 그 둘이다. 저 위의 두 코드도 하나는 함수의 별명을, 하나는 함수 포인터의 별명을 설정해 줌으로써 매개변수로 함수를 보낼 때 값에 의한 전달을 할 것인지 참조에 의한 전달을 할 것인지 결정할 수 있는 것이다.    그게 바로 첫번째 문장을 \* 없이 매개변수로 설정해도 아무 문제가 없었던 이유이다. **참조에 의한 전달로 함수를 받은 것이 아니라, 값에 의한 참조로 함수를 복사해서 저장한 것이라고 생각하면 되겠다.**  +++++----- 추가 설명.  일단 함수 포인터는 하나의 자료형이다. 이것을 제대로 이해하지 못해서 생기는 문제다.  우리가 int 형 자료형에 typedef int newint; 라고 써준다면, int형 변수를 선언하려면 newint던지 int를 앞에 붙여주어야 한다. 함수 포인터도 이것과 마친가진데,  기존의 함수 포인터는  (자료형) (\*함수 포인터 **변수** 이름)(매개변수 자료형);  이렇게 선언해야 해서 상당히 귀찮다. 그래서 typedef로  typedef (자료형) (새로운 함수포인터 이름)(매개변수 자료형);  써준 다음에  (새 함수 포인터 이름) \* (함수 포인터 변수 이름);  이렇게 간단하게 만들어 주는 것이다. (같은 함수를 저장하는 다른 포인터를 만들 때 편해진다. -> 함수의 매개변수로 함수를 받으려고 하면 특히 그렇다.)  또는  typedef (자료형) (\*새로운 함수포인터 이름)(매개변수 자료형);  (새 함수 포인터 이름) (함수 포인터 변수 이름);  이렇게 사용할 수도 있다. |
|  | 동적할당 | Table a;  Table \* a = (Table\*)malloc(sizeof(Table));  에 대해.  둘다 같은 역할을 하고 있지만, 자료형의 차이에서 알 수 있듯이 하나는 구조체 변수, 하나는 구조체 포인터이다. 다른 함수에서 구조체를 인수로 받아오려면 구조체 포인터를 인수로 작성해서 받아오게 되는데, 후자의 경우 포인터의 이름만 써줘도 된다는 편리함이있다.  제일 중요한 차이점은, 전자처럼 지역변수나 매개변수 등은 스택 메모리에 저장되어 함수를 빠져나가면 해제되고, 동적할당은 힙 메모리에 저장되어 그러지 않는다는 점이다. 즉, 메모리를 프로그램이 돌아가는 동안 계속해서 생성하고 해제하기 위해서는 동적할당이 필수적이다.  이 이유 때문에 다양한 함수를 통해 사용되는 구조체의 경우(노드 등) 포인터에 동적 할당 통해 메모리 공간을 할당한다. |
|  | 배열 메모리 사용 | 배열을 선언했을 때 컴퓨터가 해당 배열에 대한 메모리 공간을 어떻게 할당하는지 어느정도는 알 것이다.  2차원 배열 또한 1차원 배열과 마찬가지로 직선 공간에 배열의 크기만큼 순차적으로 공간을 할당한다는 것도 잘 알 것이다.  따라서 2차원 배열을 선언하고 참조할 때, [] 연산자에 들어가는 인덱스들을 선언시 행과 열의 크기로 제한하지 않아도 된다.  굳이 이렇게 쓸 일은 없으나, 가능하다. 따라서 int arr[5][5] : arr[1][9] 로 참조가 가능하다. arr[1][9] 가 의미하는 것은 arr[1 \* width + 9] 로, arr[2][4] 와 정확하게 같은 것이다.  마치 포인터로 2차원 배열을 참조할 때 사용하는 방법과 유사하다.  아래 포인터 배열의 글에서 2중 포인터에 2번 동적할당을 함으로써 2차원 배열을 구현하는 코드가 있는데, 그 코드가 이에 대한 직관적 이해에 도움이 될 것이다.  파면 팔수록 2차원 배열, 2중 포인터 등의 2중 [] 연산자 사용은 사용자 편의를 위해 컴파일러가 알아서 처리하는 하나의 모양새일 뿐이라는 생각이 든다. |
|  | 배열이름 | 배열이름은 자주 배열의 시작요소의 위치를 대신하여 포인터 변수처럼 사용되곤 한다.  이의 배경에는 배열과 포인터가 표기법의 혼용을 허용한다는 점이 존재한다.  (int data[5] or int \* data) : data[0] = \*(data + 0) 이 가능하기 때문에, 배열의 이름은 포인터처럼 기능할 수 있는 것이다. 이는 함수에 배열의 주소를 전달하는 과정을 이해하는데 매우 핵심적이다. |
|  | 배열포인터 | 2차원 배열을 함수에 매개변수로 전달할 때 주로 배열 포인터를 사용했을 것이다.  이는 왜 이렇게 작동하는지, 다른 방식으로 2차원 배열을 전달하는 방법은 없는지에 대한 고민들을 정리해나가면서 포인터의 다양한 형태와 차원에 대해 이해를 높여 본다.  ++(읽기 전)  c에서 포인터를 생각함에 있어서 가장 중요하게 이해해야 하는 점은 포인터도 하나의 값이라는 것이다. 특정 자료형의 주소값을 포인터에 저장한 것일 뿐이다. c는 변수를 선언하면 특정 메모리 공간에 변수가 해제되기 전까지 이를 할당한다. 함수의 매개변수의 대입만으로는 한 번에 배열 전체를 복사할 수 없기 때문에(새 배열을 초기화 할 때 다른 배열을 바로 대입하지 못하는 것과 같다.) 배열 첫 번째 주소를 전달함으로써 다른 함수 간 같은 배열을 사용하는 것이다. 이 부분만 정확하게 이해한다면, 다양한 형태나 차원의 포인터를 주고 받더라도 이해에 어려움이 없을 것이다.  ++(작은 것 하나 더)  기본적이지만 꽤 중요한 내용인 것 같아 남겨둔다. 주로 배열의 이름은 그 배열의 첫 요소의 주소값을 가지고 있다고 배운다. (int arr[2] : arr == &arr[0]) 하지만 각각은 분명이 다르다. 하나는 배열이고, 하나는 주소값이다. (궁금하면 typeid를 써보도록) 따라서 arr == &arr[0] 이고 arr[0] == &arr[0][0] 이므로 arr == &(&arr[0][0]) 이라는 논리는 가능할 수 없다.(포인터와 배열이 표기를 혼용하긴 하지만 배열은 배열이고 포인터는 포인터이다.)  그래서 배열 포인터를 전달해 줄 때에는 배열 포인터의 자료형에 알맞는 형태로 호출을 해줘야 한다. 비록 배열 포인터와 1차원 포인터 모두 첫 요소의 주소값을 가질지라도, 그 타입은 분명 다르기 때문에 각각 arr, arr[0] 혹은 &arr[0], &arr[0][0] 로 호출해주어야 한다.  1. 배열 포인터란?  이 내용을 공부하면서 가장 우선적으로, 정확하게 이해해야 할 것은 c에서의 메모리 공간 활용이다.  c는 변수를 선언하면 특정 메모리 공간에 변수의 값을 집어넣는다.  그 변수가 저장되어 있는 특정 위치를 저장할 수 있는 것이 포인터이다.  그 포인터의 값을 활용해서 특정 위치의 이웃한 메모리 공간들을 간접 참조할 수 있다.  그게 바로 배열의 원리이다.    배열은 특정한 메모리 공간에 순차적으로 변수들의 값을 저장한다. 따라서 배열을 구성하는 변수 하나만의 위치를 알더라도 배열 전체를 참조할 수 있다.  2차원 배열도 마찬가지이다. c는 2차원 배열도 linear하게 저장하기 때문에, 열의 크기만 안다면 열의 크기를 현재 인덱스에 더함으로써 다음 행의 요소에 접근할 수 있다.  그렇다면 배열 포인터는 무엇인가?  배열 포인터는(적어도 내가 아는 바로는) 대체로 1차원 배열의 1차원 포인터를 말한다. 포인터가 1차원 배열을 가르키기 때문에, 이 포인터를 ptr[] 연산자로 참조하게 되면 열 단위로 이동하는 것이 아니라 2차원 배열의 행단위로 이동하게 된다. 한 포인터가 1차원 배열 전체를 지칭하기 때문이다.  따라서 1차원 포인터이기 때문에 [] 한 번, 1차원 배열이기 때문에 [] 한 번 사용해서 주소로 받아온 2차원 배열의 시작 주소를 통해 ptr[][] 로 마치 2차원 배열처럼 사용할 수 있다.  2. 2차원 배열을 왜 배열 포인터로 입력받아야 하는가?  사실 다양한 방법이 있으나 이 방법이 가장 간편한 것으로 알고 있다. 호출 시 2차원 배열의 첫 시작 주소를 (어떠한 방식이던간에) 전달하면 그 주소를 (\*arr)[Max] 를 통해 받아온다. 그 지점부터 메모리 공간을 Max 단위로 나누어 2차원 배열처럼 참조가 가능하게 해준다.  3. 1차원 포인터로의 2차원 배열 받아오기  위에서 배열 포인터도 1차원 포인터이기 때문에 그 값으로 배열 시작위치의 메모리 위치를 가진다는 것을 이해했을 것이다. 따라서 1차원 포인터로도 같은 값(배열의 시작위치)을 전달하게 되면, 2차원 배열을 충분히 전달할 수 있다.  다만 유일한 차이점이라 할 수 있는 것은 1차원 포인터로 값을 받아오는 것이기 때문에 일반적인 2차원 배열처럼 [] 연산자를 연속으로 사용할 수 없다. 배열 포인터로 읽어오는 경우와의 표현적 차이 정도라고 생각하면 되겠다. 따라서 1차원 포인터로 주소를 받아오는 경우 2차원 배열의 메모리 사용 방식을 고려하여 한 번의 [] 연산자로 2차원 배열을 사용하면 된다.  4. 2중 포인터가 일반적인 2차원 배열을 받아올 수 없는 이유  어떻게 생각하면 당연하다. 2중 포인터는 포인터의 포인터다. 포인터의 주소를 전달하던지 해야 하는데, 배열의 시작주소, 1차원 포인터의 값에 해당하는 값을 전달하려고 하니까 문제가 생기는 것이다. 굳이 2중 포인터를 사용하려면 2차원 배열을 달리 정의해야 한다.  5. 주소만 전달받는 것이라면, 서로 자료형이 다르면 왜 에러가 나는지?  포인터를 선언할 때 자료형 구분없이 포인터의 사이즈가 모두 8바이트여도 다른 자료형의 주소값을 포인터에 할당할 수 없는 것처럼 함수전달도 같은 내용이라고 생각한다. 여기서 실 주소 대신 배열의 이름으로 호출하는 경우가 있어 헷갈릴 수 있는데, 이는 포인터, 배열 표기법의 혼용에 따라 문법에서 유일하게 허용하는 경우라고 생각하면 될 듯 하다. |
|  | 포인터배열 | 1. 포인터 배열이란?  말 그대로 포인터를 요소로 같는 배열을 말한다.  2. 포인터 배열이 사용되는 경우  2중 포인터로 2차원 배열을 입력 받는 경우. 2중 포인터는 포인터의 포인터이기 때문에 주소 값을 요소로 가진 배열의 주소 값으로 초기화 해주어야 한다. 그래야 2중 포인터를 2차원 배열처럼  [][] 의 2중 참조 연산자를 사용할 수 있기 때문. 즉, 배열을 포인터처럼 바꾼 다음에 포인터를 배열처럼 활용하는 것이다. 아래는 그 코드.  void practice(int \*\* ptr){      for(int i = 0; i < 2; i++){          for(int j = 0; j < 5; j++){              cout << ptr[i][j] << " ";          }          cout << "\n";      }  }  int main(void){      int arr[2][5], idx = 0;      for(int i = 0; i < 2; i++){          for(int j = 0; j < 5; j++){              arr[i][j] = idx++;          }      }      int \* parr[2];      for(int i = 0; i < 2; i++){          parr[i] = arr[i];      }      practice(parr);      return 0;  }  보다시피 어차피 데이터를 집어넣기 위해 2차원 배열을 선언해야 할 뿐 아니라 1차원 포인터 배열도 선언해서 행 개수만큼 반복해서 또 초기화를 해주어야 한다. 쓸데없는 짓이 따로 없다.  애초에 2차원 배열이 정적 형태가 아니어도 된다면, 동적할당을 활용할 수 있다.  int main(void){      int \* parr[2];      for(int i = 0; i < 2; i++){          parr[i] = (int\*)malloc(sizeof(int) \* 5);      }      // 1차원 포인터 배열을 만들어서 필요한 배열의 크기만큼 동적 할당하기(초기화)      // 이렇게 구현하면 첫번째[] 연산자는 배열참조, 두번째 []연산자는 포인터 참조인 것이다.      int idx = 0;      for(int i = 0; i < 2; i++){          for(int j = 0; j < 5; j++){              parr[i][j] = idx++;          }      }        practice(parr);      for(int i = 0; i < 2; i++){          free(parr[i]);      }      return 0;  }  동적할당을 여러번 사용하는 2중 포인터.  int main(void) {      int idx = 0;        // 2차원 배열을 2차원 배열처럼 사용(인덱스 2번사용)하려면      // 동적할당을 여러번 해야한다.      int\*\* arr;      arr = (int\*\*)malloc(sizeof(int\*) \* 2);      arr[0] = (int\*)malloc(sizeof(int) \* 2 \* 5);      for (int i = 1; i < 2; i++) {          arr[i] = arr[i - 1] + 5;      }      // 이렇게 하면 동적 할당은 2번만 사용했지만      // 간접 참조로 인해 []연산자로 순차적으로 요소에 접근이 가능하다.      // 특별히 관련은 없으나 정적선언에서와 비슷한 메모리 구성        for (int i = 0; i < 2; i++) {          for (int j = 0; j < 5; j++) {              arr[i][j] = idx++;          }      }      practice(arr);      free(arr[0]);      free(arr);      return 0;  } |
|  |  |  |
| **PYTHON** | | |
| 파이썬  기본 | ps  데이터 입력 | 파이썬에서는 다양한 방법으로 데이터를 입력받아올 수 있지만, ps 에서 주어지는 데이터의 형식이 유형별로 상당히 일관적이기 때문에 데이터 입력 구현을 어느정도 정리해 두면 문제를 신속하게 해결하는데 도움을 줄 수 있다. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |