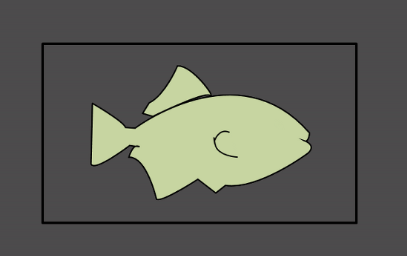
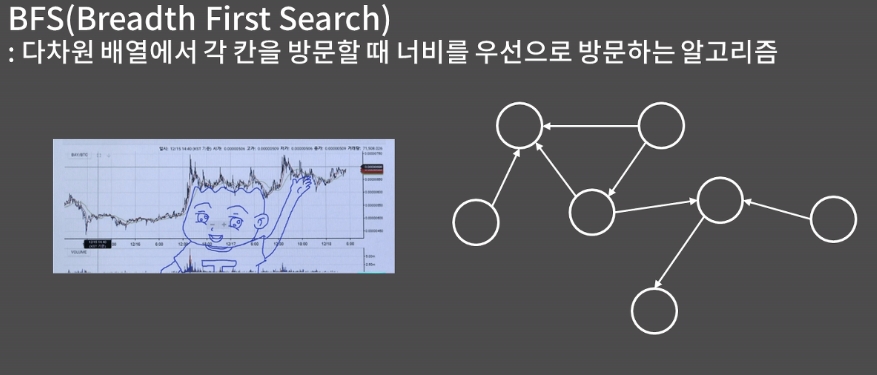
BFS

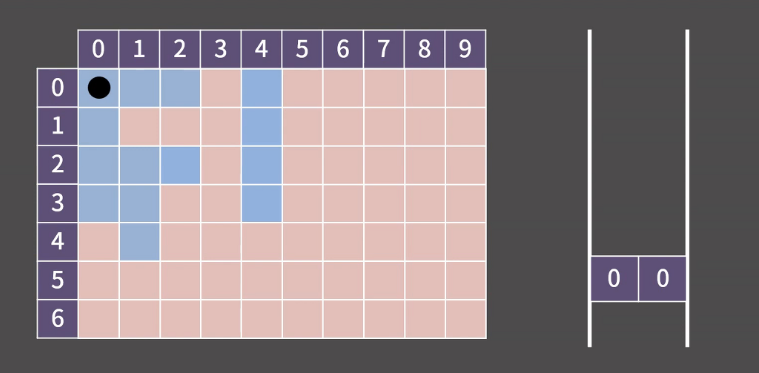


그림판의 페인트 기능을 이용하면 물고기의 색을 바꿀 수가 있습니다. 페인트 기능은 외부 윤곽선을 따라서 구분되는 영역의 색을 한꺼번에 바꾸는 거고, 이런걸 Flood Fill이라고 부르기도 합니다.

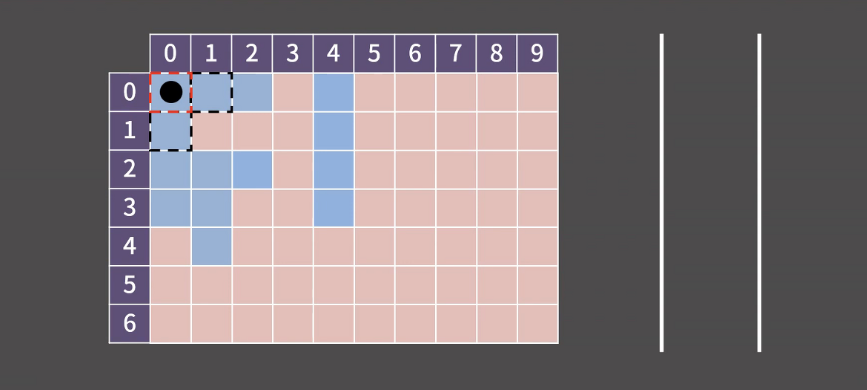


 원래 BFS는 그래프라는 자료구조에서 모든 노드를 방문하기 위한 알고리즘입니다. 여기서 말하는 그래프는 우리가 흔히 아는 왼쪽과 같은 형태의 그래프가 아니라 오른쪽 모양의 그래프이고, 정확한 정의는 정점과 간선으로 이루어진 자료구조입니다.

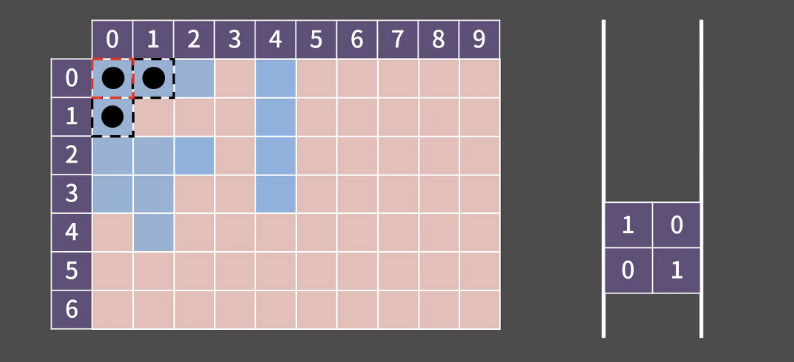
그래서 BFS를 엄밀하게 정의할 수는 없지만, 실제로 어떻게 동작하는지를 보면서 다차원 배열에서의 BFS를 이해해보도록 하겠습니다.



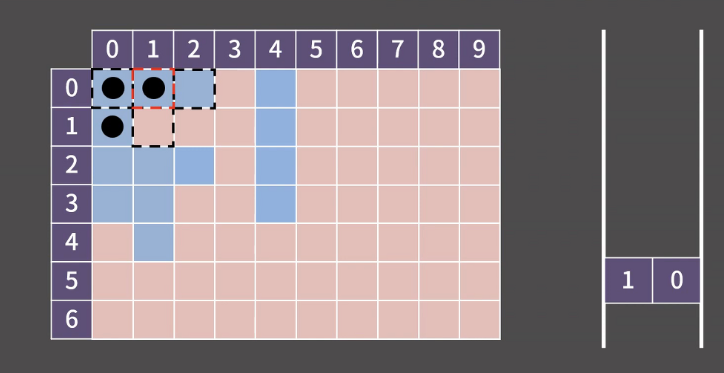
BFS 알고리즘에서는 좌표를 담을 큐가 필요합니다. BFS 알고리즘이 시작되면 우선 (0, 0)에 방문했다는 표시를 남기고 해당 칸을 큐에 넣습니다. 이 초기 세팅이 끝난 후에는 큐가 빌 때까지 계속 큐의 front를 빼고 해당 좌표의 상하좌우를 살펴보면서 큐에 넣어주는 작업을 반복하게 됩니다.



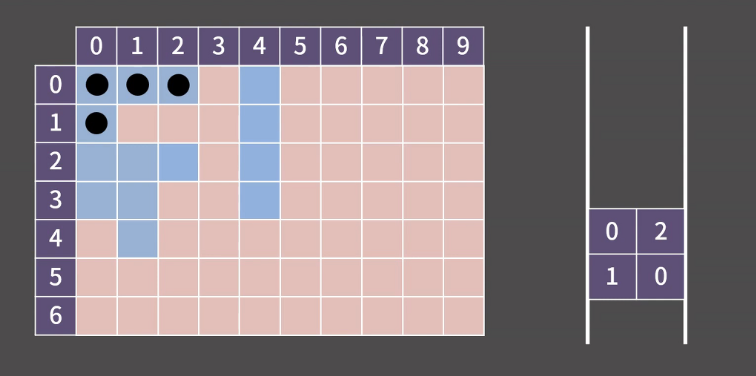
지금 상황에서 큐의 front는 (0, 0)이고 pop을 합니다. 그리고 (0, 0)의 상하좌우 칸을 보는데, 이 중에서 우리는 파란색 칸이면서 아직 방문하지 않은 칸을 찾을 것입니다.



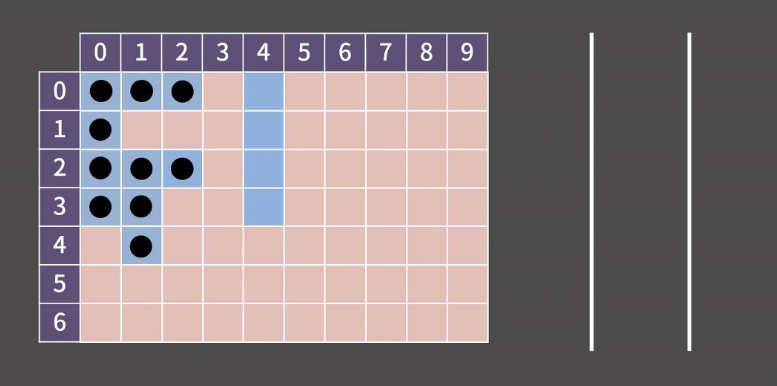
지금 상황을 보면 (0, 0)과 상하좌우로 인접한 (0, 1)과 (1, 0)은 모두 파란 칸이면서 아직 방문하지 않았습니다. 이 2개의 칸에 방문했다는 표시를 남기고 큐에 넣습니다.



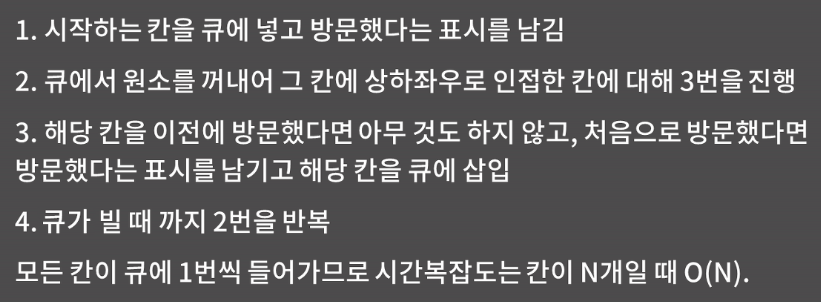
(0, 0)에서 할건 다 했고, 다음으로 넘어갑니다. 현재 큐의 front는 (0, 1)이고 pop을 합니다. 참고로 (0, 1)에서 0은 행을 의미하고 1은 열을 의미합니다. 그리고 이번에도 (0, 1)의 상하좌우 칸을 확인합니다. 이 칸들 중에서 (0, 0)은 파란 칸이지만 이미 방문을 했고, (1, 1)은 빨간 칸입니다. 유일하게 (0, 2)만 파란색 칸이면서 아직 방문하지 않은 칸이니까 (0, 2)에 방문했다는 표시를 남기고 큐에 넣습니다.



생략

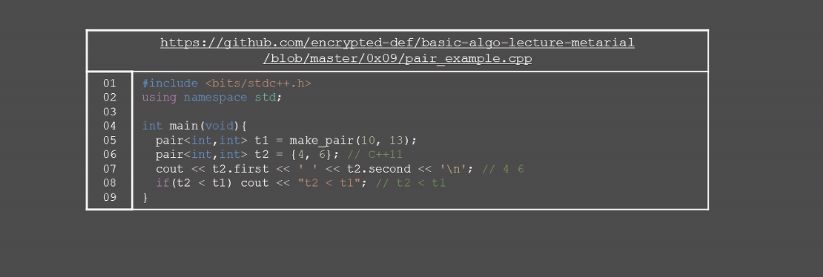


이렇게 큐가 빈 순간 과정은 종료되고 (0, 0)과 상하좌우로 이어진 모든 파란 칸을 잘 방문했음을 알 수 있습니다.



일단 시작하는 칸을 큐에 넣고 방문했다는 표시를 남깁니다. 그리고 큐가 빌 때까지 큐에서 원소를 꺼내고 상하좌우로 인접한 칸에 대해 처음으로 방문했다면 해당 칸을 큐에 삽입하는 것을 반복하면 됩니다.

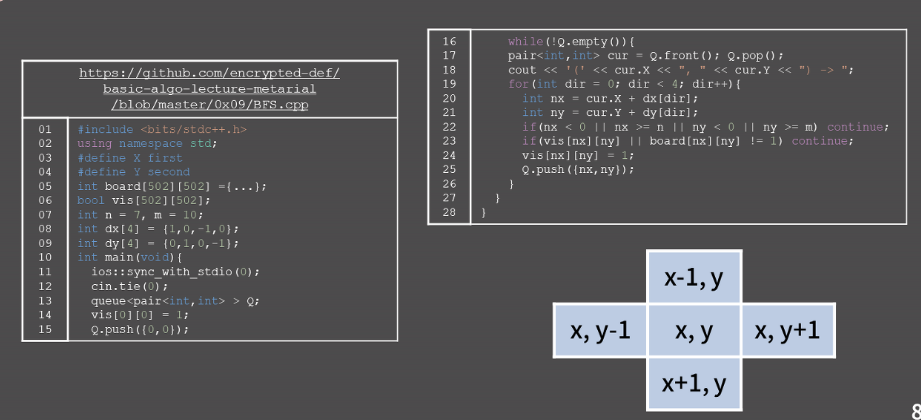
BFS의 시간복잡도를 생각해보면 방문 표시를 남기기 때문에 모든 칸은 큐에 1번씩만 들어가게 됩니다. 그렇기 때문에 시간복잡도는 칸이 N개일 때 O(N)이 됩니다. 만약 행이 R개이고 열이 C개이면 O(RC)가 될 것입니다.



Pair 소개

알아서 앞쪽의 값을 먼저 비교하고, 이후 뒤쪽의 값을 비교합니다.

BFS를 구현할 때 큐에 좌표를 넣어야 하는데, 이때 pair를 쓸 것입니다.



BFS는 어느 정도 정석적인 구현이 있어서 지금 보여드릴 코드를 거의 외우다시피 해도 괜찮습니다. 특히 삼성 A형을 치기 위해서는 BFS가 정말 숙달되어 있어야 하는데, 어느 정도냐면 자고 있다가 누가 툭 쳐서 BFS를 짜라고 시켜도 한 5분 내로 기본 틀을 좌르륵 쳐낼 수 있어야 합니다.

 03, 04번째 줄에서 #define을 해놓은 건 pair를 조금 더 편하게 쓰기 쓰기 위함인데, first/second 대신 t.X/t.Y로 쓰고 싶어서 저렇게 했습니다.

board는 말 그대로 판을 의미합니다. 1이면 파란 칸이고 0이면 빨간 칸이라고 생각하면 됩니다. 그리고 vis는 방문 여부를 저장할 변수입니다. 칸 위에 올리던 동그라미를 코드 상에서는 vis 값을 1로 변경함으로써 처리하는 것입니다. 그리고 n과 m은 각각 행과 열의 개수를 의미하고, dx와 dy는 상하좌우를 영리하게 처리하기 위한 변수인데 이건 실제로 쓰일 때 설명을 드리겠습니다.

그 후로 (0, 0)에 방문 표시를 하고 큐에 추가하는 과정이 코드에서는 14, 15번 줄이 됐습니다. 그다음에는 큐가 빌 때까지그 상하좌우의 칸을 추가하는걸 쭉 반복하는데, 일단 큐의 front를 cur에 저장하고 pop을 합니다. 그다음 줄은 그냥 방문 순서를 나타내 주기 위해서 넣은 출력문입니다.

19번째 줄에서 26번째 줄까지의 루틴이 좀 중요한 루틴입니다. 아까 정체를 알 수 없던 dx, dy가 여기서 쓰입니다. 판에서 저희가 상하좌우를 생각해보면 오른쪽 아래의 그림과 같습니다. 그런데 노파심에 말을 하자면, 이 코드에서 x가 행을, y가 열을 의미합니다. 그런데 사람에 따라 x가 열이고 y가 행인 게 더 익숙할 수 있습니다. 이게 별게 아닐 수 있지만 생각을 할 때 굉장히 헷갈리게 하는 요소가 될 수 있습니다. 당장 지금 저 그림도 만약 x가 열이고 y가 행이라면 왼쪽 칸이 x-1, y가 되어야 합니다. 뭐 어떻게 하든 답은 잘 나오겠지만 저 뿐만 아니라 대부분의 BFS 코드에서 x가 행을, y가 열을 의미하는 경우가 많았어서 이 방식으로 따라오시는게 좋을 것 같습니다.

아무튼 우리는 (cur.X, cur.Y)에 대해 상하좌우 칸인 (cur.X-1, cur.Y), (cur.X, cur.Y-1), (cur.X, cur.Y+1), (cur.X+1, cur.Y)를 확인할 필요가 있습니다. 그리고 이걸 쉽게 하는 방법이 바로 19번째 줄부터 이어지는 for문인데, nx와 ny의 값을 같이 봅시다. 보면 cur.X, cur.Y에 dx[dir], dy[dir]을 각각 더하죠? 그리고 08, 09번째 줄로 시선을 돌려보면 dx[dir], dy[dir] 값이 (1, 0), (0, 1), (-1, 0), (0, -1)이니까 nx, ny에 상하좌우의 좌표값이 깔끔하게 담긴다는 것을 알 수 있습니다. dx, dy 값에 따른 순서는 아랫쪽, 오른쪽, 윗쪽, 왼쪽 순서이긴 한데 이건 크게 중요하지는 않습니다.

이후 22번째 줄에서 일단 범위에 들어오는지를 확인하고, 23번째 줄에서 이미 방문했거나 파란 칸이 아닌 경우를 걸러내고 난 후에는 방문했다는 표시를 vis[nx][ny] = 1로 바꿈으로서 남기고 큐에 추가해주면 됩니다. 만약 22번째 줄과 23번째 줄의 순서가 바뀌면 어떻게 될까요? 그렇게 되면 vis[-1][0]과 같은 값을 참조할 수 있어서 런타임에러가 날 수 있습니다. 그렇기 떄문에 일단 범위 내에 들어오는지를 꼭 먼저 확인한 후에 vis, board 배열값을 봐야합니다.

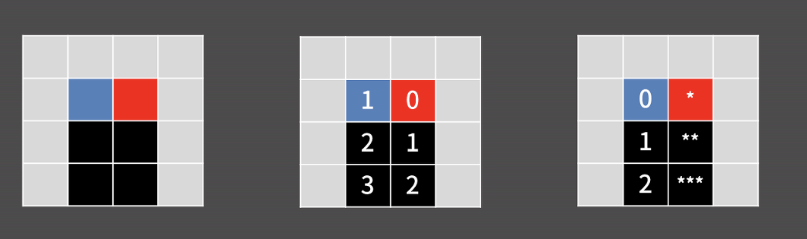
응용1- 거리측정

솔루션) vis[현재] = vis[이전] +1

응용2- 시작점이 여러 개일 때

솔루션) 모든 시작점을 큐에 넣고 BFS를 시작한다.

응용3- 시작점이 두 종류일 때



4179불

 이런 문제는 불에 대한 BFS와 지훈이에 대한 BFS를 모두 돌림으로서 해결이 가능합니다.

먼저 지훈이는 신경쓰지 말고 불에 대한 BFS를 돌려서 미리 각 칸에 불이 전파되는 시간을 다 구해둡니다. 두 번째의 맵이  바로 각 칸에 불이 전파시간을 의미합니다.

그 다음에는 지훈이에 대한 BFS를 돌리며 지훈이를 이동시킵니다. 이 때 만약 지훈이가 특정 칸을 x시간에 최초로 방문할 수 있는데 그 칸에는 x시간이나 그 이전에 불이 붙는다면 그 칸을 못가게 됩니다.

예를 들어 \*\*으로 마킹한 칸을 보면 지훈이는 저 칸에 2시간이 될 때 방문하게 됩니다. 그런데 불은 이미 1시간만에 전파되었기 때문에 지훈이는 저 곳을 갈 수 없습니다. \*, \*\*\*으로 마킹한 칸도 마찬가지 이유로 지훈이가 갈 수 없는 칸입니다.

원래 BFS의 구현에서는 큐 안에서 (nx, ny)를 살펴볼때 방문했는지 여부를 vis[nx][ny]가 true인지 혹은 dist[nx][ny]가 0 이상인지 확인하고, 이미 방문한 칸이라면 continue를 합니다. 이 문제에서는 추가로 해당 칸에 불이 붙은 시간을 확인해서 필요에 따라 continue를 하면 됩니다. 이렇게 BFS와 지훈이에 대한 BFS를 따로 해서 문제를 그렇게 어렵지는 않게 해결할 수 있습니다.

이렇게 시작점이 두 종류인 문제를 해결할 수 있게 됩니다. 그런데 시작점이 두 종류인 문제에 관해서 저희가 생각해야 할 점이 사실 추가로 있습니다. 지금 이 방식이 가지고 있는 문제는 무엇인가 하면, 지금은 지훈이의 이동은 불의 전파에 영향을 받지만 불의 전파는 지훈이의 이동에 영향을 받지 않아서 불만 먼저 전파를 쭉 시키는게 가능했습니다. 그런데 예를 들어 시작점이 A, B 두 종류가 있고, A의 전파에 B가 영향을 주고 B의 전파에도 A가 영향을 준다고 해봅시다. 지금 이 문제에서 불과 지훈이가 아니라 불과 소방수 내지는 불과 물이 전파되는 문제여서 둘이 만나면 뭔가 상호작용이 발생한다고 생각을 하는거죠.

그런 상황을 생각해보면 어느 하나를 먼저 끝까지 전파시키는게 불가능합니다. 제가 출제했던 [18809번 문제](https://www.acmicpc.net/problem/18809)가 딱 그런 문제입니다. 아쉽게도 이 문제는 백트래킹 기법을 추가로 알고 있어야 해결이 가능하기 때문에 당장 풀어볼 수는 없지만, 두 종류의 BFS에서 BFS를 돌 때 어느 하나가 독립적이지 않고 서로에게 영향을 준다면 지금 보여드린 방법으로는 해결할 수 없다는 것을 꼭 이해하셔야 합니다. 그런 상황에서는 시간 순으로 A와 B를 동시에 진행시켜야 합니다.

응용4- 1차원에서 BFS

상하좌우 -> -1, +1, \*2로 바뀐 것뿐

BFS의 범위를 어디까지로 해야할까요? 0부터 100,000으로 하면 되는거 아닌가하고 쉽게 생각을 하셨을수도 있는데, 문제를 보시면 수빈이와 동생의 위치가 0에서 100,000 사이라고 했지 수빈이가 이동 중에 반드시 0에서 100,000 사이에만 있어야한다는 조건은 없습니다. 예를 들어 100,000 밖으로 나갔다가 다시 안으로 올 수도 있습니다. 그래서 이 부분을 고려할 필요가 있습니다.

일단 상식적으로 생각했을 때 음수로 갈 일은 없을 것입니다. 그건 진짜 절대 가장 빠른 경로가 될 수 없기 때문입니다. 그리고 100,000 바깥으로 나갈 수 있겠지만 일단 한 번 나갔다면 그 이후로는 -1만 계속 하게 됩니다. 그렇기 때문에 동생을 가장 빠르게 찾아나가는 상황에서는 아무리 멀리가도 200,000을 넘어가지는 않습니다.

이러한 생각을 거쳐서 0에서 200,000 사이에서만 BFS를 돌려도 답을 구하는데는 문제가 없음을 알 수 있게 되고, 여기서 더 깊게 생각을 해보면 사실 100,000을 나가는 것 자체가 손해라는 것을 알 수 있습니다. +1로 100,000을 탈출하는건 정말 바보짓이고, x2로 100,000을 탈출하는 상황이 있을 수 있겠다 싶지만, x2를 한 후 -1을 여러번 할 바에야 -1을 먼저 하고 x2를 하는게 더 낫기 때문입니다.

지금 설명이 잘 이해가 가지 않을 수 있습니다. 이해가 가지 않더라도 상관은 없지만 이 문제에서 당연히 수빈이가 0에서 100,000 사이에서만 움직인다고 멋대로 가정을 하고 풀면 안된다는 점은 꼭 짚고 넘어가면 좋겠습니다. 이 문제에서는 운 좋게 논리적으로 생각을 했을 때 수빈이가 0에서 100,000 사이에서만 움직이게 되었지만 다른 문제에서는 멋대로 가정한 것 때문에 말아먹을수도 있습니다.