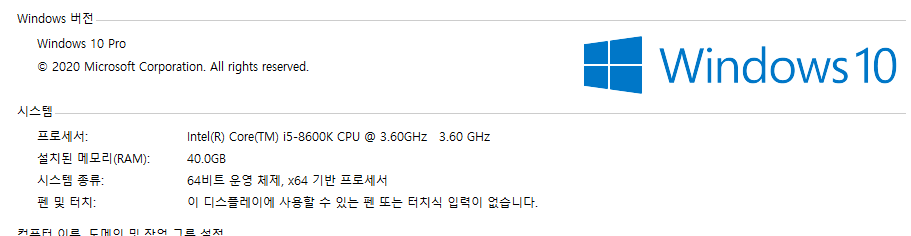
1. PC 환경



제약 조건. 배열의 크기가 100 이상 정도부터 유의미한 차이가 발생합니다.

2. 첫번째 방법으로 완전탐색을 이용해 최대 구간과 합을 구현했습니다. N의 개수를 가지고 있는 배열일 경우, 첫번째 반복문은 첫 인덱스부터 마지막 인덱스까지 반복 실행하며 두번째 반복문은 첫번째 반복문의 인덱스부터 시작해 마지막 인덱스까지 반복 실행하며 값을 확인합니다. 반복문을 실행하면서 ret 변수의 값보다 높을 때마다 ret, left, right 구간의 값을 갱신합니다. 시간 복잡도는 O(n^2)입니다.

두번째 방법으로는 이분탐색을 이용해 최대 구간과 합을 구했습니다. 배열을 left right로 값을 주고 절반으로 나눠 밑부분인 인덱스 값(left == right)에 도착하면 다시 올라가면서 값을 병합합니다. 기저조건인 left == right의 경우 해당 인덱스값을 반환합니다. 이외의 경우엔 mid의 값부터 좌측과 우측으로 인덱스를 증감시키며 현재의 left, right에 해당하는 최대값과 구간을 얻고, 왼쪽 구간과 오른쪽 구간의 이분탐색을 통해 얻었던 값과 비교하여 가장 높은 값을 반환합니다. 시간 복잡도는 절반씩 나눠서 들어가는 logN의 값과 해당 left, right에서 값을 얻기위한 N값이 곱해져 O(NlogN)입니다.

마지막 방법은 예를 이용해 최대 구간과 합을 구현했습니다. 이분탐색을 통해 구현하다가 메모리제이션을 이용해 풀면 더 빠른 시간을 얻을 수 있을거라 생각해 구현했습니다. DP[i] = i의 인덱스일 때 최대값을 의미하며 점화식은 DP[i] = max(DP[I – 1] + vec[i], vec[i]) 입니다. DP의 단점은 공간을 N만큼 더 할당 받아야 한다는 점이고 장점으로는 시간복잡도가 N번만 반복하면 되기 때문에 O(N)이라는 점입니다.

5.

image\_sum\_value 함수는 이중반복문을 이용해 모든 값을 더해주었습니다 시간 복잡도는 O(N^2)입니다.

Integral\_image 함수는 마찬가지로 이중 반복문을 이용해 값을 더해주되, ii\_image[i]의 값은 image[i]의 값과 width < i && height < i 값의 부분을 더해주었습니다. 적분영상을 얻기 위해 사용되는 시간복잡도는 O(N^2)이며 공간은 추가적으로 N이 필요합니다.

Intergral\_image\_sum\_value의 함수는 적분영상을 이용해 해당 영역의 합을 구해주는 함수입니다. 구하고자 하는 left, top, right, bottom의 값이 있을 때 ii\_image[bottom][right] + ii\_image[top - 1][left] – ii\_image[top][right] – ii\_image[bottom][left - 1] 으로 합을 결정합니다. 위의 식에서 뺄셈의 이유는 ii\_image[y][x]의 값은 (0, 0)의 좌표에서 (y, x)의 좌표까지의 값을 의미하는데, top과 left의 값은 0, 0이 아니기 때문에 빼주게 됩니다. (ii\_image[top -1][left] – ii\_image[bottom][left -1]) 이 때 겹치는 부분인 ii\_image[top -1][left]를 해줌으로 써 겹치는 부분을 보완해 줍니다.

적분영상의 장점으로는 구간의 합을 요청하는 함수가 잦을 경우에 유의미한 시간복잡도를 얻게 됩니다. 일반합의 경우 요청할 때마다 O(N^2)가 소요되지만, 적분영상을 이용한 합은 적분영상을 생성할 때를 제외하고 O(1)의 시간복잡도를 가지기 때문입니다.

ii\_image의 값이 overflow가 발생하는 상황은 ii\_image의 자료형으로 주어진 unsigned int의 범위를 벗어나는 경우라고 생각합니다. Ii\_image는 i\_image의 값으로 생성되는데, i\_image의 값은 rand() % 256의 값으로 구성되어 있습니다. Rand() 함수는 랜덤함수이지만, 컴파일러에 따라 고정적인 값을 할당합니다. clang version 7.0.0-3의 기준 4565의 값이 최대값입니다. Ii\_image를 생성하면서 ii\_image[i] = ii\_image[I – 1] + i\_image[i]으로 구성되는데, ii\_image[i] < ii\_image[I – 1]의 조건이 참이 된다면 overflow를 반환함으로써 확인하였습니다.

6.

함수 설명

|  |  |
| --- | --- |
| Set\_Cost\_Mode | Cost의 값을 받고 Mode를 결정하는 함수(0 일 경우 최소 최적값, 1 일 경우 최대의 최적값을 선택한다.) |
| Solve | Cost의 크기에 맞게 Mat을 동적할당 & 알고리즘 시작 함수 |
| Make\_maximum\_mat  Make\_minimum\_mat | Mode 값에 따라 Cost값을 Mat에 할당하는 함수(Step 1), Minimum일 경우 각 행과 열의 최솟값을 현재값에서 빼주며, Maximum일 경우 각 행은 Max 값 현재 값으로 빼주며 각 열은 현재값에서 최솟값을 빼준다. |
| Mat\_change | 헝가리안 알고리즘이 완성될 때까지 Step을 반복하는 함수 |
| Is\_vaild | Mat이 헝가리안 알고리즘의 조건에 부합하는지 확인하는 함수 Mat의 모든 0을 n개의 직선으로 cover가 되는지 확인한다. |
| Step3\_assign\_mat | 1. 각 row에 0이 존재한다면, 해당 인덱스의 row와 column에 cross 처리를 하고 assigned 한다. 2. 만약 0이 있지만 cross처리가 되어 있다면 해당 0은 넘어간다. (step3-1) |
| Step3\_Mark | 1. 각 row가 assigned 되어있지 않다면 Mark 한다. 2. Mark된 row를 대상으로 column에 0값이 존재하면 Mark 한다. 3. Mark된 column을 대상으로 row 가 assigned 되어 있다면 Mark 한다. 4. 새롭게 생기는 Mark가 없을 때 까지 반복한다.   (step3-2) |
| Step4\_Mat\_Change | 1. Row는 unMark된 값들을 draw 하고,column은 mark된 값들을 draw 한다. 2. Cross 되어 있지 않은 값 중 0을 제외한 최솟값을 찾는다. 3. Two line으로 draw 된 값은 최솟값을 더해준다. 4. One line 으로 draw 된 값은 연산을 하지 않는다. 5. Non draw 값들은 최솟값을 빼준다. |
| Print\_Answer() | 정답을 출력한다. 단, N과 M의 사이즈가 다를 경우를 고려해 assigned\_index에서 Ori\_N인덱스 까지만 출력한다. |

Step1에 해당하는 함수인 Make\_mat 함수는 각 행과 열을 탐색하고 연산하며, O(N^2)의 복잡도를 가집니다.

Step3 에 해당 하는 함수인 assign\_mat 함수는 Make\_mat 함수와 동일하게 O(N ^ 2)의 복잡도를 가지며, Mark 함수는 새로운 Mark가 생길 때까지 반복하게 되지만, 최대로 생길 수 있는 Mark의 수가 N개로 한정되어 있고 한번 Mark함수가 실행될 때, O(N ^ 2)의 시간복잡도를 가지게 되어 동일합니다.

Step4 의 함수인 Mat\_Change 함수는 수정된 Mat값을 최솟값으로 수정해주는 함수로, 최솟값을 찾는 반복문, 최솟값으로 수정하는 반복문으로 동일하게 N ^ 2의 시간복잡도를 가집니다.

Is\_vaild 함수에서 Mat의 값이 조건에 부합하면 종료하게 되는데, 총 O(N ^ 4)의 시간 복잡도가 필요합니다.

Step 1 ~ 4 O(n ^ 2), is\_vaild로 인한 O(n ^ 2) 반복으로 총 O(n ^ 4)의 시간 복잡도를 가지게 됩니다.