**컴퓨터 알고리즘과 실습**

1. 4회차 실습 -

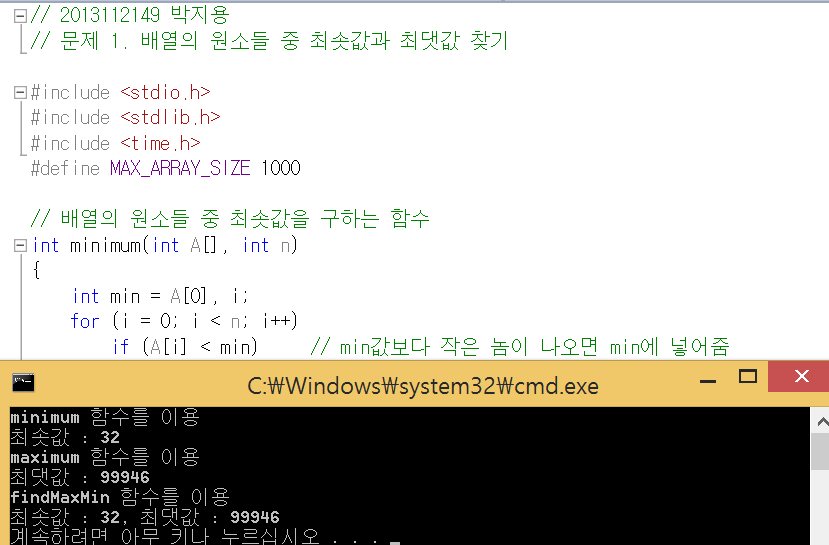
2017.03.29

컴퓨터공학과

2013112149 박지용

1. **실습문제 1번. 배열의 원소들 중 최솟값과 최댓값 찾기**

<출력 결과>



문제 1번의 출력화면입니다. 먼저 size 1000의 배열을 생성하고 rand()함수를 이용해 배열의 각 인덱스에 1에서 100000 사이의 랜덤한 값을 저장하였습니다. 또한 srand()함수를 이용해 rand()함수로 나오는 랜덤값이 매 시각 변할 수 있도록 하였습니다. 이 배열을 input으로 배열의 원소들 중 최솟값을 구하는 minimum함수를 호출해 최솟값 32를 구한 것을 볼 수 있습니다. 또한 배열의 원소들 중 최댓값을 구하는 maximum 함수를 호출해 최댓값 99946을 구했고 최솟값과 최댓값을 동시에 구할 수 있는 findMaxMin함수를 이용해 minimum함수에서 얻은 최솟값과 maximum함수에서 얻은 최댓값과 동일한 값을 동시에 구하는 것을 확인하였습니다. rand()함수를 통해 나온 값은 short형의 최댓값인 32767 이상을 넘을 수가 없는 것으로 보여서 소스코드 상에서 rand 출력을 100000까지 할 수 있도록 변화를 주었습니다.

<소스코드>

// 2013112149 박지용

// 문제 1. 배열의 원소들 중 최솟값과 최댓값 찾기

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_ARRAY\_SIZE 1000

// 배열의 원소들 중 최솟값을 구하는 함수

int minimum(int A[], int n)

{

int min = A[0], i;

for (i = 0; i < n; i++)

if (A[i] < min) // min값보다 작은 놈이 나오면 min에 넣어줌

min = A[i];

return min;

}

// 배열의 원소들 중 최댓값을 구하는 함수

int maximum(int A[], int n)

{

int max = A[0], i;

for (i = 0; i < n; i++)

if (A[i] > max) // // max값보다 큰 놈이 나오면 max에 넣어줌

max = A[i];

return max;

}

int min\_value = 0, max\_value = 0; // 최솟값을 저장할 min\_value와 최댓값을 저장할 max를 전역변수로 선언

// 배열의 원소들 중 최솟값과 최댓값을 동시에 구하는 함수

void findMaxMin(int A[], int n)

{

int i;

min\_value = max\_value = A[0];

for (i = 0; i < n; i++)

{

if (A[i] < min\_value)

min\_value = A[i];

if (A[i] > max\_value)

max\_value = A[i];

}

}

int main(void)

{

int i, n = MAX\_ARRAY\_SIZE;

int A[MAX\_ARRAY\_SIZE];

srand((unsigned)time(NULL)); // rand 함수에서 나오는 값들에 변화를 주기 위함

for (i = 0; i < n; i++)

A[i] = (int)((rand() << 15) | rand()) % 100000 + 1; // A배열의 각 원소들에 1 ~ 100000 사이의 랜덤한 값을 넣어줌

printf("minimum 함수를 이용\n최솟값 : %d\n", minimum(A, n));

printf("maximum 함수를 이용\n최댓값 : %d\n", maximum(A, n));

findMaxMin(A, n);

printf("findMaxMin 함수를 이용\n최솟값 : %d, 최댓값 : %d\n", min\_value, max\_value);

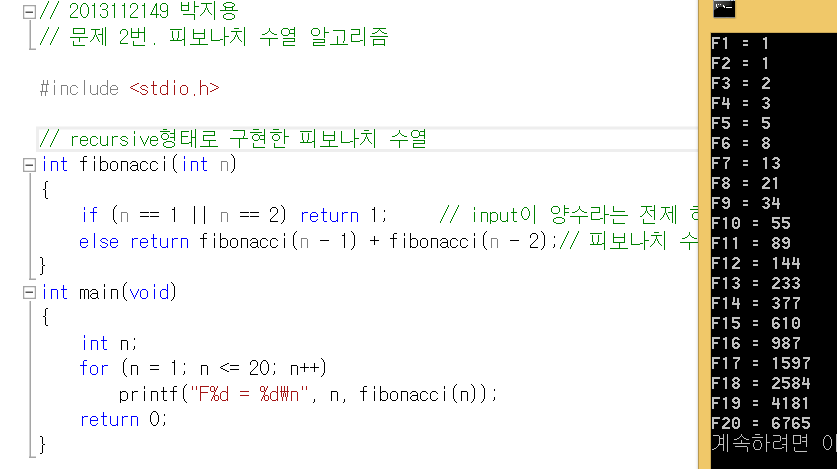
return 0;

}

문제 1번의 소스코드입니다. 먼저 minimum 함수를 살펴보면 배열의 원소들을 맨 앞에서부터 맨 뒤까지 index를 1씩 올리면서 차례차례 비교해서 작은 값을 저장하는 식의 아이디어로 최솟값을 찾아냈습니다. maximum함수도 동일한 과정을 거쳐 큰 값을 저장하는 식으로 최댓값을 찾아냈습니다. findMaxMin 함수에서는 이 과정에서 작은 값을 저장하는 min\_value 변수와 큰 값을 저장하는 max\_value 변수를 함께 두어 최솟값과 최댓값을 동시에 구했습니다. 함수에서 이 둘을 모두 반환할 수는 없으므로 전역변수 혹은 포인터를 사용하는 방법이 있는데 간편하게 min\_value와 max\_value를 전역변수로 사용하여 문제를 해결하였습니다.

1. **실습문제 2번. 피보나치 수열 알고리즘**

<출력 결과>



문제 2번 출력화면입니다. 피보나치 수열 공식을 이용해서 n번째 피보나치 수의 값을 계산하여 반환하는 fibonacci함수의 input을 1에서 20까지 줘서 1번째부터 20번째까지의 피보나치 수를 출력하였습니다.

<소스코드>

// 2013112149 박지용

// 문제 2번. 피보나치 수열 알고리즘

#include <stdio.h>

// recursive형태로 구현한 피보나치 수열

int fibonacci(int n)

{

if (n == 1 || n == 2) return 1; // input이 양수라는 전제 하에 0이하의 수에 대해서 고려하는 코드는 작성하지 않음

else return fibonacci(n - 1) + fibonacci(n - 2); // 피보나치 수열의 정의

}

int main(void)

{

int n;

for (n = 1; n <= 20; n++)

printf("F%d = %d\n", n, fibonacci(n));

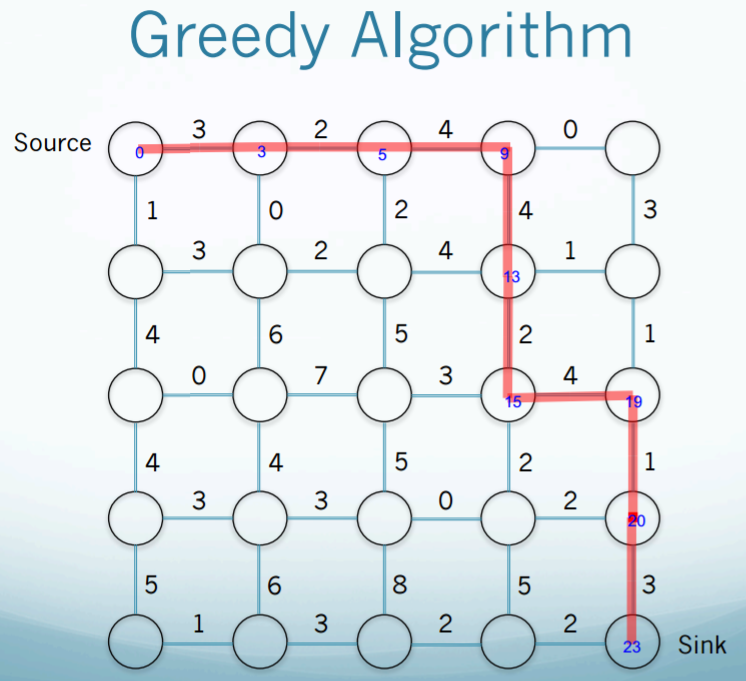
return 0;

}

문제 2번 소스코드입니다. 피보나치 수열의 정의를 이용하여 recursive(순환) 형태로 fibonacci 함수를 구현하였습니다. 피보나치 수열의 첫번째 수와 두번째 수는 1이므로 n = 1 or 2 일 경우 1을 return시켜줬고 나머지 경우에는 재귀문으로 처리하였습니다. 그리고 main함수에서 input을 1에서 20까지 주면서 fibonacci함수를 호출해 20개의 피보나치 수를 구했습니다.

1. **실습문제 3번. Manhattan Tourist Problem**
2. **by Greedy 알고리즘**

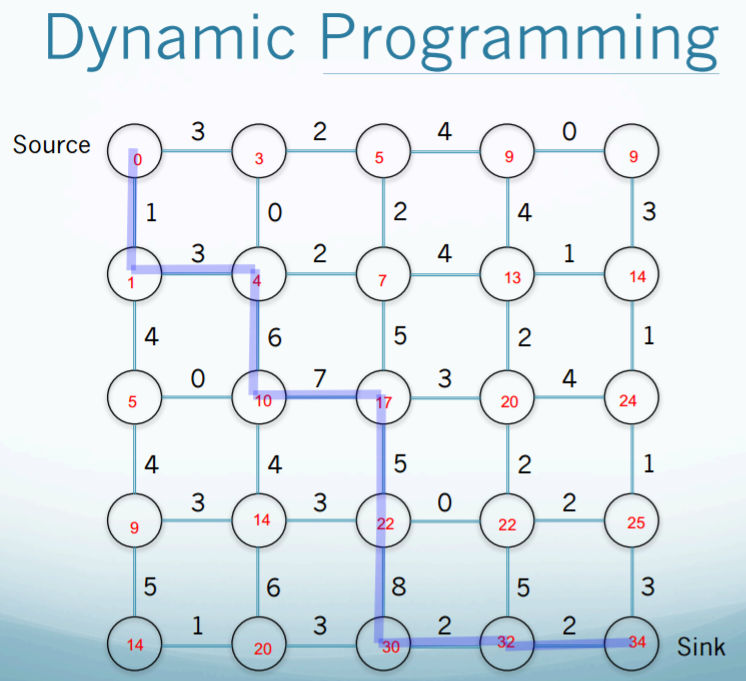
<경로>



오른쪽과 아래쪽으로만 향할 수 있다는 문제의 기본 전제에 따라 현재의 정점에서 오른쪽과 아래쪽 길의 가중치를 비교하여 가중치가 높은 쪽으로 향하는 Greedy 알고리즘을 이용하여 맨 처음 정점(시작점)부터 가중치가 높은 쪽으로 향했더니 (2, 4)정점(시작 지점을 (0, 0)이라고 했을 때)에 도달하였습니다. 5열에 도달한 순간부터는 오른쪽 길이 없기 때문에 아래쪽으로만 2번 가고 도착점에 도달합니다. 정점 안에 파란색 글씨로 써놓은 것은 지금까지 거쳐간 길들의 가중치의 합, 즉, Manhattan을 여행하면서 tourist가 구경한 물건의 총 개수입니다. Greedy 알고리즘을 적용하여 빨간색 줄을 따라서 여행한 결과 23개의 물건을 구경할 수 있었습니다.

1. **by Dinamic Programming**

<경로>



이번에는 Greedy 알고리즘이 아닌 Dinamic Programming으로 여행한 경로입니다. 마찬가지로 정점 안에 있는 빨간색 글씨는 거쳐 간 가중치의 합, 즉, 구경한 물건들의 총 개수입니다. 더 정확히 말하면, 출발점에서 시작해서 각 정점까지 가면서 최대한 많이 볼 수 있는 경로로 향했을 때의 가중치의 합입니다.

각 정점에 대한 가중치의 합이 최대가 되게 하기 위한 기본 아이디어는 0행과 0열의 모든 정점들에 대해서는 길이 유일하므로 미리 가중치를 부여해놓고 (0, 0)에 있는 정점부터

(위쪽 정점의 가중치 + 위쪽 정점과 자신 사이의 가중치) 와

(왼쪽 정점의 가중치 + 왼쪽의 정점과 자신 사이의 가중치) 중에서 큰 값을 자신의 가중치로 부여하는 것입니다. 이런 식으로 반복하면 모든 정점에 대한 가중치를 알게 되고 도착 지점에 도달했을 때 Backtracking하여 위쪽이나 왼쪽 정점 중 가중치가 높은 정점을 따라가면 됩니다. 다만 1행(위에서 두번째 줄)이나 1열(왼쪽에서 두번째 줄)에 도달하였을 때부터는 왼쪽과 위쪽 길 중에 가중치가 높은 길을 따라가야 합니다. 이는 0행과 0열의 정점들은 오른쪽 길과 아래쪽 길의 가중치 값을 비교해서 나온 가중치가 유일한 길을 따라서 부여된 가중치이기 때문입니다.

결국 가장 많이 볼 수 있는 경로로 갔을 때 본 물건의 총 개수는 마지막 정점의 가중치인 34개 입니다. Greedy 알고리즘을 통해 본 23개보다 훨씬 많은 물건을 보았습니다.

<유사코드>

**- 각 정점에 가중치 부여**

for i ← 1 to 5 do

←



for j ← 1 to 5 do

←



for i ← 1 to 5 do

for j ← 1 to 5 do

← Big(, )



**- Back Tracking**

while(i ← 4 to 0 && j ← 4 to 0) do

if i=1 or j=1

if >



i←i-1

else



j←j-1

else

if >=



i←i-1

else

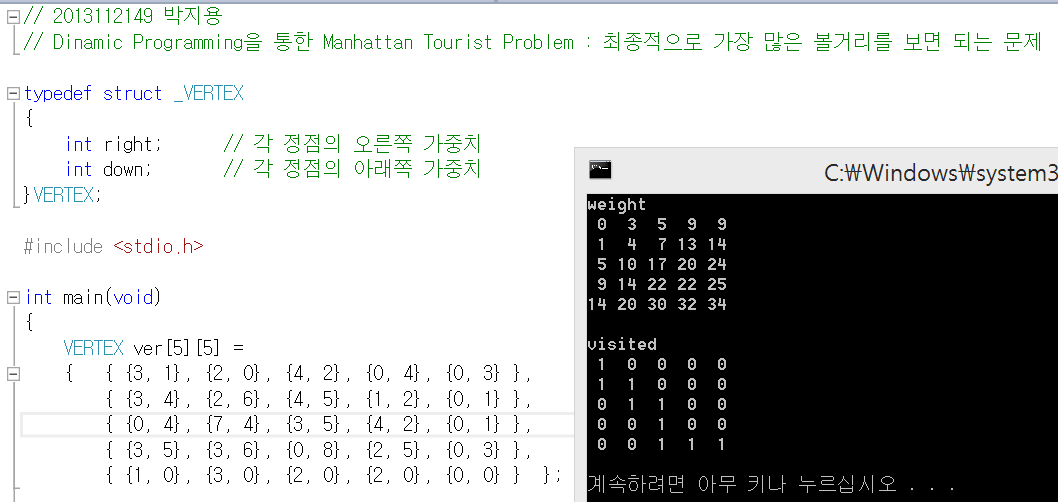


j←j-1

유사코드의 윗부분은 경로를 표시해 둔 그림 아래서 설명했던 것처럼 각 정점들마다 그곳까지 가는 데 가중치를 넣어주고 그것을 S라고 표기했습니다. W는 그 정점에서의 오른쪽 혹은 아래쪽을 나타내는 간선입니다. 그리고 Big( )이란 것은 둘 중에 큰 쪽을 고르겠다는 것을 의미합니다. 유사코드의 아랫부분은 모든 정점들에 가중치를 다 부여해 준 뒤에 Back Tracking해서 최적의 경로를 찾아주는 과정을 표현한 것입니다. visited는 최적의 루트로 갔을 때 각 정점의 방문여부를 저장하는 배열입니다.

Dinamic programming이 상당히 흥미로운 문제해결 방법이라서, 또 유사코드만으로 이해가 어려운 부분이 있을 것 같아서 직접 코딩을 해서 결과를 확인해보았습니다.

<출력화면>



weight가 위에 있는 경로 화면에서의 빨간색 숫자를 담은 배열(유사코드 상에서 S)이고 ver(ver[i][j].right, ver[i][j].down)이 각 간선들에 부여된 가중치입니다. visited는 가장 많은 물건을 본 경로를 지났을 때 각 정점의 방문 여부를 저장한 배열입니다. 초기값은 모두 0이며 방문하였으면 원소값을 1로 바꿔줍니다. 출력화면은 weight 배열과 visited 배열을 출력해서 위에서 봤던 그림과 가중치와 경로가 똑같이 나온 것을 볼 수 있습니다.

소스코드에 대한 자세한 설명은 주석으로 달아두었으니 생략하겠습니다.

<소스코드>

// 2013112149 박지용

// Dinamic Programming을 통한 Manhattan Tourist Problem : 최종적으로 가장 많은 볼거리를 보면 되는 문제

typedef struct \_VERTEX

{

int right; // 각 정점의 오른쪽 가중치

int down; // 각 정점의 아래쪽 가중치

}VERTEX;

#include <stdio.h>

int main(void)

{

VERTEX ver[5][5] =

{ { {3, 1}, {2, 0}, {4, 2}, {0, 4}, {0, 3} },

{ {3, 4}, {2, 6}, {4, 5}, {1, 2}, {0, 1} },

{ {0, 4}, {7, 4}, {3, 5}, {4, 2}, {0, 1} },

{ {3, 5}, {3, 6}, {0, 8}, {2, 5}, {0, 3} },

{ {1, 0}, {3, 0}, {2, 0}, {2, 0}, {0, 0} } }; // 각각의 정점은 정점의 {오른쪽 가중치, 아래쪽 가중치}로 정의

int weight[5][5] = { 0, }, visited[5][5] = { 0, }; // weight는 각 정점들의 가중치를 담는 배열, visited는 방문된 정점을 확인하기 위한 배열

int i, j, a = 0, b = 0;

weight[0][0] = 0; visited[4][4] = 1; // 첫번째 정점의 가중치는 0이어야하고 맨 마지막 정점은 반드시 방문되어야 함

for (i = 1; i < 5; i++)

weight[i][0] = weight[i - 1][0] + ver[i-1][0].down; // 0열에 있는 정점들의 가중치 부여

for (j = 1; j < 5; j++)

weight[0][j] = weight[0][j - 1] + ver[0][j - 1].right; // 0행에 있는 정점들의 가중치 부여

for (i = 1; i < 5; i++)

{

for (j = 1; j < 5; j++)

{

a = weight[i][j - 1] + ver[i][j - 1].right; // 왼쪽에서 오른쪽으로 가는 경우

b = weight[i - 1][j] + ver[i - 1][j].down; // 위에서 아래로 갈 경우

if (a > b)

weight[i][j] = a; // 오른쪽의 가중치를 더해준다

else

weight[i][j] = b; // 왼쪽의 가중치를 더해준다

}

}

i = 4; j = 4; // 맨 마지막 정점부터

while (i >= 0 && j >= 0) // 제일 앞 정점까지 Backtracking

{

if (i == 1 || j == 1) // 정점에 부여된 가중치(weigh)만 따라갈 경우에 발생하는 문제 해결(1행 이하 혹은 1열 이하 일 때만)

{

if (ver[i - 1][j].down > ver[i][j - 1].right) // 현재 정점의 위쪽 간선과 왼쪽 간선을 비교

{

visited[i - 1][j] = 1;

i--;

}

else

{

visited[i][j - 1] = 1; // 가중치가 더 큰 쪽으로 간다

j--;

}

}

else

{

if (weight[i - 1][j] >= weight[i][j - 1])

{

visited[i - 1][j] = 1;

i--;

}

else

{

visited[i][j - 1] = 1; // 맨 뒤에서 전 단계의 정점(위 or 왼쪽) 중 가중치가 큰 쪽으로 향한다

j--;

}

}

}

printf("weight\n");

for (i = 0; i < 5; i++)

{

for (j = 0; j < 5; j++)

printf("%2d ", weight[i][j]); // 정점에 부여된 가중치 출력

printf("\n");

}

printf("\n");

printf("visited\n");

for (i = 0; i < 5; i++)

{

for (j = 0; j < 5; j++)

printf("%2d ", visited[i][j]); // 방문된 위치들 출력

printf("\n");

}

printf("\n");

}