*ptr → the ~~address of~~ declaration of the pointer (declare pointer)

ptr = &num1; → ptr points to num1.

& is the memory address of the num1

num2 = *ptr + 5; Content
↳ the ~~address~~ of ptr

& delivers the address of a simple variable

* dereferences a pointer variable to obtain the simple variable to which it points.

```
int[] powers = { 1, 4, 9, 16, 25, 36 };
main()
{   int *ptr = powers;

    printf("addr %lx Contains %d \n", ptr, *ptr);
        ↳ addr 3000 Contains 1

    ptr = ptr + 4;

    printf("addr %lx contains %d \n", ptr, *ptr);
        ↳ addr 3010 Contains 25.
}
int ** results = malloc(size of(int *) * 20);
```

Use . notation to access the elements in a struct

Use → notation to direct access to a member of a struct referenced via a pointer.

A22

Struct stock_item *item_ptr;

item_ptr → Cost;

Struct stock_item {

char [21] name;

float Cost;

int number;

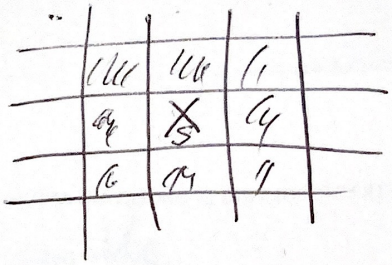
int min_level;

};

landscape → Search space

matrix → neighborhood

struct stock_item. cost



[examine the neighbor nodes one by one]

Simple Hill Climber:

Step 1: ~~Creat a random solution generator~~ evaluate the initial state. If it's a goal state then stop and return success. Otherwise, make inital state as Current state

Step 2: Loop until the Solution state is found or there are no new operators present which can be applied to the Current state.

a) Select a state that has not yet been applied to the Current state and apply it to produce a new state

b) Perform evaluation on new state

if the Current state is goal state. then stop and return success

if it's better than the current state, then make it current state and proceed further

if it's not better than the current state, then continue in the loop until a solution is found

Steepest Hill Climber:

Step 1: Evaluate the initial state. If it's a goal state then stop and return success. Otherwise, make initial state as current state

Step 2: Repeat these steps until a solution is found or current state does not change

a) Select a state that has not been yet applied to the current state

b) Initialize a new 'best state' equal to current state and apply it to produce a new state

c) evaluate new state

i) if the current state is a goal state, then stop and return success

ii) If it's better than best state, then make it best state else continue loop with another new state.

d) Make the best state as current state and go to step 2. (b). part.

[Examine all the neighboring nodes and selects the node closest to the solution state as of the next]

Stochastic Hill Climber

[does not examine all the neighbors before deciding which node to select. Just select a neighbor at random]

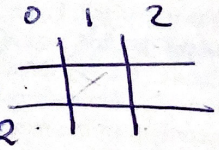
Among the generated neighbors which are better than the current state choose a state randomly.

You need a function to swap two points.

```
void swap(path_point &current, path_point &next)
{   Path_point temp = current;
    current = next;
    next = temp;
}
```

You need a function to find a set of neighbors in the current position

```
*Path_point* neighbors (Path_point &current)
{  Path_point *neighbors = malloc(sizeof(Path_point)
        * 8);
   for(int pos = 0; pos < 8; pos++)
      neighbors[pos] -> x = current -> x -1 +pos
      neighbors[pos] -> y = current -> y -1 +pos
   return neighbors;
}
```



(P1)

user_path
global_peak.
matrix.

```
Path_point find_highest_point()
{
  Path_Point *current_peak;

  generate_landscape(100000);
  // see what the landscape looks like before doing
  any hill climbing.

  Path_point *initial_state = random_select_point();

  while (!declare_peak(initial_state->x,
                       initial_state->y))

  generate_view(matrix, initial_state->y,
                initial_state->x);

  Path_point peek = neighbors_peak();
  Path_point **neighbors = Compare_neighbors(initial_state);

if (peak != null)
    Path_point **best = best_neighbor(neighbors);
    initial_state = best;
  else
    return peak;

}

return initial_state;

}
```

(P2)

```
function Path_point * random_select_point (~~float **matrix~~)
{
    int x_axis = random() % landscape_width;
    int y_axis = rand() % landscape_hight;
    Path_point * search_point = calloc (size of
        (Path_point));
    search_point -> x = x_axis;
    search_point -> y = y_axis;
    return search_point;
}
function int Compare_neighbors (Path_point * current,
                                Path_point * neighbor)
{ return ( current -> x <= neighbor <- x &&
           current -> y <= neighbor <- y ? 1 : 0);
}
function int** Compare_neighbors (Path_point * current)
{ int ** is_local_opts = malloc ( size of (int *) *
                                  size of (matrix[0]));
```

size of (int *) — size of a single row.

size of (matrix[0]) — no. of columns

```
    int no_rows = size of (matrix) / size of (matrix[0])
    int row, column;
    for (row = 0; row < no_rows; row++)
```

```
{
    for (column = 0;  column < sizeof(matrix[0]); column++)
    {
        Path_point * neighbor = matrix[row][column];
        is_local_opts[row][column] = compare_neighbor(
            current, neighbor);
    }
}
return is_local_opts;
}.
Path_Point neighbors_peak()
{ Path_point *local_peak;
    int cols = sizeof(matrix);
    int rows = sizeof(matrix) / sizeof(matrix[0]);
    int row, col;
    for (row = 0;  row < rows;  row++)
        for (col = 0;  col < cols;  col++)
            if (declare_peak(matrix[row][], matrix[][col],
                { local_peak = matrix[row][col];
                  break;
                }
}. return local_peak;
```

Path_point best_neighbor (int ** neighbor_flags) (Py)

```
{
  Path_point * best;
  Path_point ** best_Candidates; (1-D array of best Candidates)
  int cols = sizeof (matrix);
  int count = 0;
  int rows = sizeof (matrix) / sizeof (matrix[0]);
  int row, col;
  for ( row = 0; row < rows; row++)
    for ( col=0; col < cols; col++)
      if (neighbor_flags [row][col])
      { best_Candidate[count] = matrix[row][col];
        count++;
      }
  int ineighbor;
  for ( ineighbor =0; ineighbor < sizeof (best_Candidates)-1;
                                  ineighbor++)
    if (compare_neighbor (best_Candidates [ineighbor],
                          best_Candidates [ineighbor+1] ))
      best = best_Candidates [ineighbor+1];
  return best;
}
```