Chap 3. Stacks and Queues (2)

Contents

3.5 A Mazing Problem

- Rat in a maze
  - Experimental psychologists train rats to search mazes for food

- For us, a nice application of *stacks*
  - Searching the maze for an entrance to exit path.

Implementation in C

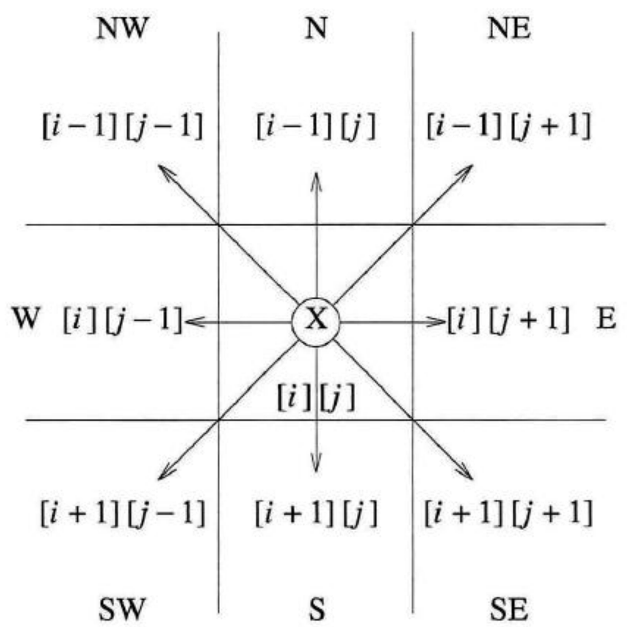
- Representation of a maze
  - *A two-dimensional array,* ***maze***
  - 0 : the open paths, 1 : the barriers

- Assumptions
  - Rat starts at the top left, exits at the bottom right

entrance	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	
	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	
	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	
	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0	
	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	
	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	0	1	1	1	1	1	0	1	
	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	
	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	
	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	exit

- The current location of the rat in the maze
  - *maze*[*row*][*col*]

- The possible *8 moves* from the current position



- Not every position has eight neighbors.
  - If [*row*, *col*] is on a border, then less than eight.

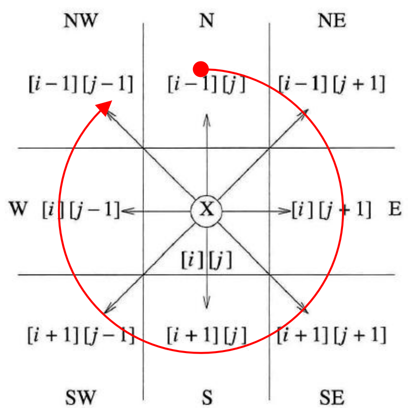
- To avoid checking for border conditions
  - We can surround the maze by a border of ones.

1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
1	0	1	0	0	1	1	1	1
1	0	0	1	0	1	1	1	1
1	1	1	1	1	0	0	1	1
1	0	1	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1
1	0	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1

< *m* × *p* maze >
(m+2) × (p+2) array, ***maze***
entrance : *maze*[1][1]
exit : *maze*[*m*][*p*]

- Predefining the possible directions to move, in an array ***move***

Name	Dir	*move[dir].vert*	*move[dir].horiz*
N	0	–1	0
NE	1	–1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	–1
W	6	0	–1
NW	7	–1	–1




```
typedef struct {
        short int vert;
        short int horiz;
        } offsets;
offsets move[8]; /* array of moves for each direction */
```

- The position of the next move, *maze[nextRow][nextCol]*

```
nextRow = row + move[dir].vert;
nextCol = col + move[dir].horiz;
```

- Since we do not know which choice is best,
  - we save our current position and
  - arbitrarily pick a possible move.

- By saving our current position,
  - we can *return to it* and *try another path* if we take a hopeless path.

- We examine the possible moves
  - starting from the north and moving clockwise.

- Maintaining a second 2D array, ***mark***
  - to record the maze positions already checked

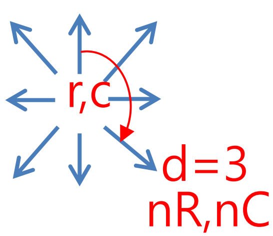
  - initialize the *mark*'s entries to *zero*
  - When we visit a position *maze*[*row*][*col*],

    we change *mark*[*row*][*col*] to *one*

현 위치 (r, c)에서 **탐색방향 d<8** 이고 **경로가 발견되지 않은** 한 다음을 반복

현 위치 **(r, c)** 에서 계산한 다음 위치 **(nR, nC)**에 대해

① if **출구인 경우**
경로발견!
② else if **이동가능하고 이전에 방문하지 않은 경우**
push(백트래킹 후 탐색할 위치와 방향) // push(r, c, d++)
다음위치 방문했음을 표시
다음위치로 이동
③ else
탐색방향증가 // d++



Q1. 언제 push를 수행하는가?

Q2. 언제 pop을 수행하는가?

Maze Search : Example

maze

entrance [1,1]

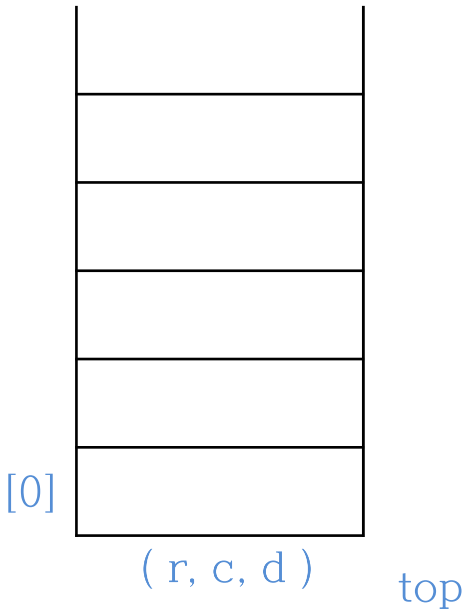
0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

mark

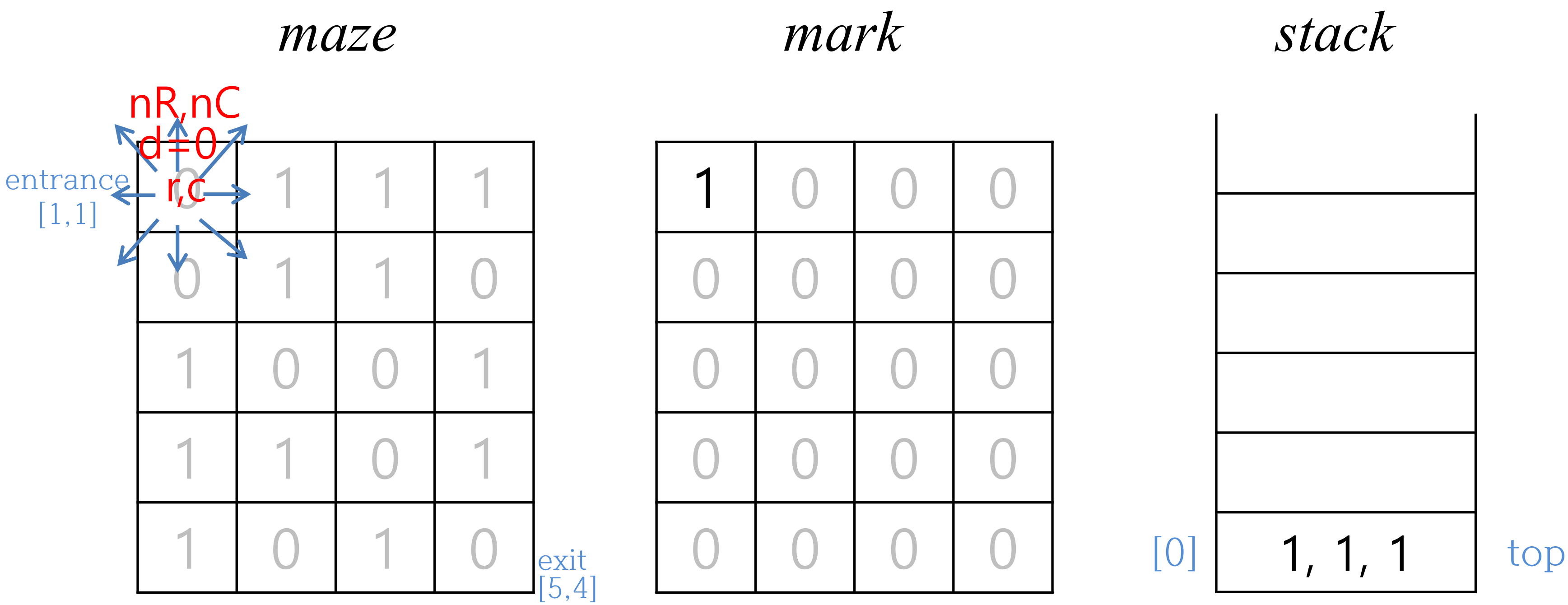
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack



Program initialization

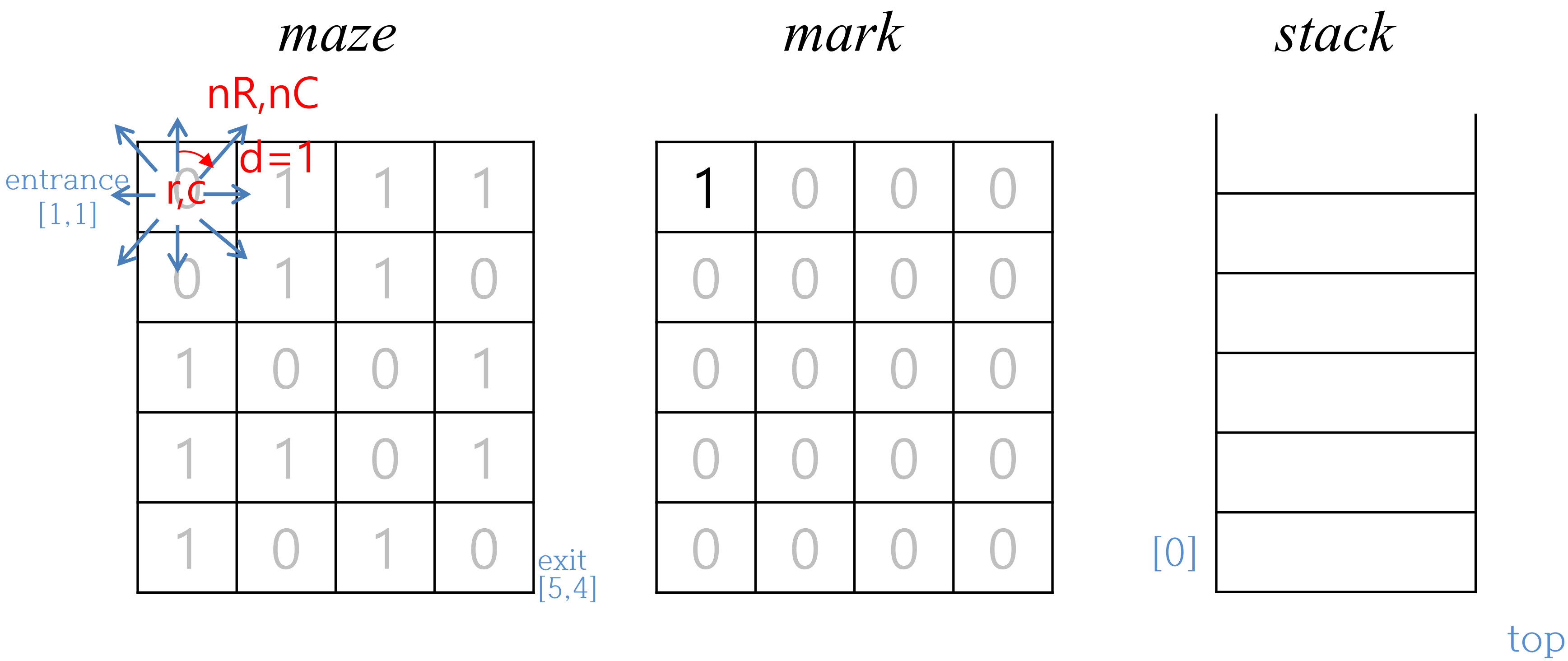
Maze Search : Example



Initialization of function path()

(1, 1) 방문
push(1, 1, 1)

Maze Search : Example



pop()
(r, c, d) = (1, 1, 1)

Maze Search : Example

maze

entrance [1,1]

r,c

d=2

nR,nC

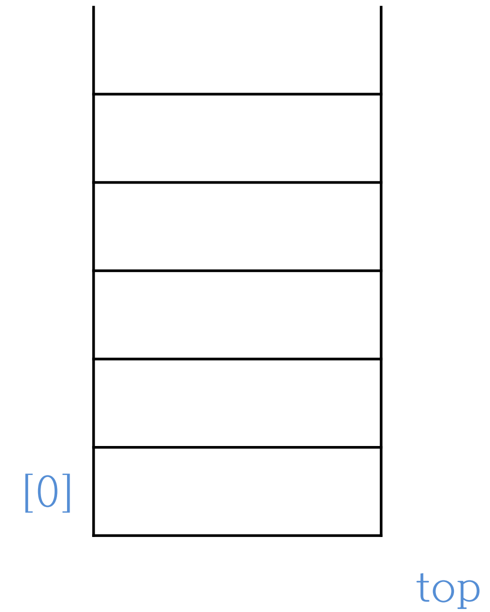
0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

mark

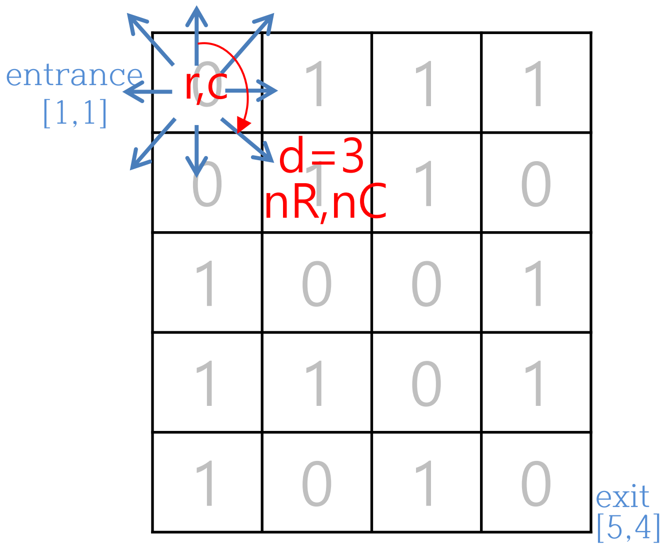
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack



Maze Search : Example

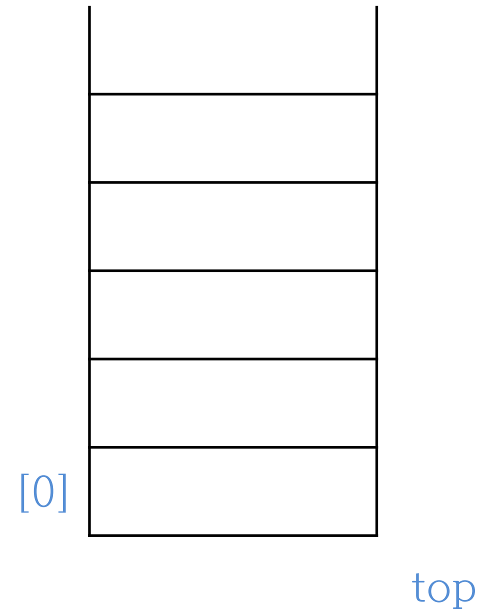
maze



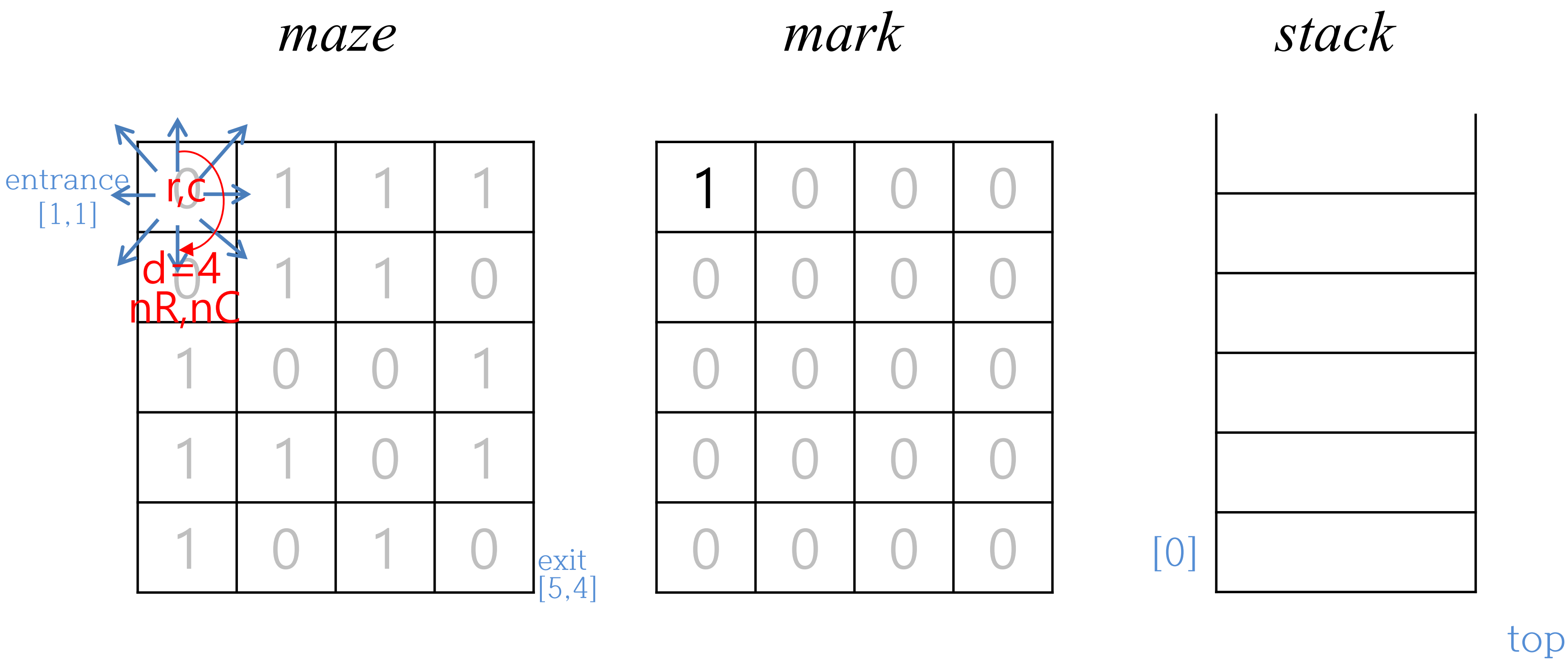
mark

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack



Maze Search : Example



Maze Search : Example

maze

entrance [1,1]

d=0

nR,nC

r,c

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

mark

1	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack

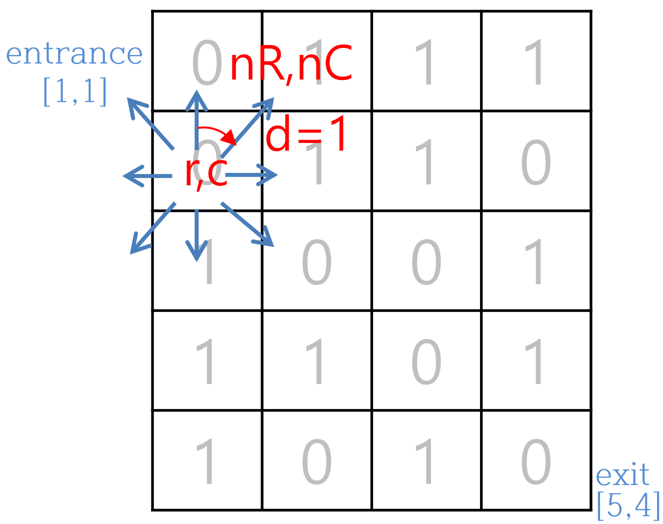
[0]	1, 1, 5	top

push(1, 1, 5)

(2, 1) 방문, (r, c, d) = (2, 1, 0)

Maze Search : Example

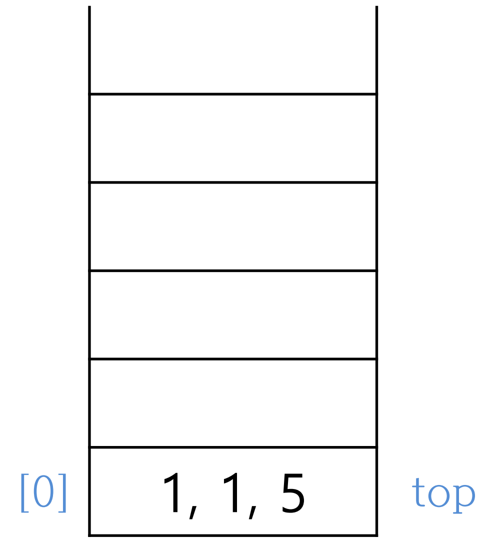
maze



mark

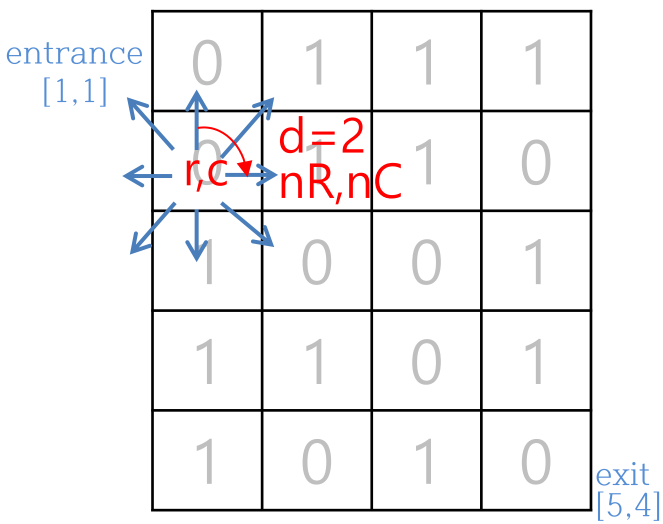
1	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

stack



Maze Search : Example

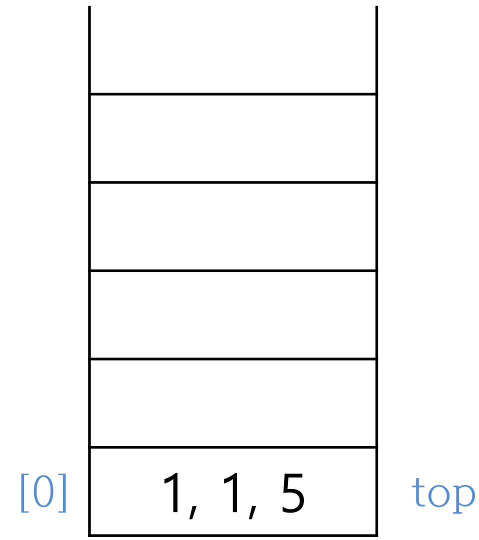
maze



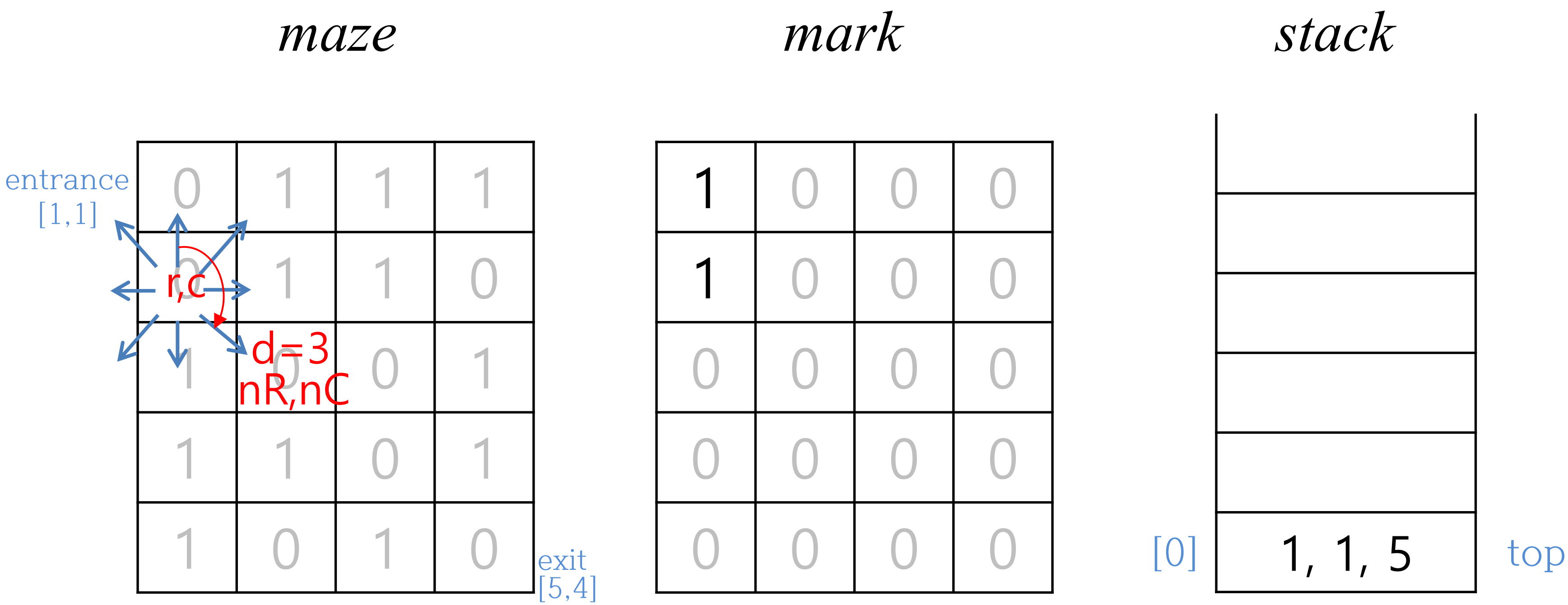
mark

1	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

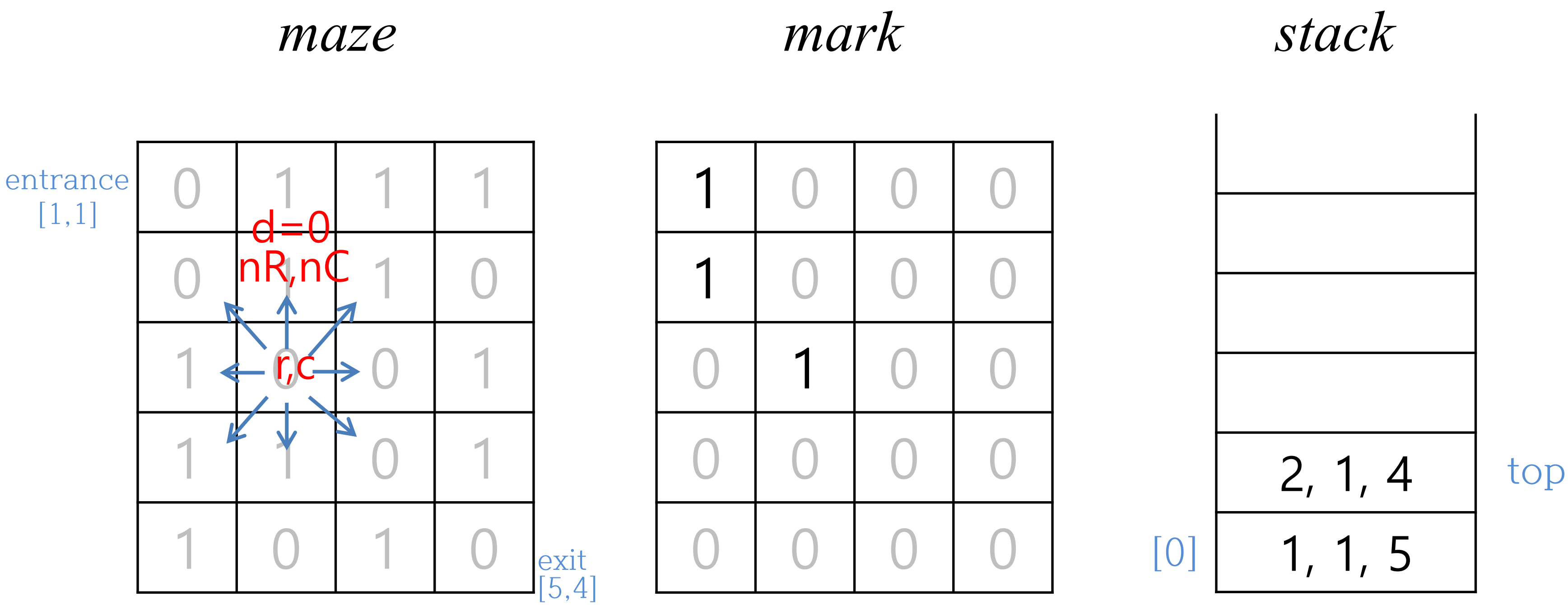
stack



Maze Search : Example

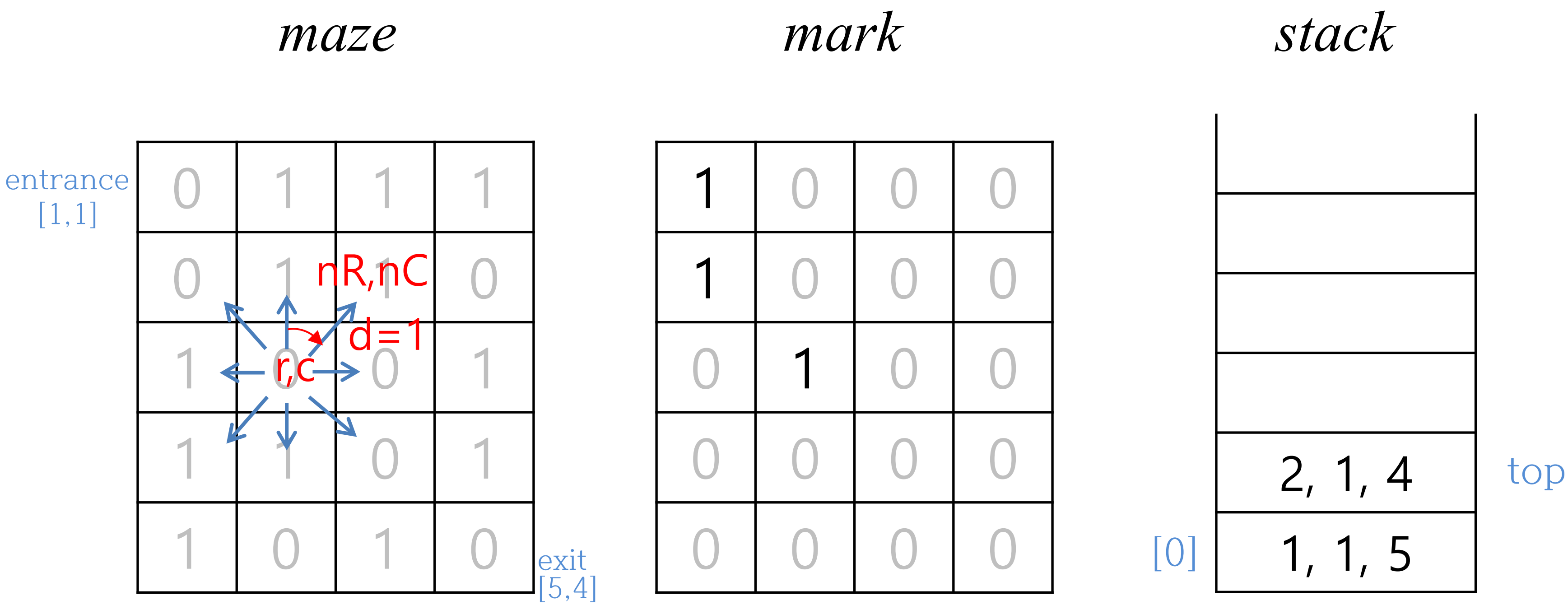


Maze Search : Example

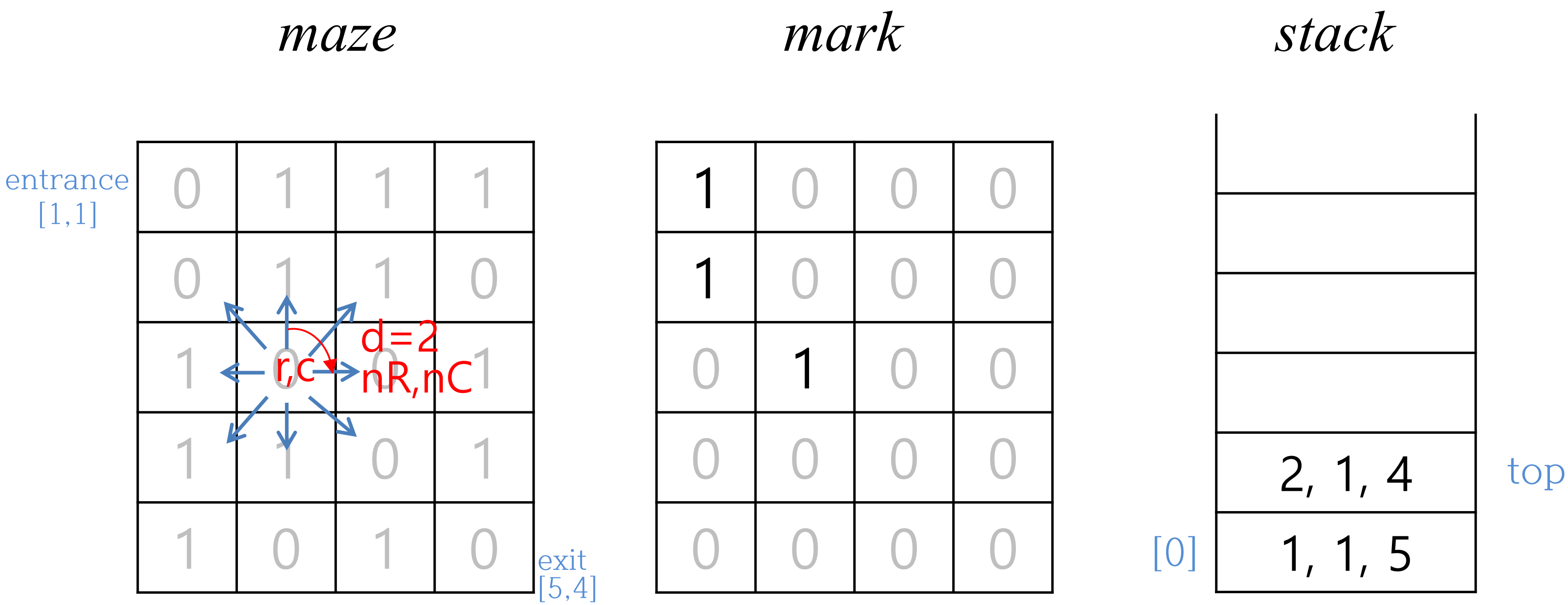


push(2, 1, 4)
(3, 2) 방문, (r, c, d) = (3, 2, 0)

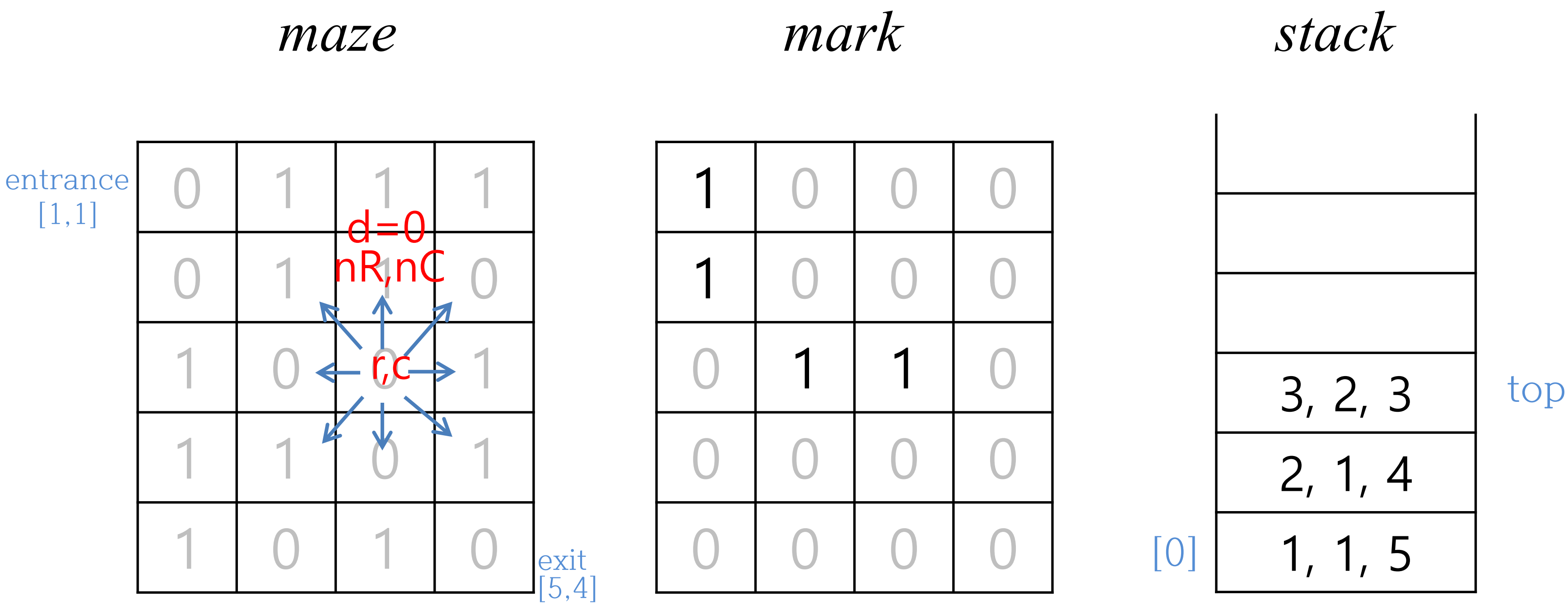
Maze Search : Example



Maze Search : Example



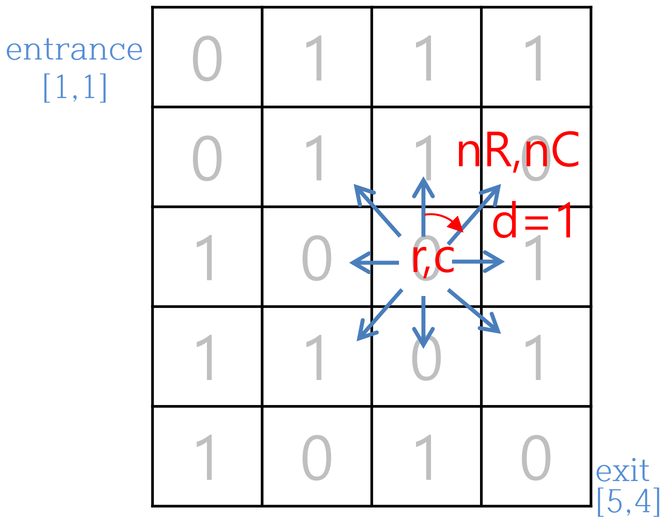
Maze Search : Example



push(3, 2, 3)
(3, 3) 방문, (r, c, d) = (3, 3, 0)

Maze Search : Example

maze



mark

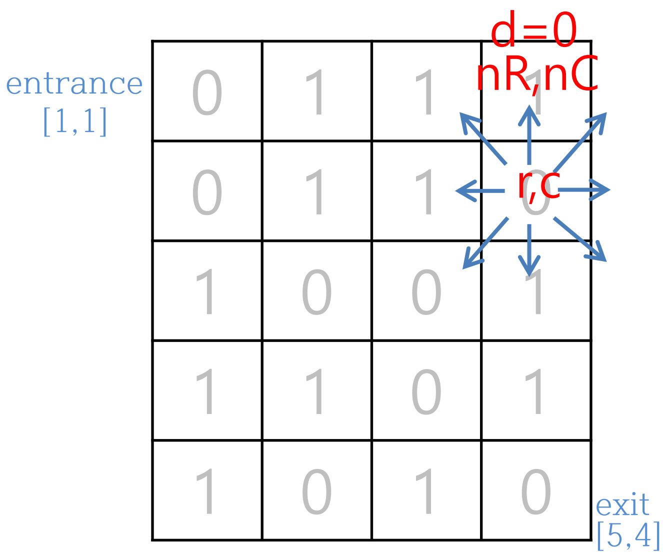
1	0	0	0
1	0	0	0
0	1	1	0
0	0	0	0
0	0	0	0

stack

	3, 2, 3 (top)
	2, 1, 4
[0]	1, 1, 5

Maze Search : Example

maze



mark

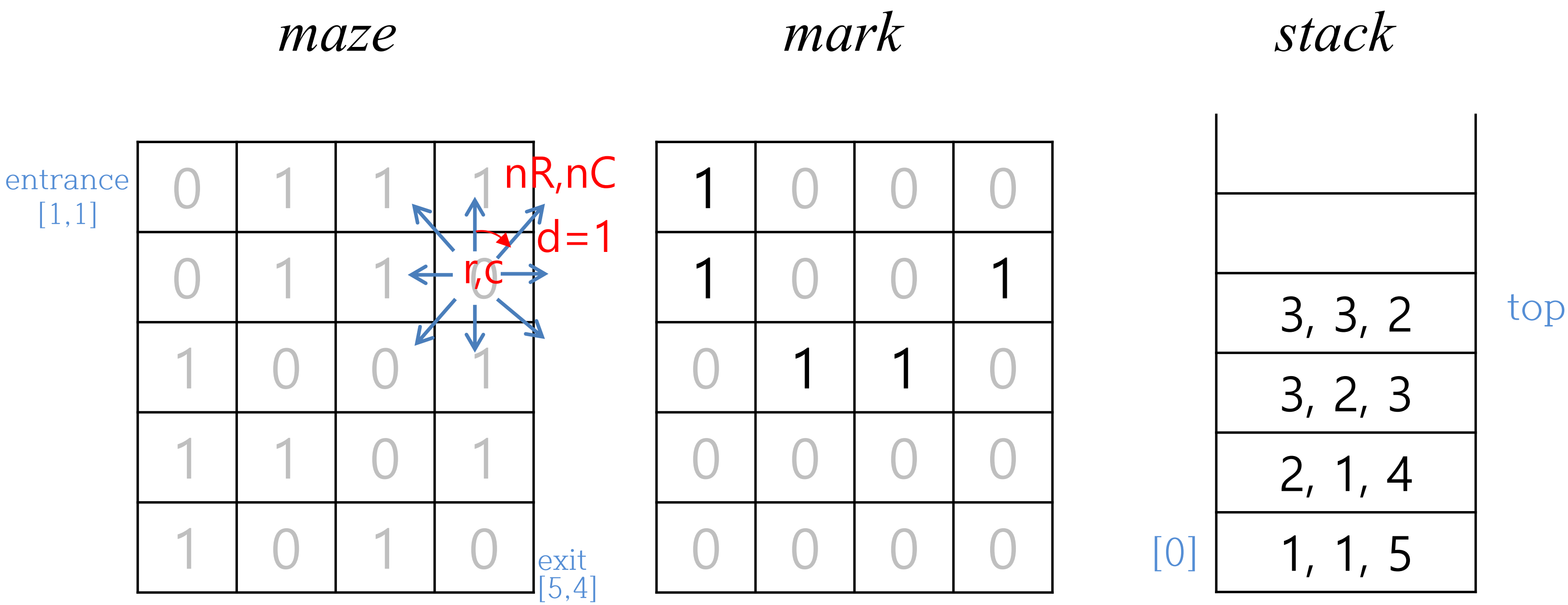
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
1, 1, 5	[0]

push(3, 3, 2)
(2, 4) 방문, (r, c, d) = (2, 4, 0)

Maze Search : Example



Maze Search : Example

maze

entrance [1,1]

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

r,c

d=2

nR,nC

mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

top	3, 3, 2
	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

Maze Search : Example

maze

entrance [1,1]

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

r,c

d=3

nR,nC

exit [5,4]

mark

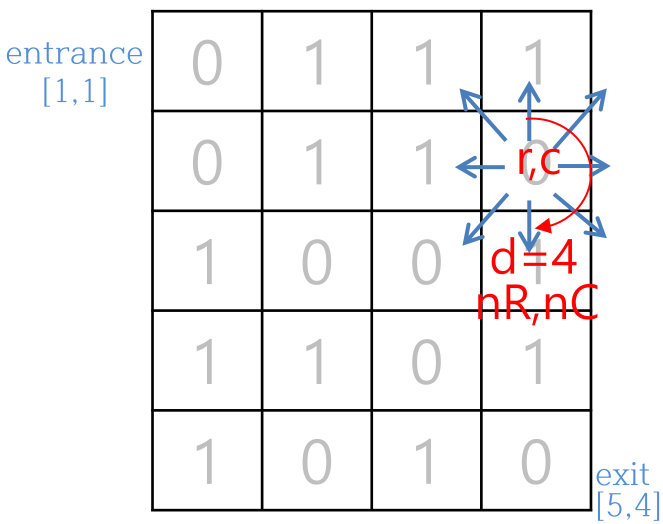
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

3, 3, 2	top
3, 2, 3	
2, 1, 4	
[0]	1, 1, 5

Maze Search : Example

maze



mark

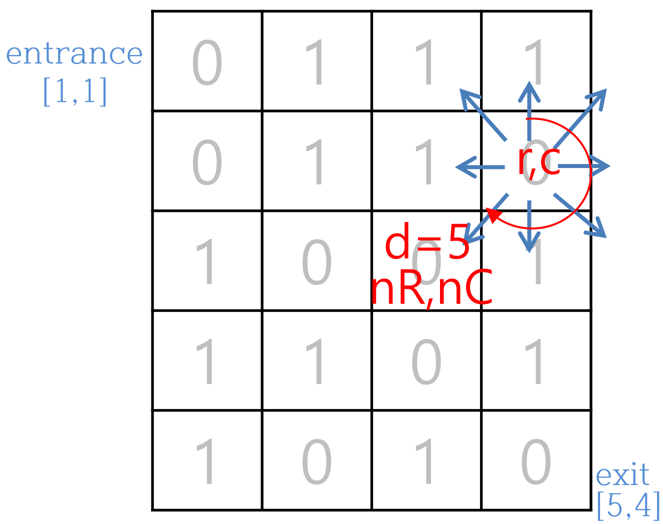
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

	3, 3, 2	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze



mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

top	3, 3, 2
	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

Maze Search : Example

maze

entrance [1,1]

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

d=6

nR,nC

r,c

mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

	3, 3, 2	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze

entrance [1,1]

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

exit [5,4]

d=7

nR,nC

r,c

mark

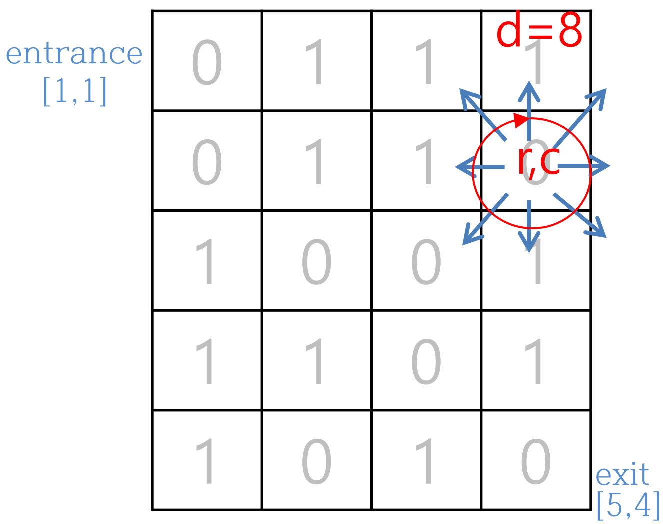
1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

stack

	3, 3, 2	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze



mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	0	0
0	0	0	0

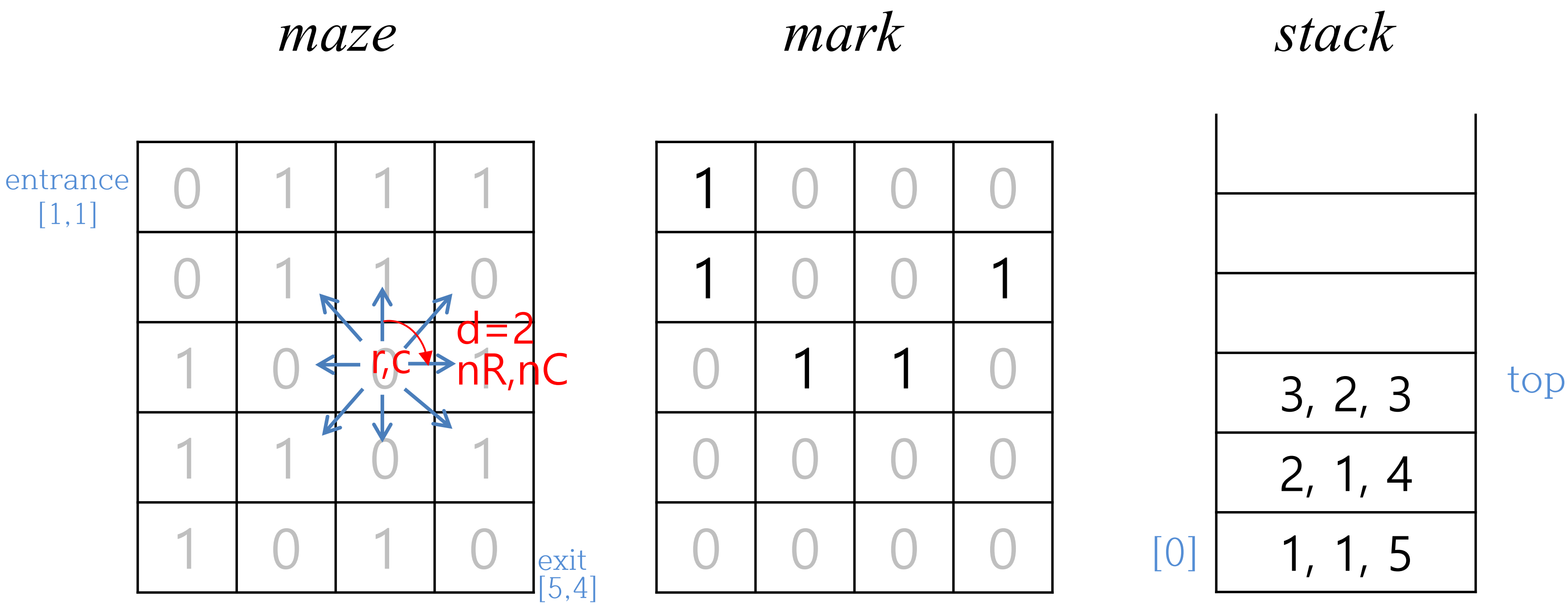
stack

top	3, 3, 2
	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

(d < 8 && !found) ? No!

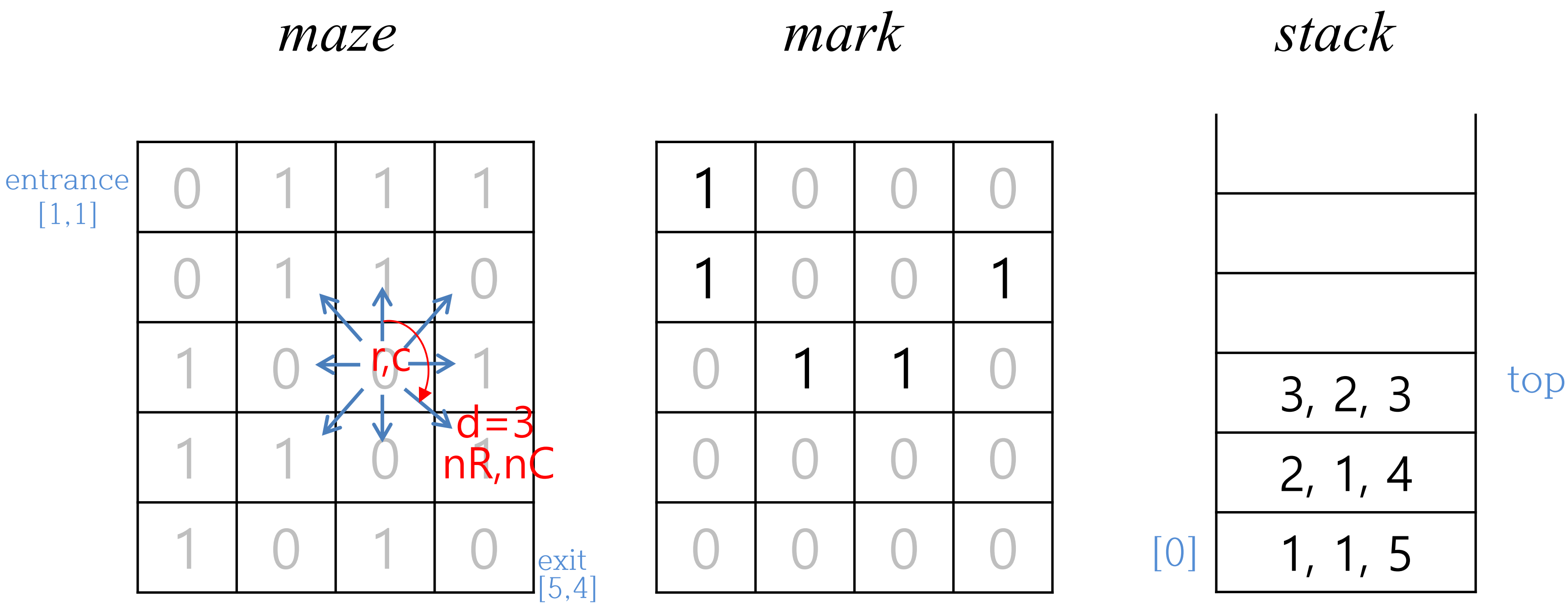
(top > -1 && !found) ? Yes!

Maze Search : Example

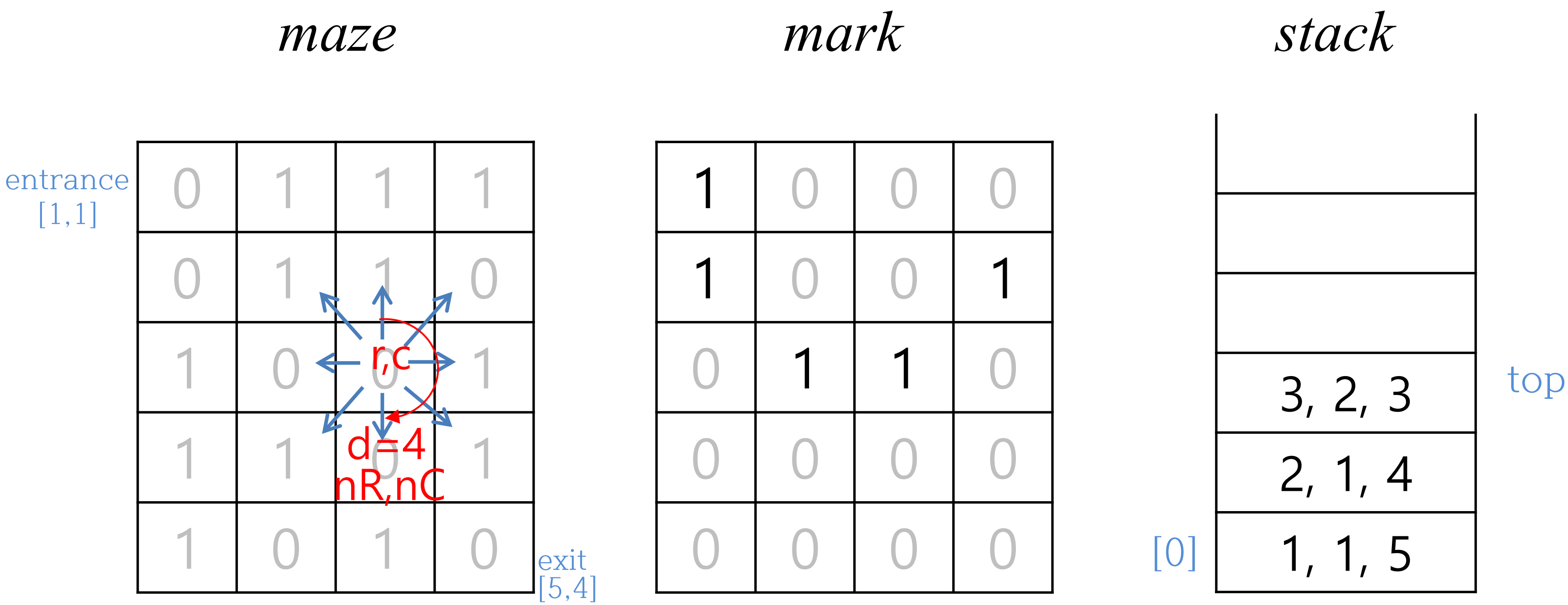


pop()
(r, c, d) = (3, 3, 2)

Maze Search : Example

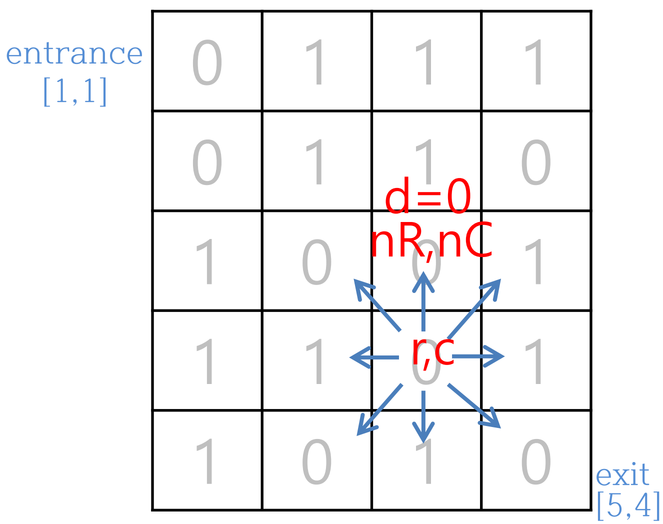


Maze Search : Example



Maze Search : Example

maze



mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

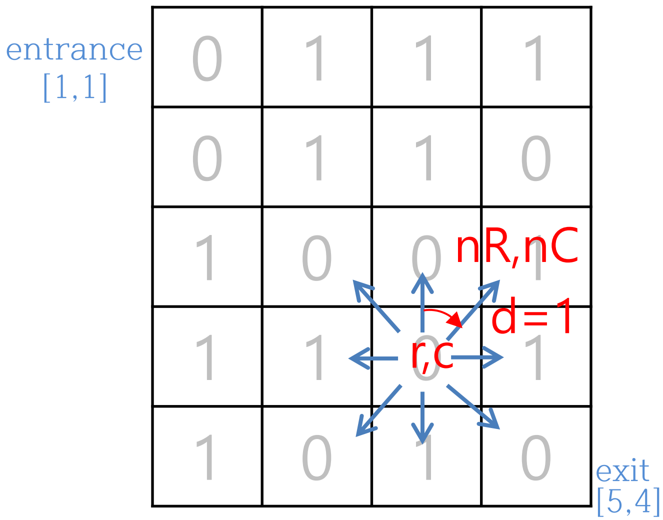
stack

	3, 3, 5	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

push(3, 3, 5)
(4, 3) 방문, (r, c, d) = (4, 3, 0)

Maze Search : Example

maze



mark

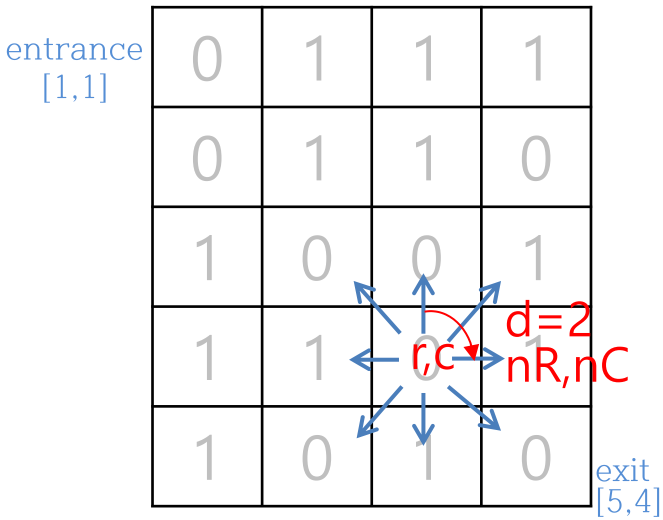
1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

stack

	3, 3, 5	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze



mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

stack

	3, 3, 5	top
	3, 2, 3	
	2, 1, 4	
[0]	1, 1, 5	

Maze Search : Example

maze

entrance [1,1]

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0 (r,c)	1
1	0	1	0 (nR,nC)

d=3

exit [5,4]

mark

1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

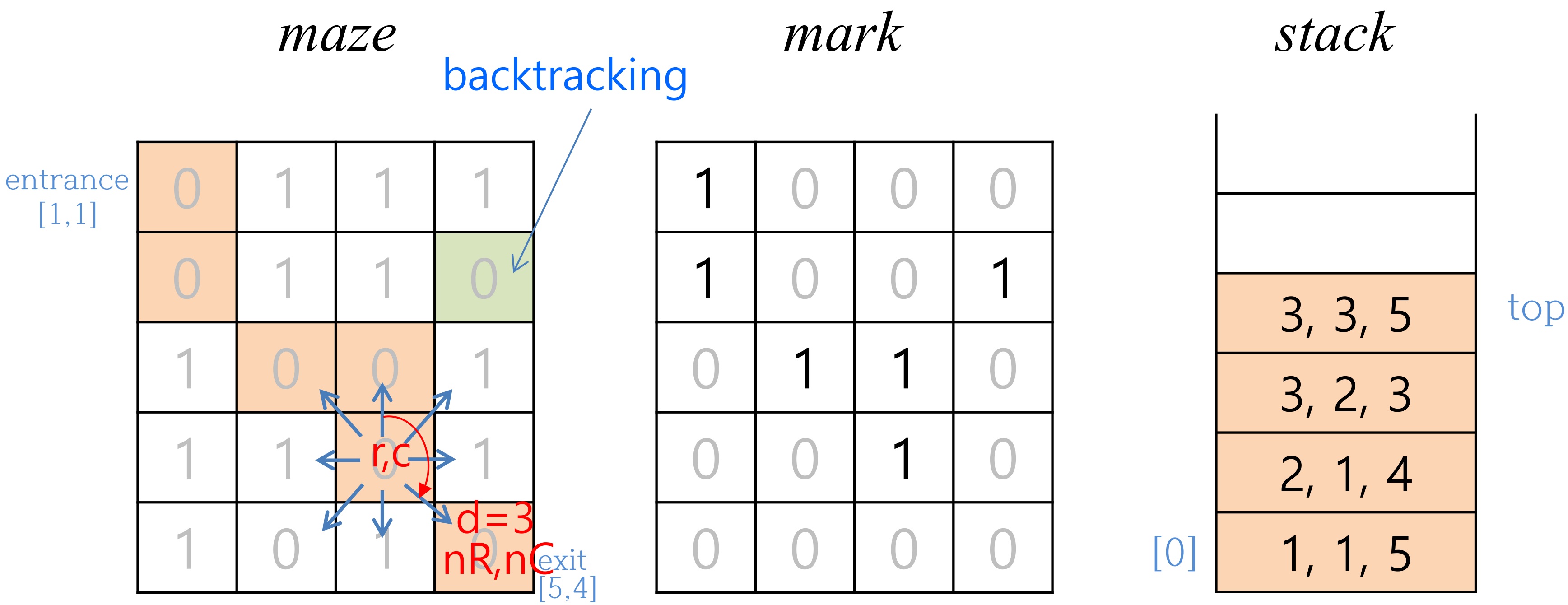
stack

top	3, 3, 5
	3, 2, 3
	2, 1, 4
[0]	1, 1, 5

다음 위치 (nR, nC)가 출구(EXIT_ROW, EXIT_COL)임

경로발견!

Maze Search : Example



< 경로출력순서>

① stack[0] → stack[top]

② 현재 위치 (r, c)

③ 출구 위치 (EXIT_ROW, EXIT_COL)

path: (1, 1), (2, 1), (3, 2), (3, 3), (4, 3), (5, 4)

- Stack
  - Use the implementation of section 3.1 or 3.2

```
typedef struct {
        short int row;
        short int col;
        short int dir;
        } element;
```

  - Capacity
    - Each position in the maze is visited no more than once.
    - An $m \times p$ maze has at most mp zeroes.
    - ***mp*** is sufficient for the stack capacity.

```
initialize a stack to the maze's entrance coordinates and
direction to north;
while (stack is not empty) {
   /* move to position at top of stack */
   <row,col,dir> = delete from top of stack;
   while (there are more moves from current position) {
      <nextRow, nextCol> = coordinates of next move;
      dir = direction of move;
      if ((nextRow == EXIT_ROW) && (nextCol == EXIT_COL))
         success;
      if (maze[nextRow][nextCol] == 0 &&
                    mark[nextRow][nextCol] == 0) {
      /* legal move and haven't been there */
         mark[nextRow][nextCol] = 1;
         /* save current position and direction */
         add <row,col,dir> to the top of the stack;
         row = nextRow;
         col = nextCol;
         dir = north;
      }
   }
}
printf("No path found\n");
```

Program 3.11: Initial maze algorithm

```c
void path(void)
{/* output a path through the maze if such a path exists */
  int i, row, col, nextRow, nextCol, dir, found = FALSE;
  element position;
  mark[1][1] = 1; top = 0;
  stack[0].row = 1;  stack[0].col = 1;  stack[0].dir = 1;
  while (top > -1 && !found) {
    position = pop();
    row = position.row;  col = position.col;
    dir = position.dir;
    while (dir <  8 && !found) {
      /* move in direction dir */
      nextRow = row + move[dir].vert;
      nextCol = col + move[dir].horiz;
      if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
        found = TRUE;
      else if ( !maze[nextRow][nextCol] &&
      ! mark[nextRow][nextCol]) {
        mark[nextRow][nextCol] = 1;
        position.row = row; position.col = col;
        position.dir = ++dir;
        push(position);
        row = nextRow; col = nextCol; dir = 0;
      }
      else ++dir;
    }
  }
  if (found) {
    printf("The path is:\n");
    printf("row  col\n");
    for (i = 0; i <= top; i++)
      printf("%2d%5d",stack[i].row, stack[i].col);
    printf("%2d%5d\n",row,col);
    printf("%2d%5d\n",EXIT_ROW,EXIT_COL);
  }
  else printf("The maze does not have a path\n");
}
```

Program 3.12: Maze search function

Analysis of *path*:

- each position within the maze is visited no more than once,
- worst case complexity : $O(mp)$, for $m \times p$ maze

3.6 Evaluation of Expressions

3.6.1 Expressions

- Complex expressions
  - ((*rear*+1==*front*)||((*rear*==*MAX_QUEUE_SIZE*-1)&& !*front*))
  - operators, operands, parentheses

- Complex assignment statements
  - x = a / b – c + d * e – a * c

- The order in which the operations are performed?

  - If a = 4, b = c = 2, d = e = 3,
    - x = ((a/b)-c)+(d*e)-(a*c) = ((4/2)-2)+(3*3)-(4*2) = 1
    - x = (a/(b-c+d))*(e-a)*c = (4/(2-2+3))*(3-4)*2 = -2.66666…

<table>
<tr><th>Token</th><th>Operator</th><th>Precedence[1]</th><th>Associativity</th></tr>
<tr><td>()
[]
→ .</td><td>function call
array element
struct or union member</td><td>17</td><td>left-to-right</td></tr>
<tr><td>-- ++</td><td>decrement, increment[2]</td><td>16</td><td>left-to-right</td></tr>
<tr><td>-- ++
!
~
- +
& *
sizeof</td><td>decrement, increment[3]
logical not
one's complement
unary minus or plus
address or indirection
size (in bytes)</td><td>15</td><td>right-to-left</td></tr>
<tr><td>(type)</td><td>type cast</td><td>14</td><td>right-to-left</td></tr>
<tr><td>* / %</td><td>multiplicative</td><td>13</td><td>left-to-right</td></tr>
<tr><td>+ -</td><td>binary add or subtract</td><td>12</td><td>left-to-right</td></tr>
<tr><td><< >></td><td>shift</td><td>11</td><td>left-to-right</td></tr>
<tr><td>> >=
< <=</td><td>relational</td><td>10</td><td>left-to-right</td></tr>
<tr><td>== !=</td><td>equality</td><td>9</td><td>left-to-right</td></tr>
<tr><td>&</td><td>bitwise and</td><td>8</td><td>left-to-right</td></tr>
<tr><td>^</td><td>bitwise exclusive or</td><td>7</td><td>left-to-right</td></tr>
<tr><td>|</td><td>bitwise or</td><td>6</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>logical and</td><td>5</td><td>left-to-right</td></tr>
<tr><td>||</td><td>logical or</td><td>4</td><td>left-to-right</td></tr>
<tr><td>?:</td><td>conditional</td><td>3</td><td>right-to-left</td></tr>
<tr><td>= += -= /= *= %=
<<= >>= &= ^= |=</td><td>assignment</td><td>2</td><td>right-to-left</td></tr>
<tr><td>,</td><td>comma</td><td>1</td><td>left-to-right</td></tr>
</table>

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

Figure 3.12: Precedence hierarchy for C

Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first.

3.6.2 Evaluating Postfix Expressions

- **Infix notation**
  - binary operator is in-between its two operands
- **Prefix notation**
  - operator appears before its operands
- **Postfix notation**
  - Each operator appears after its operands
  - Used by compiler
  - *Parentheses-free notation*
  - To evaluate expression, we make a single left-to-right scan of it (no precedence hierarchy)
  - Use *stack*

Infix	Postfix
2+3*4	2 3 4*+
$a*b+5$	$ab*5+$
(1+2)*7	1 2+7*
$a*b/c$	$ab*c/$
$((a/(b-c+d))*(e-a)*c)$	$abc-d+/ea-*c*$
$a/b-c+d*e-a*c$	$ab/c-de*+ac*-$

Figure 3.13: Infix and postfix notation

Token	Stack [0]	Stack [1]	Stack [2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
−	6/2−3			0
4	6/2−3	4		1
2	6/2−3	4	2	2
*	6/2−3	4*2		1
+	6/2−3+4*2			0

Figure 3.14: Postfix evaluation

postfix expression
6 2/3-4 2*+

- Representation of stack and expression
  - Assumption that expression contains only
    - Binary operators : +, -, *, /, %
    - Operands : single digit integers

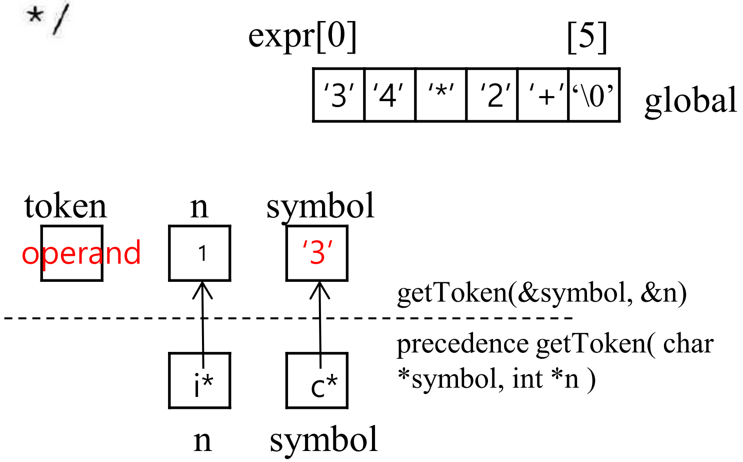
```
#define MAX_STACK_SIZE 100  /* maximum stack size */
#define MAX_EXPR_SIZE 100    /* max size of expression */

typedef enum { lparen, rparen, plus, minus, times, divide,
                              mod, eos, operand } precedence;

int stack[MAX_STACK_SIZE];   /* global stack */
char expr[MAX_EXPR_SIZE];    /* global input string
                                 (a postfix expression)*/
```

```
int eval(void)
{/* evaluate a postfix expression, expr, maintained as a
   global variable. '\0' is the the end of the expression.
    The stack and top of the stack are global variables.
    getToken is used to return the token type and
    the character symbol. Operands are assumed to be single
    character digits */
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0; /* counter for the expression string */
  top = -1;
  token = getToken(&symbol, &n);
  while (token != eos) {
    if (token == operand)
      push(symbol-'0'); /* stack insert */
    else {
      /* pop two operands, perform operation, and
         push result to the stack */
      op2 = pop(); /* stack delete */
      op1 = pop();
      switch(token) {
        case plus: push(op1+op2);
                   break;
        case minus: push(op1-op2);
                    break;
        case times: push(op1*op2);
                    break;
        case divide: push(op1/op2);
                     break;
        case mod: push(op1%op2);
      }
    }
    token = getToken(&symbol, &n);
  }
  return pop(); /* return result */
}
```

Program 3.13: Function to evaluate a postfix expression

※ symbol-'0' makes a single digit integer ('0': ASCII value of 48).

```
precedence getToken(char *symbol, int *n)
{/* get the next token, symbol is the character
    representation, which is returned, the token is
    represented by its enumerated value, which
    is returned in the function name */
  *symbol = expr[(*n)++];
  switch (*symbol) {
    case '(' : return lparen;
    case ')' : return rparen;
    case '+' : return plus;
    case '-' : return minus;
    case '/' : return divide;
    case '*' : return times;
    case '%' : return mod;
    case '\0' : return eos;
    default  : return operand; /* no error checking,
                                   default is operand */
  }
}
```



Program 3.14: Function to get a token from the input string

3.6.3 Infix to Postfix

- An algorithm by hand

(1) Fully parenthesize the expression.

(2) Move all binary operators so that they replace their corresponding right parentheses.

(3) Delete all parentheses.

Infix : a/b-c+d*e-a*c

(1) ((((a/b)-c) + (d*e)) - (a*c))

(2) ((((a b / c- (de* + (ac*-

(3) a b / c- de* + ac*-

※ It is inefficient on a computer
because it requires 2 passes

- **Example:** Simple expression
  - Input : a+b*c
  - Output : abc*+

Token generation by ***scanning*** left to right.
- ***Operands*** are passed to the output expression.
- ***Operators*** are stacked and unstacked *by their precedence*.

Token	Stack [0]	Stack [1]	Stack [2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+			1	*ab*
c	+	*		1	*abc*
eos				−1	*abc*+*

icp[token]

isp[stack[top]]

* push

Figure 3.15: Translation of $a+b*c$ to postfix

- **Example:** Parenthesized expression
  - Input : a*(b+c)*d
  - Output : abc+*d*

Token		Stack [0]	Stack [1]	Stack [2]	Top	Output
a					−1	a
$*$		$*$			0	a
(	>	$*$	(		1	a
b		$*$	(		1	ab
$+$	>	$*$	(	$+$	2	ab
c		$*$	(	$+$	2	abc
)		$*$ pop→print			0	$abc+$
$*$	≤	$*$			0	$abc+*$
d		$*$			0	$abc+*d$
eos					-1	$abc+*d*$

Figure 3.16: Translation of $a*(b+c)*d$ to postfix

- ***Left parenthesis***
  - behaves like **a low-precedence operator** <u>**when it is on the stack**</u>, and **a high-precedence one** <u>**when it is not**</u>.

  - is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found

- ***Right parenthesis*** is never put on the stack.

- ***isp***(in-stack precedence) and ***icp***(incoming precedence)

```
precedence stack[MAX_STACK_SIZE];

/* isp and icp arrays – index is the value of precedence
lparen, rparen, plus, minus, times, divide, mod, or eos */
static int isp[] =  {  0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] =  { 20, 19, 12, 12, 13, 13, 13, 0 };
```

Q. *isp* [*plus*] ?

```c
void postfix(void)
{/* output the postfix of the expression. The expression
    string, the stack, and top are global */
  char symbol;
  precedence token;
  int n = 0;
  top = 0;    /* place eos on stack */
  stack[0] = eos;
  for (token = getToken(&symbol, &n); token != eos;
                           token = getToken(&symbol,&n)) {
    if (token == operand)
      printf("%c", symbol);
    else if (token == rparen) {
      /* unstack tokens until left parenthesis */
      while (stack[top] != lparen)
        printToken(pop());
      pop();  /* discard the left parenthesis */
    }
    else {// operator, lparen
      /* remove and print symbols whose isp is greater
         than or equal to the current token's icp */
      while(isp[stack[top]] >= icp[token])
        printToken(pop());
      push(token);
    }
  }
  while ( (token = pop()) != eos)
    printToken(token);
  printf("\n");
}
```

① 첫 번째 연산자 입력에 대해서도 (isp[stack[top]] >= icp[token]) 계산이 가능
② 마지막 while문의 조건문 검사도 쉽게 할 수 있음

Program 3.15: Function to convert from infix to postfix

- Analysis of ***postfix***
  - n : number of tokens in the expression
  - extracting tokens and outputting them : $\Theta(n)$
  - in two while loop, the number of tokens that get *stacked* and *unstacked* is linear in n : $\Theta(n)$
  - So, the total complexity : $\boldsymbol{\Theta(n)}$