S/W Testing Advanced level

Practical guide

Ivan Keliukh, Oleksandr Andrieiev

Goals

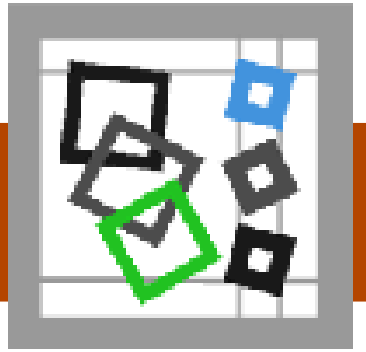
1. Make sure all applicants can use standard algorithms
2. Avoid typical mistakes
3. Speed up passing the test
4. Increase percentage of certifications

S/W certification testing

It is not a conventional contest

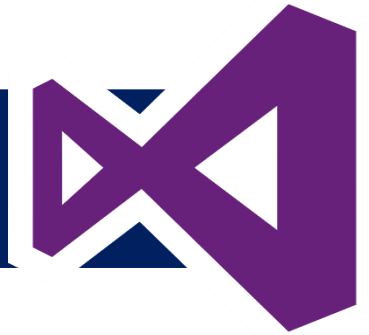
- Internet access is blocked
- Individual contest
- Usage of language features is restricted
- Poor environment
- Only Visual Studio for C and C++ is available
- Other courses target programming contests, not S/W testing

Preparation



Use ejudge

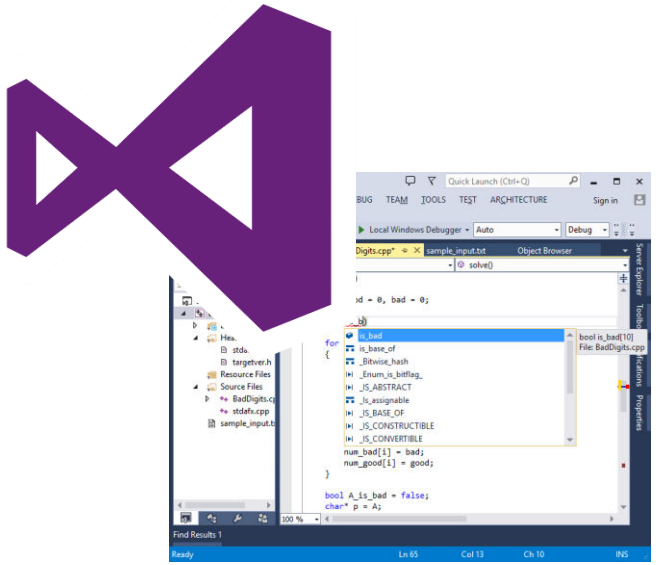
- Contains tasks from previous testing sessions
- Is accessible outside of SRK network
- Use timer to limit available time



Work in Visual Studio

- It is the only IDE available in SCS test system
- It is better to get used to shortcuts, window layout and code style

Visual Studio – hot keys

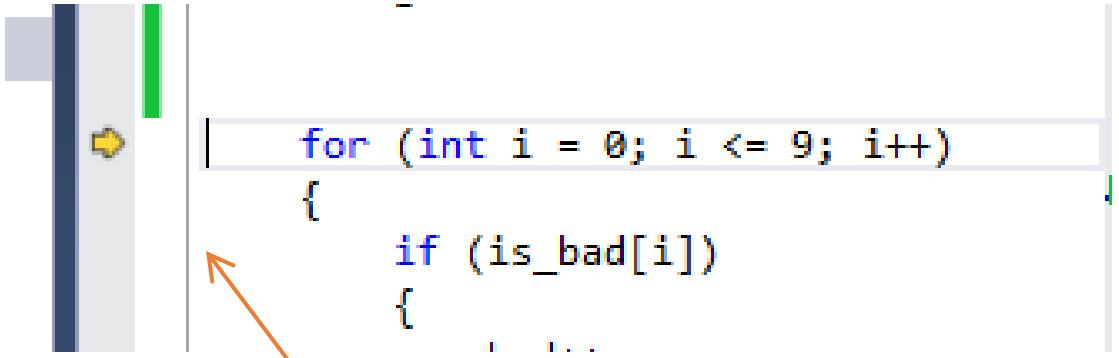


Editor

- **Ctrl+K,C** – comment, **Ctrl+K,U** – uncomment
- **Ctrl+Space** – autocomplete

Project/Debug

- **F7** – build
- **F5** – run application
- **F9** – toggle breakpoint
- **F10** – step over, **F11** – step into
- **Ctrl+F10** – run to…



Use **“Set next statement”** feature (**Ctrl+Shift+F10** or **drag the yellow arrow**)

At the test

- Read the statement
- Read the description of input and output
  - Now read the statement again, carefully
- Use **pen** and **paper** for self-check
  - Try to manually solve sample_input.txt and compare your answers to correct ones: this confirms your understanding of the statement
  - Allows you to write and debug algorithm without writing a single line of code
  - Fast and simple
- Prepare several test cases of your own and solve them manually
- Copy the initial code from SCS Test System to Visual Studio and run it. Chances are there are technical issues with VS and you need to change a seat.
- In some cases you can implement a solution based on brute-force search to check your understanding. It may even pass the test.

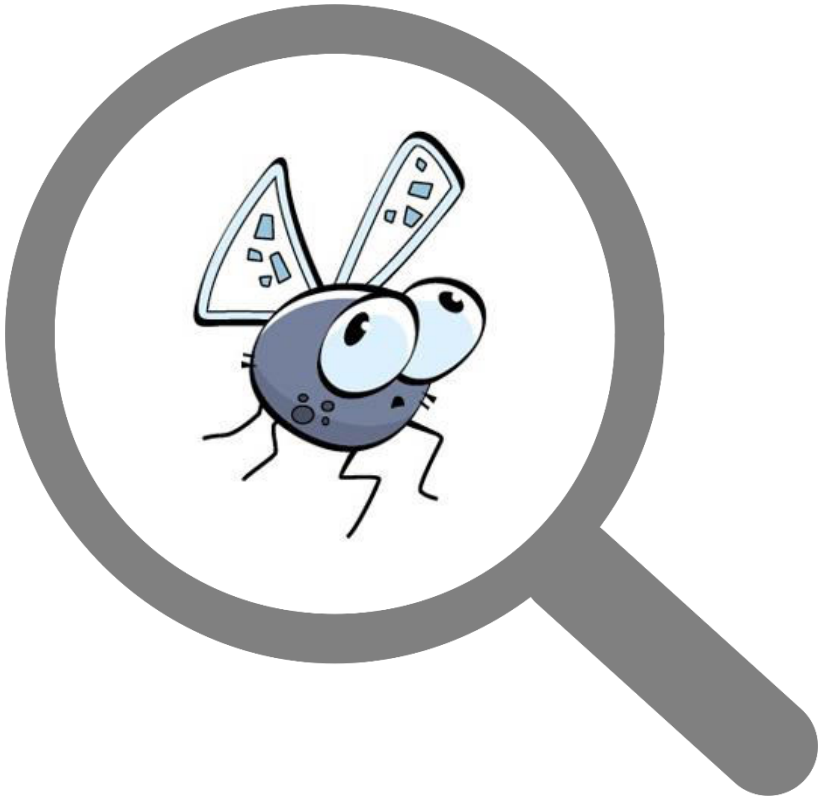
Production code

DON’T:

- Use copy-paste (best SW design pattern)
- Use short variable names everywhere

DO:

- Validate input
- Perform error handling



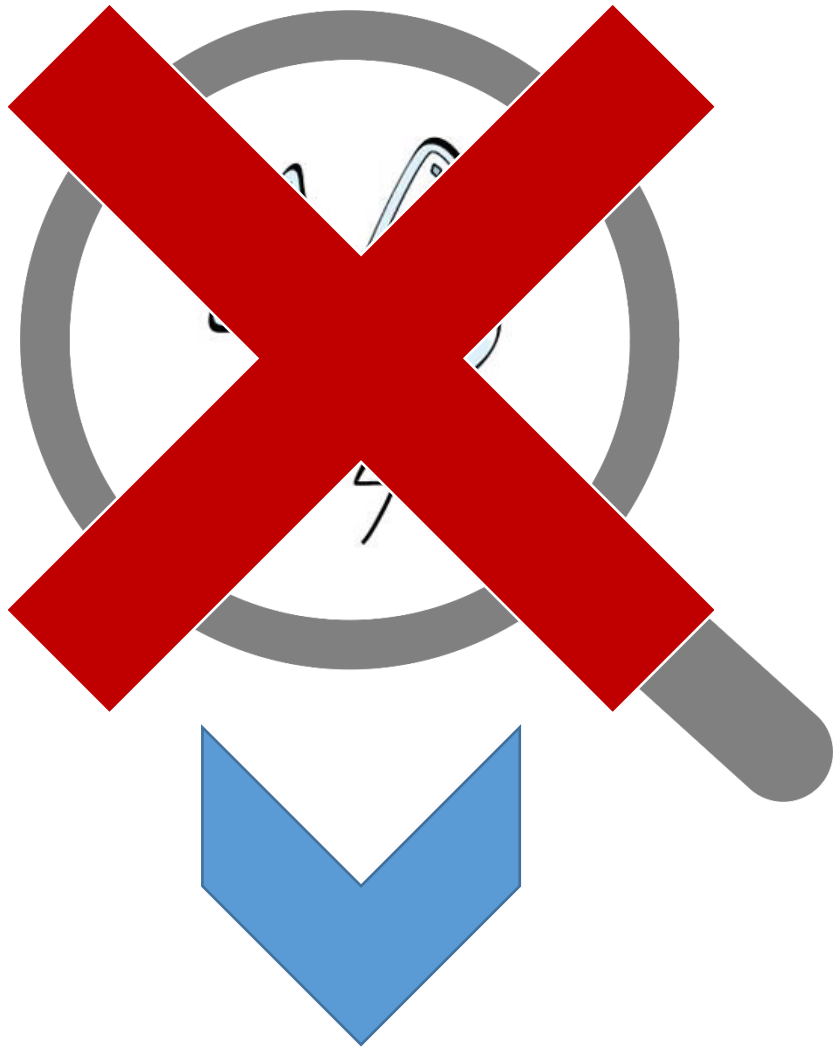
Code in S/W testing

DO:

- Use copy-paste (best SW design pattern)
- Use short variable names everywhere

DON'T:

- Validate input
- Perform error handling



Implementation speed

Code quality

For SW certification you don’t need:

- Scalability – no one will change the scope of the task
- Maintainability – no one will use this code after the test
- Readability – no one will read your code

What you need:

- Fast and simple code
- Correct answer

“Beautiful” code takes time, time that you don’t have – use “quick and dirty” prototyping

Input data – sample input

Task statement contains:

- Popup link with test cases: **sample_input.txt**
- Correct answers for test cases given

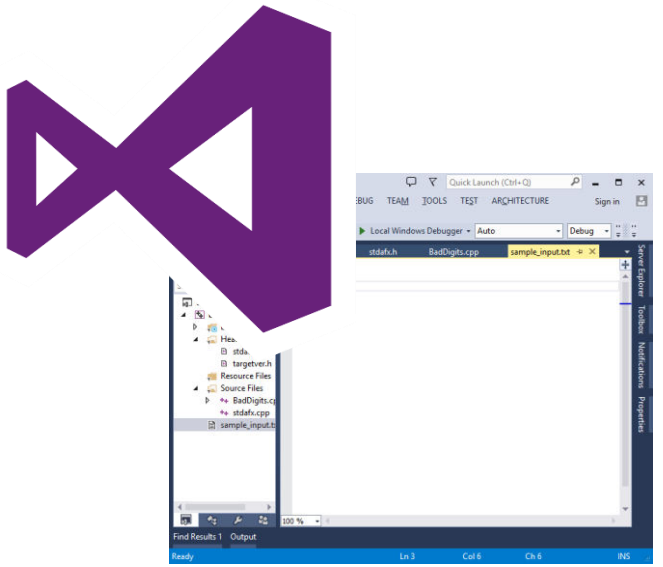
Redirecting input:

- Initialized solution code contains **//freopen** to redirect standard input to sample_input.txt
- Copy this code to Visual Studio and uncomment it
- **Remove before submit**

Using sample input:

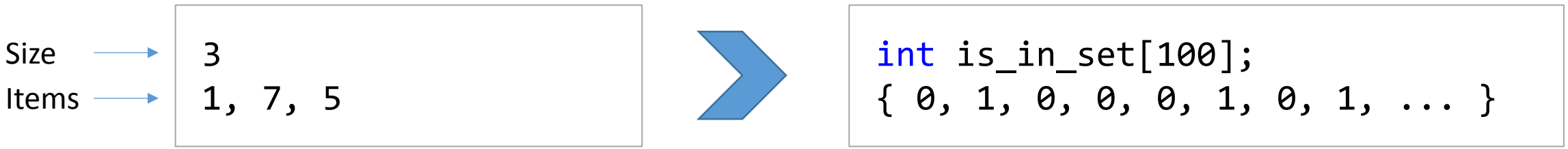
- Visual Studio project contains empty sample_input.txt
- Copy sample test cases there to verify solution (compare with results in statement)
- Add your own test cases for debug

sample_input.txt



Input data – processing

- Never spend precious time on implementing input data checks. All input data is always valid.
- Choose appropriate data type: make sure all possible values can be stored in your variable
- Sometimes it makes sense to immediately convert the data to convenient form
  - Graph should be converted from Edge List to Adjacency List (see later)
  - Set can be converted to array



Input data – processing

It is often the case that data as provided is not easy to work with. In this case it is often beneficial to transform it, even if transformed data requires more space/structures.

E.g. “endoscope” task:

Given data is an array of types of pipe: “┼”, “│”, etc. To check whether we can go from (i, j) to (i-1, j) we need to write a lot of code:

```
switch (a[i][j]) {
case “┼”:
if (a[i-1][j] == “┼” || a[i-1][j] == “│” || etc) {
    go_up;
}
```

However, we can use additional arrays, holding directions that current cell can be connected with (this array can be easily filled during the data input). Condition becomes something like:

```
if (goesTo[i][j][up] == 1 && goesTo[i-1][j][down] == 1)
    go_up;
}
```

Language

Java

- You may experience issues with performance of parsing input data, processing strings and with data types
- Do not use Java in SW Testing unless you have experience with using it in Competitive Programming (what are you doing on this session then?)

STL

- Using of STL in SW Testing is prohibited
- So far none of the tasks for Advanced level required using of STL

Language – dynamic memory and pointers

Never use dynamic memory – it is not needed for Advanced level test

- Instead of using dynamic memory, declare arrays for the worst case scenario
- You can infer the limit for input data from the task statement and for internal data from simple calculation (see later)
- Memory limit applies to dynamic memory as well

Do not use pointers and pointer arithmetic

- Use indices to access array elements for both traversal and reference

```
int* array = malloc(sizeof(int)*N);
```

```
int array[10000];
```

Language – stack

Limit usage of local variables – stack can be quickly exhausted. In most cases variables can be safely moved to global scope.

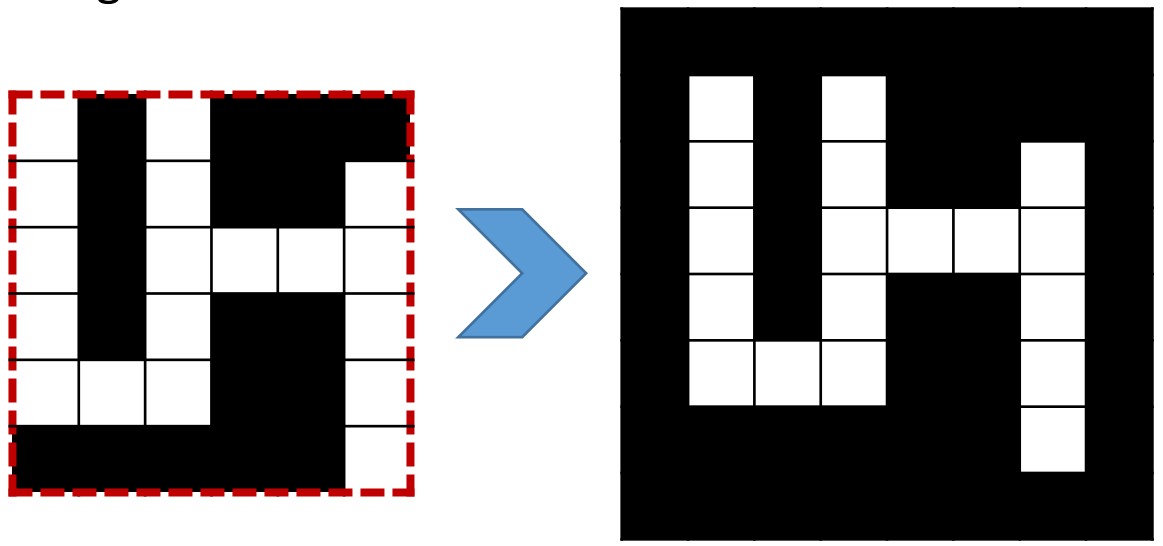
```
int solve()
{
    int A[1024][1024];
}
```

```
int A[1024][1024];

int solve()
{
}
```

Code style: 1-based indexing

- Only small amount of data is wasted by ignoring index 0
- Simplifies debugging a lot
- Can be used for numbering items, in identifiers and indexes
- Simplifies range checking by adding sentinel items
  - Specific example: unpassable walls around the grid

```
if (i >= 1) process(i - 1, j    );
if (i <  N) process(i + 1, j    );
if (j >= 1) process(i,     j - 1);
if (j <  N) process(i,     j + 1);
```



Code style - infinity

In most implementations it is possible to initialize the optimal result with big enough or small enough value.

```
int best_result = 300000000;
while (condition)
{
    int result;
    ... /* calculate result */

    if (best_result > result)
    {
        best_result = result;
    }
}
```

Code style – two-dimensional arrays

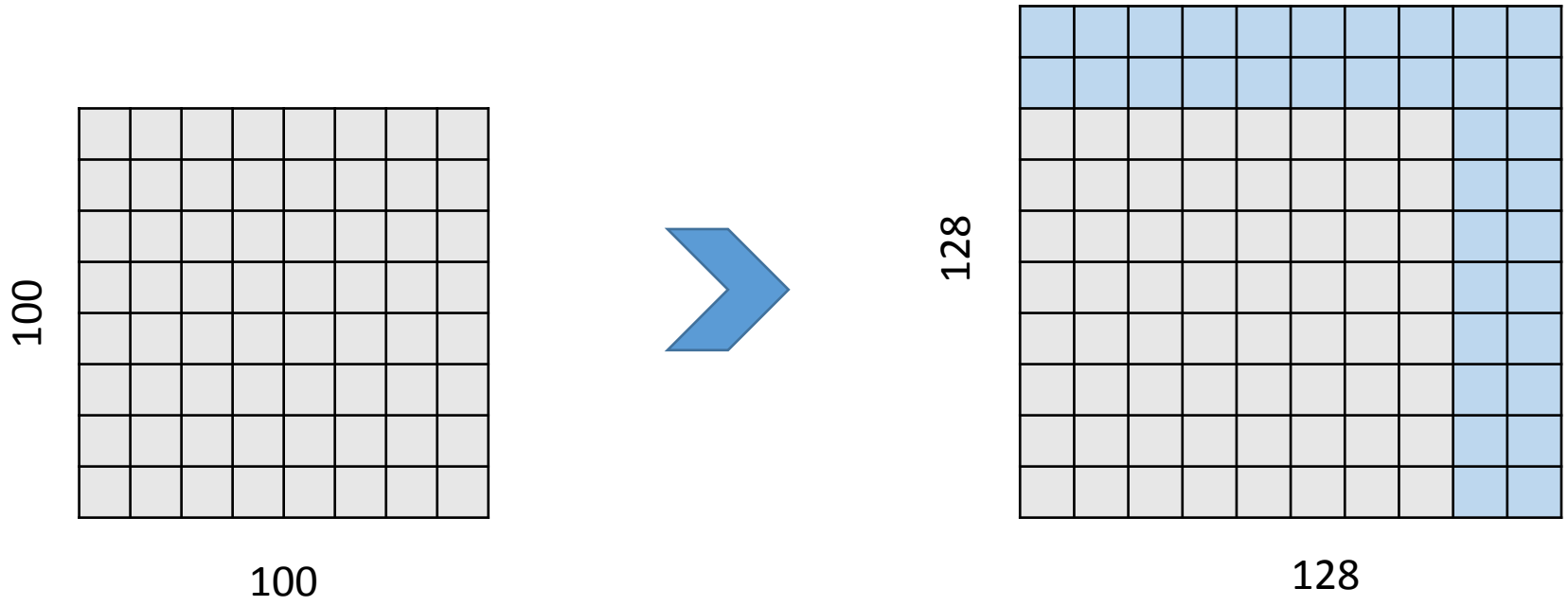
Declare and use two-dimensional arrays instead of emulating them with indexes or pointers

```
int matrix[10000]; // 100x100
...
int element = matrix[i*100 + j];
```

```
int matrix[100][100];
...
int element = matrix[i][j];
```

Code style – slack space

- Add slack space to arrays just for your convenience. For example, if the task statement requires 50x50 matrix, you can declare 64x64 array.



Code style – if statement

Always add brackets in if-statement

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
    goto fail;
    goto fail;
}
```

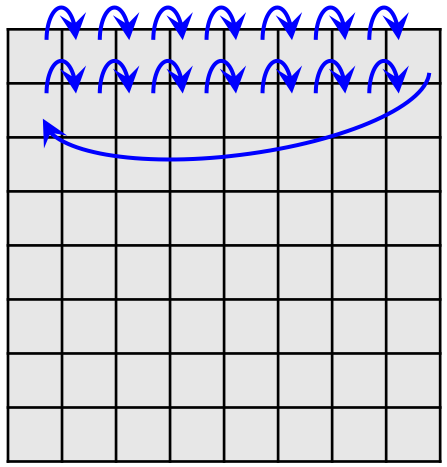
Code style – if statement

Break complex conditions

```
if (i == N-1 || i > 1 && A[i-1] != B[i-1] || i == 0 && A[i] == 17)
```

```
int is_last = i == N - 1;
int is_first_good = i == 0 && A[i] == 17;
if (is_last || is_first_good || i > 1 && A[i-1] != B[i-1])
```

Algorithms – brute-force search



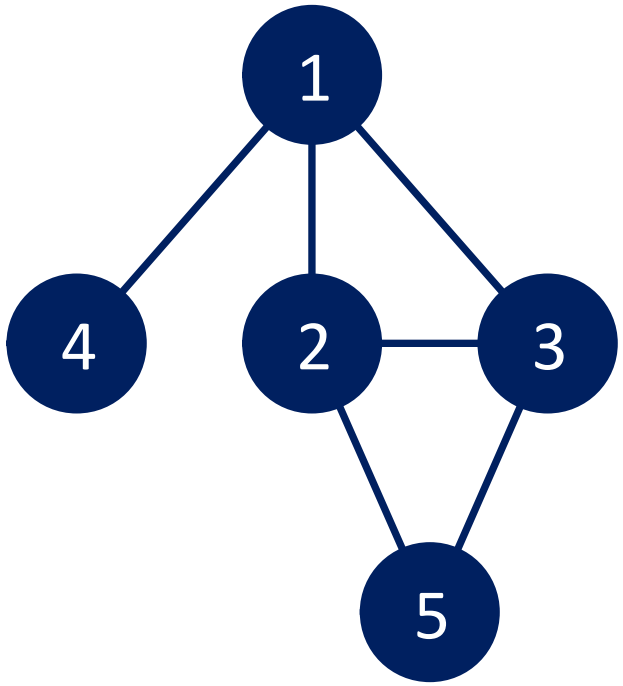
- Can be a valid solution if number of possible options to check is low (~10) – at least two examples in SW Testing Advanced level
- Make sure not to check the same value multiple times, consider saving and reusing results in an array
- Can be combined with other algorithms (e.g. start BFS from every cell in the matrix)
- Can be used to test and verify your optimal solution on small random sample sizes (only use if you know what you're doing)

Only use it if it is easy to implement

Algorithms – Adjacency List

Edge List

1-2
1-3
1-4
2-3
2-5
3-5



Adjacency List

[i]	Size	Adjacent		
1	3	2	3	4
2	3	1	3	5
3	3	1	2	5
4	1	1		
5	2	2	3	

Algorithms – Adjacency List Operations

① Declaration

```
int size[128];
int a[128][128];
```

② Initialization

```
for (int i = 0; i <= N; i++)
{
    size[i] = 0;
}
```

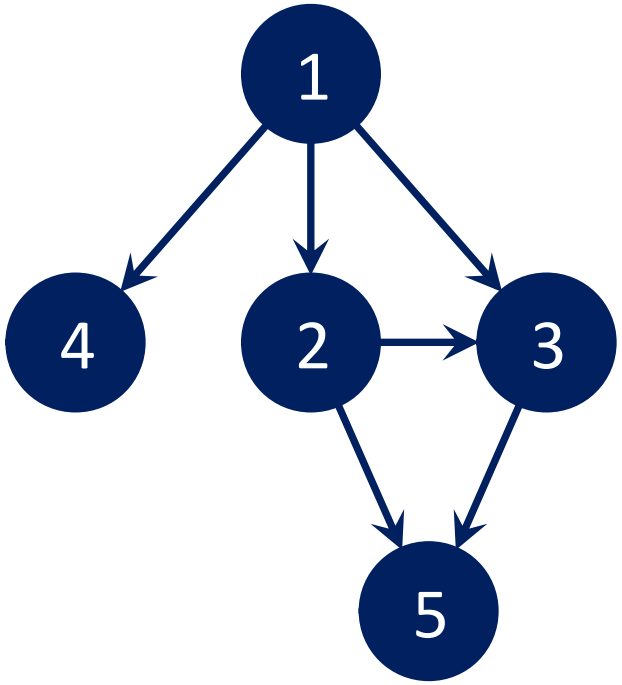
③ Adding edge (i-j)

```
a[i][size[i]] = j; size[i]++;
a[j][size[j]] = i; size[j]++;
```

④ Iterating adjacent vertices

```
for (int j = 0; j < size[i]; j++)
{
    int b = a[i][j];
    // process edge (i-b)
}
```

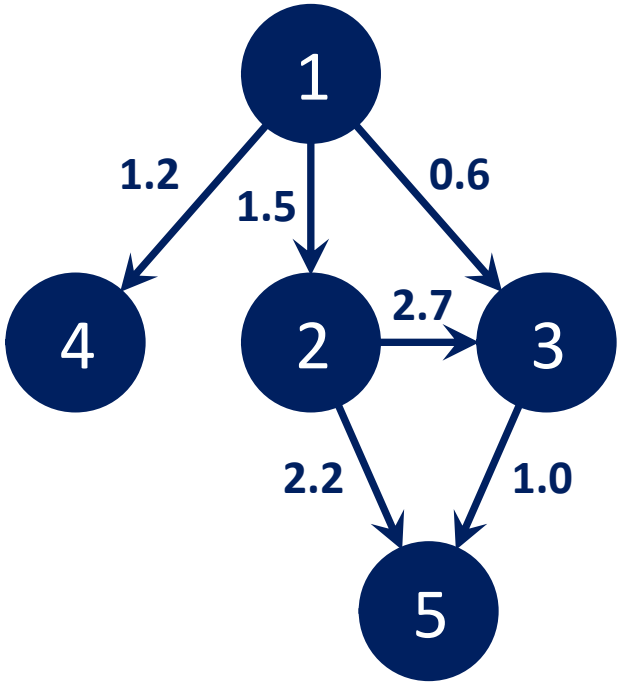
Algorithms – Adjacency List – directed graph



Adjacency List

[i]	Size	Adjacent		
1	3	2	3	4
2	2	3	5	
3	1	5		
4	0			
5	0			

Algorithms – Adjacency List – weighted graph

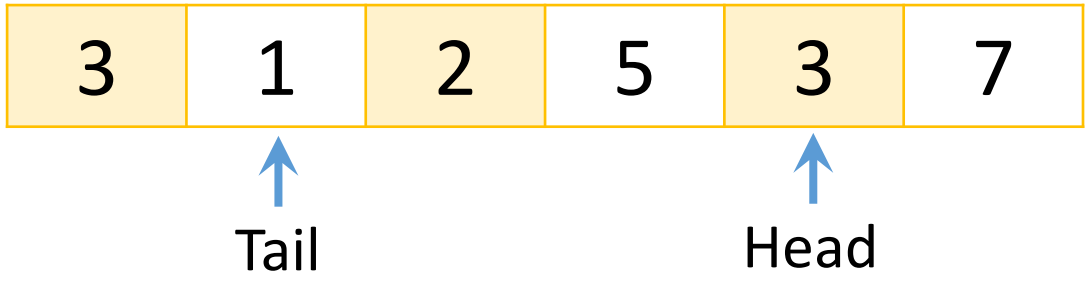


Adjacency List

[i]	Size	Adjacent			Weight		
1	3	2	3	4	1.5	0.6	1.2
2	2	3	5		2.7	2.2	
3	1	5			1.0		
4	0						
5	0						

Algorithms - Queue

Representation



① Declaration and initialization

```
int q[128];
int q_h = 0, q_t = 0;
```

② Add element (push_back)

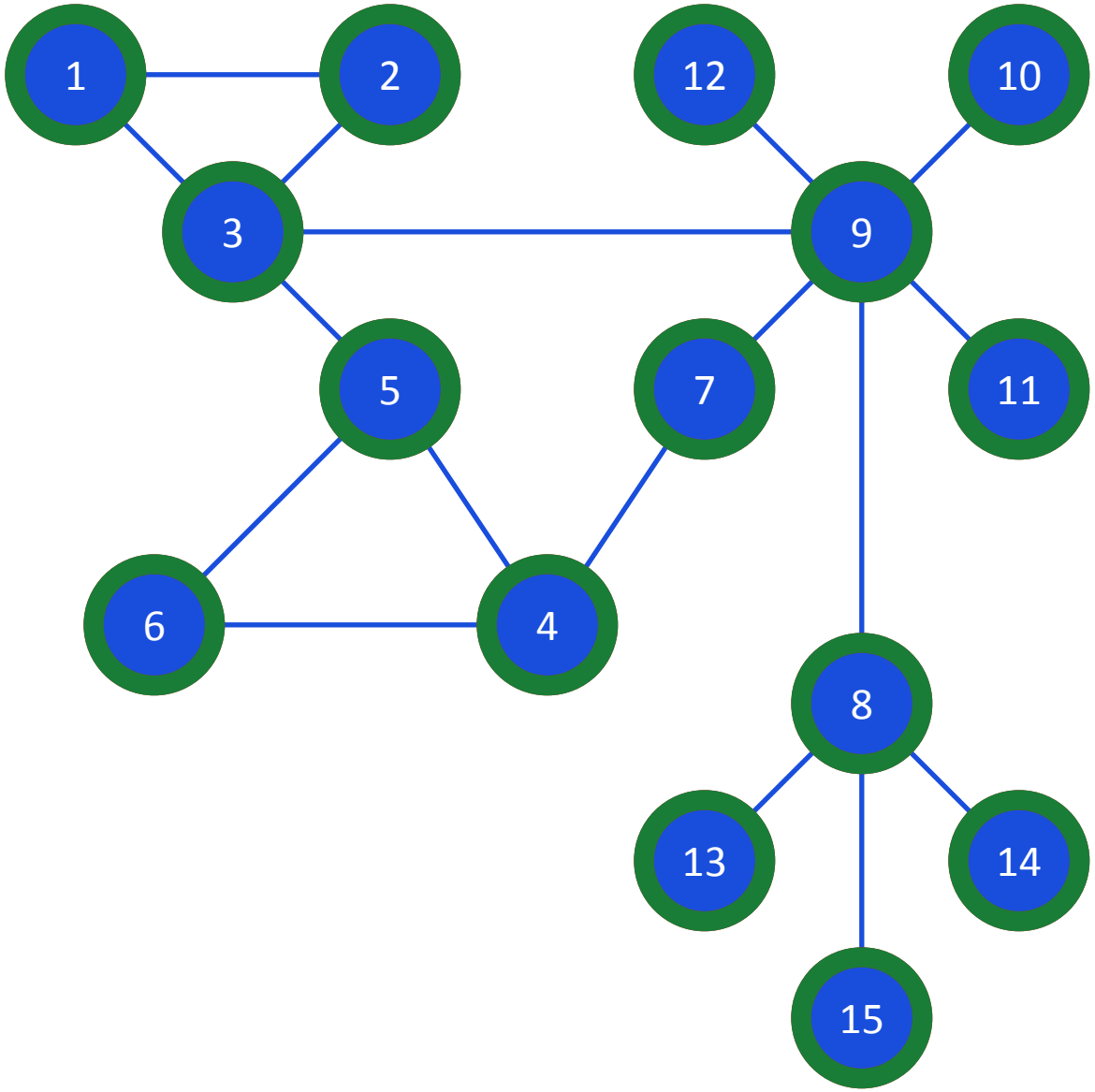
```
q[q_h] = v; q_h++;
```

③ Remove element (pop_front)

```
int res = q[q_t]; q_t++;
return res;
```

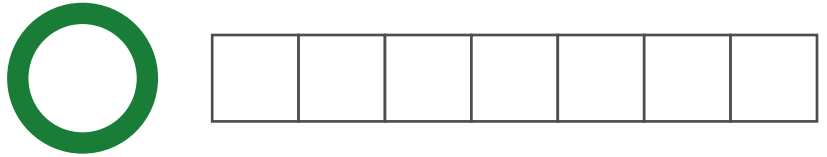
④ Is empty

```
return q_t == q_h;
```

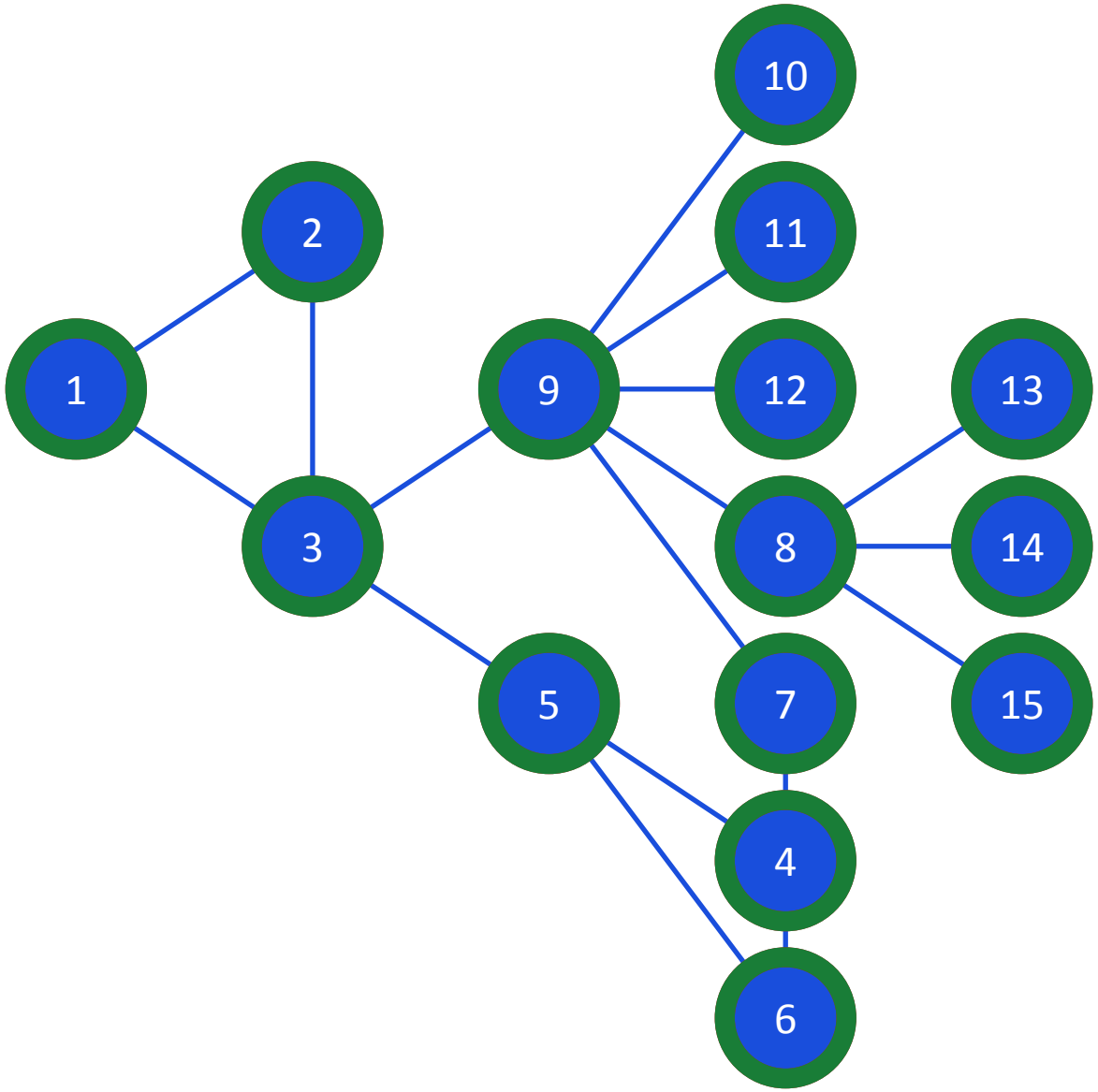


Fast Mode Normal Mode

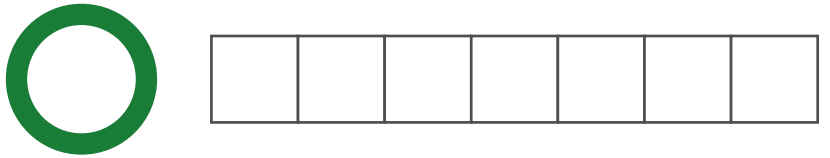
9 12 10 13 8 14 15 4



BFS in Graphs



Fast Automatic Mode

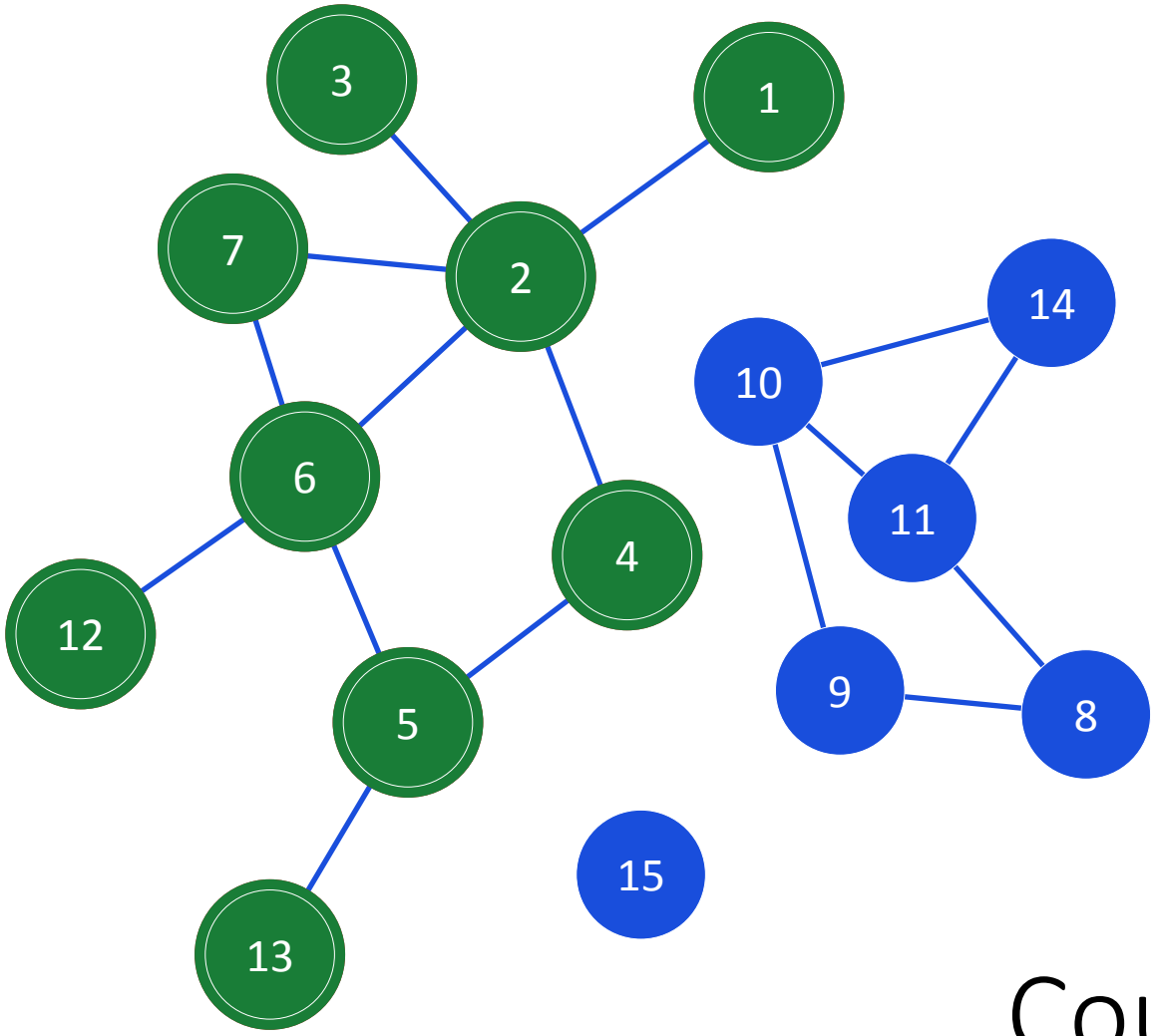


BFS in Graphs

Algorithms – BFS implementation

```
int size[128];
int a[128][128];
int q[128];
int qh, qt;
int visited[128];

void bfs(int v)
{
    qh = qt = 0;
    for (int i = 0; i < 128; i++)
        visited[i] = 0;

    visited[v] = 1;
    q[qh] = v; qh++;
 ...
```

```
    ...
    while (qh != qt)
    {
        v = q[qt]; qt++;
        for (int j = 0; j < size[v]; j++)
        {
            int b = a[v][j];
            if (!visited[b])
            {
                visited[b] = 1;
                q[qh] = b; qh++;
            }
        }
    }
}
```



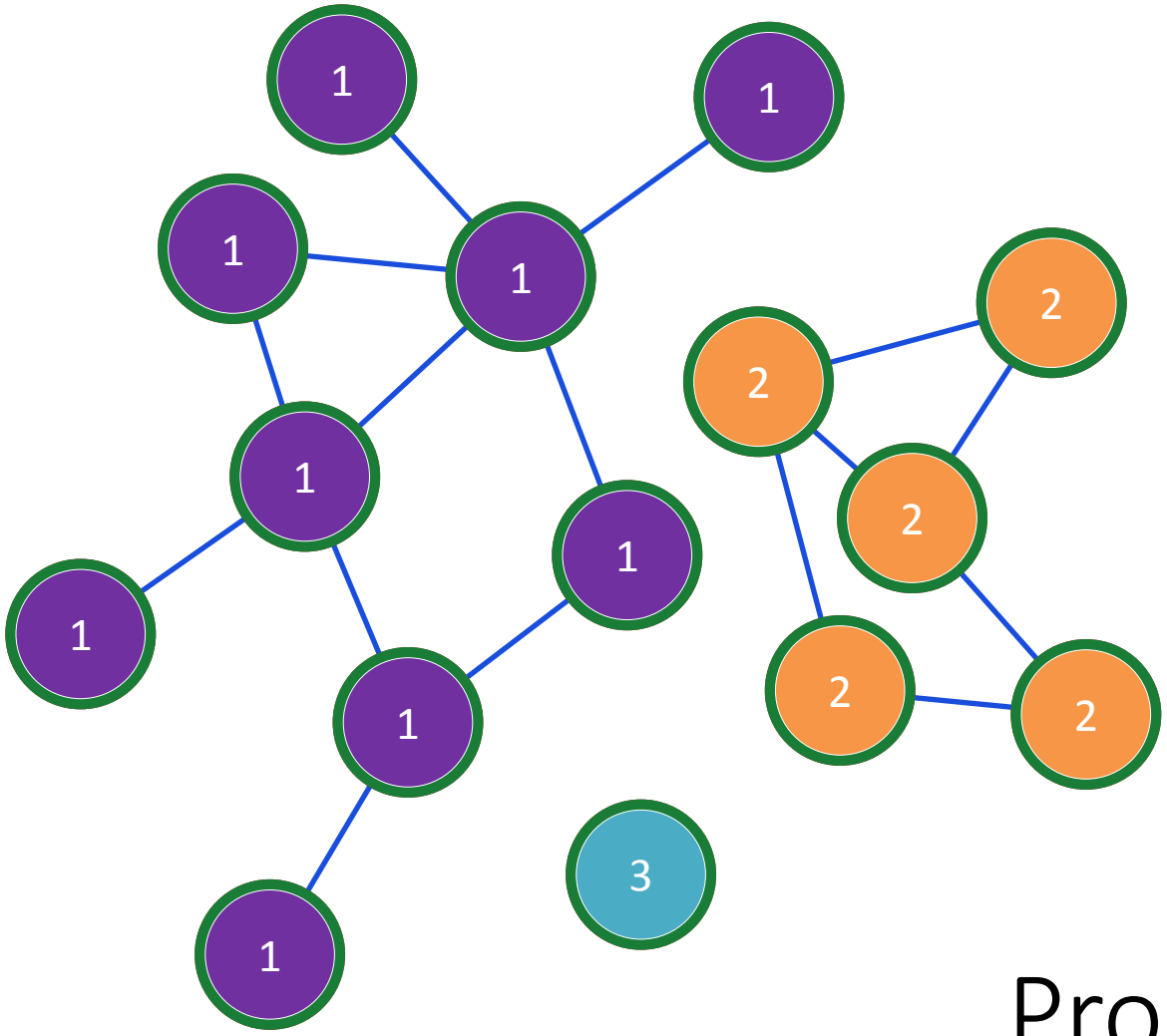

```
int q[128];
```

1	2	3	4	7	6	5	12	13

```
int qh, qt;
```

9

Count vertices in a connected component



9	5	1

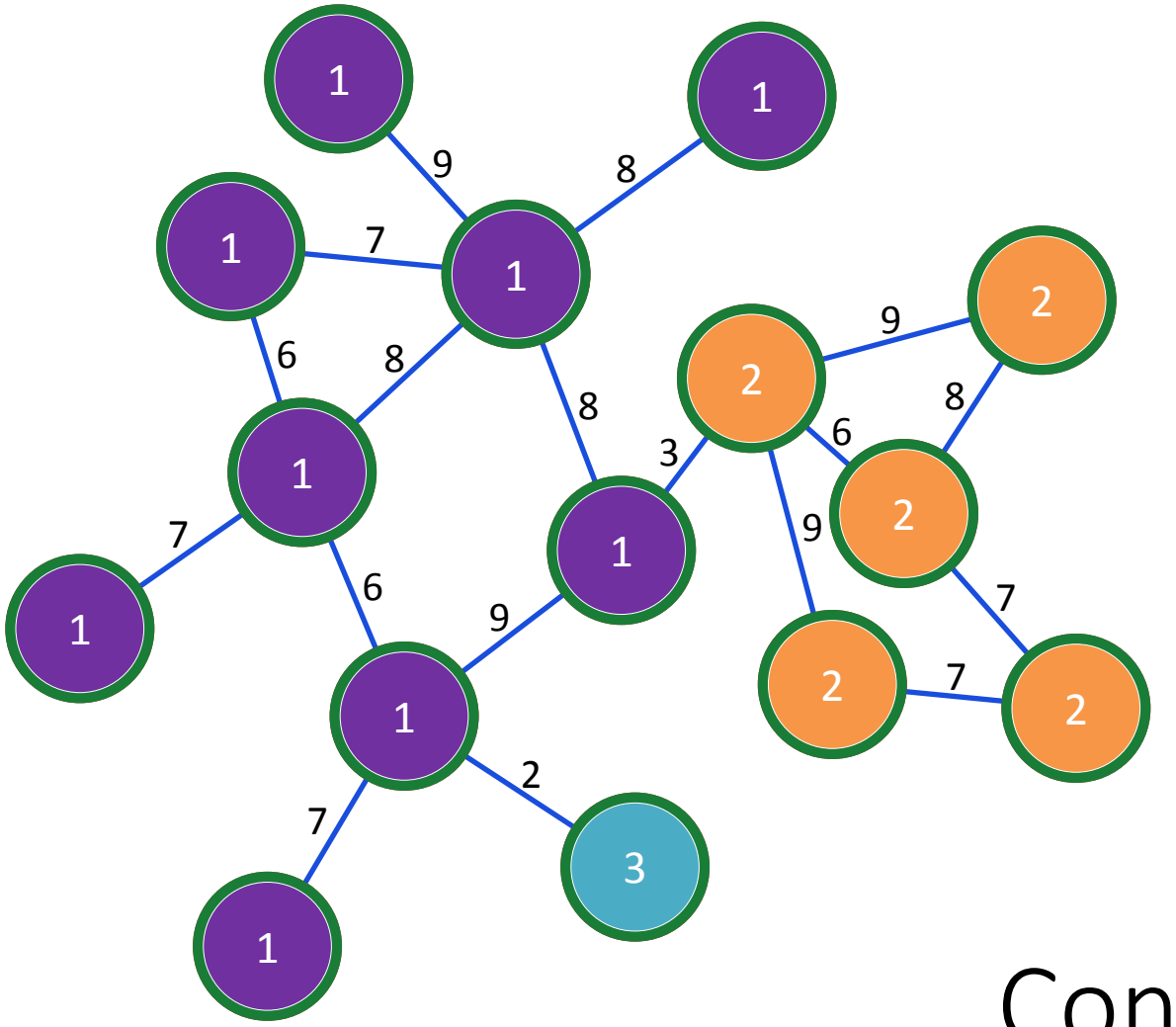
```
//int visited[128];
int current_component = 1;
int component_id[128];
int component_size[128];
void mark_components()
{
  for (int i = 0; i < 128; i++)
    component_id[i] = 0;
  for (int i = 1; i <= N; i++)
  {
    if (component_id[i] == 0)
    {
      bfs_comp(i);
      component_size[current_component] = qh;
      current_component++;
    }
  }
}
```

Process and mark all connected components

BFS for connected components

```
int current_component = 1;
int component_id[128];


...

void bfs_comp(int v)
{
    qh = qt = 0;
    for (int i = 0; i < 128; i++)
        visited[i] = 0;

    component_id[v] = current_component;
    q[qh] = v; qh++;
 ...
```

```
  while (qh != qt)
  {
     v = q[qt]; qt++;
     for (int j = 0; j < size[v]; j++)
     {
         int b = a[v][j];
         if (component_id[b] == 0)
         {
             component_id[b] =
                    current_component;
             q[qh] = b; qh++;
         }
     }
  }
}
```



9	5	1

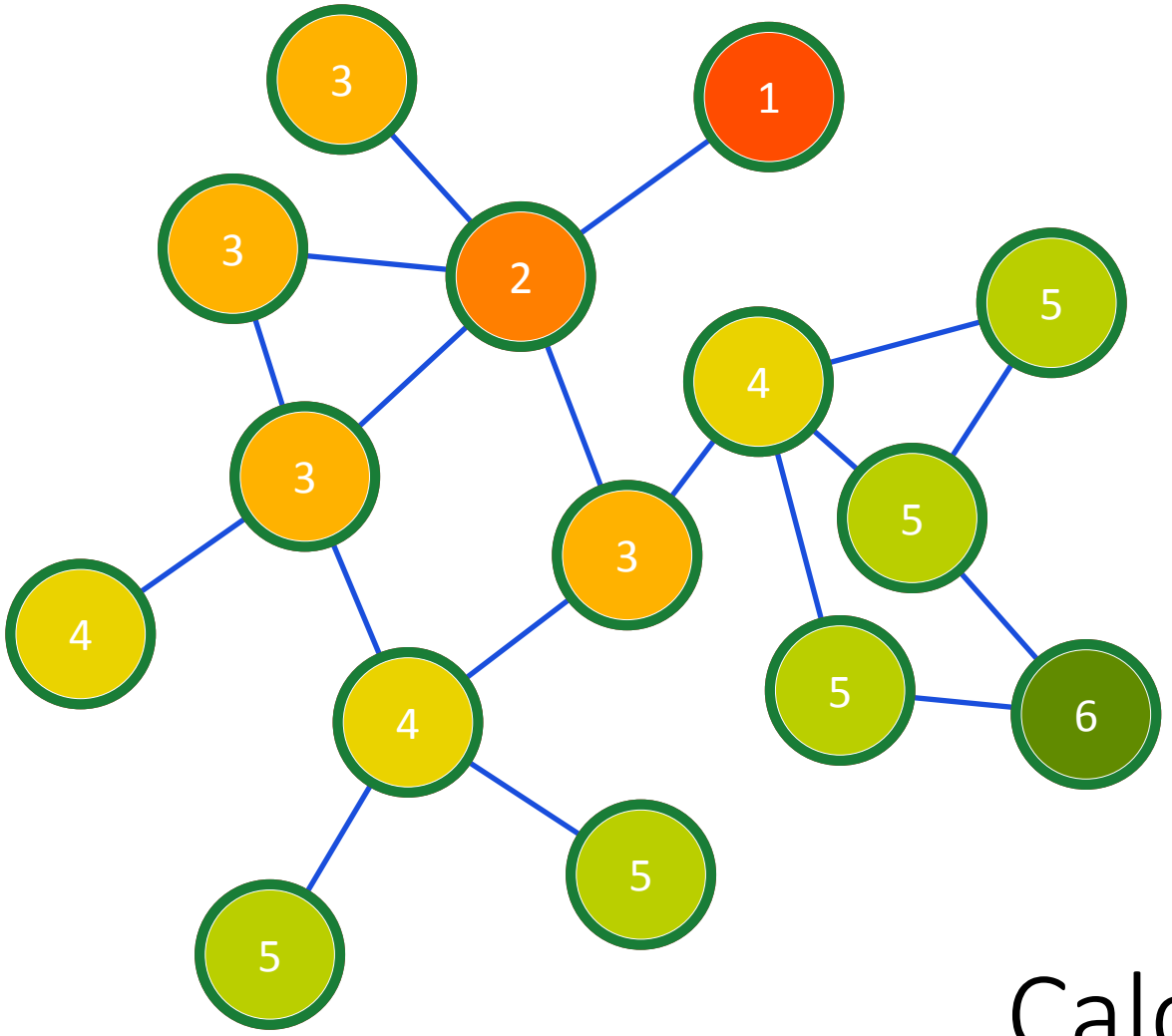
```
int weight[128][128];
...
int weight_limit = 4;
```

Conditional walk

BFS with conditionals

```
int current_component = 1;
int component_id[128];


...

void bfs_comp(int v)
{
    qh = qt = 0;


    component_id[v] = current_component;
    q[qh] = v; qh++;

    while (qh != qt) {
```

```
...
      v = q[qt]; qt++;
      for (int j = 0; j < size[v]; j++)
      {
          int b = a[v][j];
          if (component_id[b] == 0 &&
            val[v] == val[b])ight_limit)
          {
              component_id[b] =
                      current_component;
              q[qh] = b; qh++;
          }
      }
  }
}
```




```
//int visited[128];
int dist[128];
```

1	2	3	4	5	6	7	8	9
1	2	3	3	4	3	3	6	5

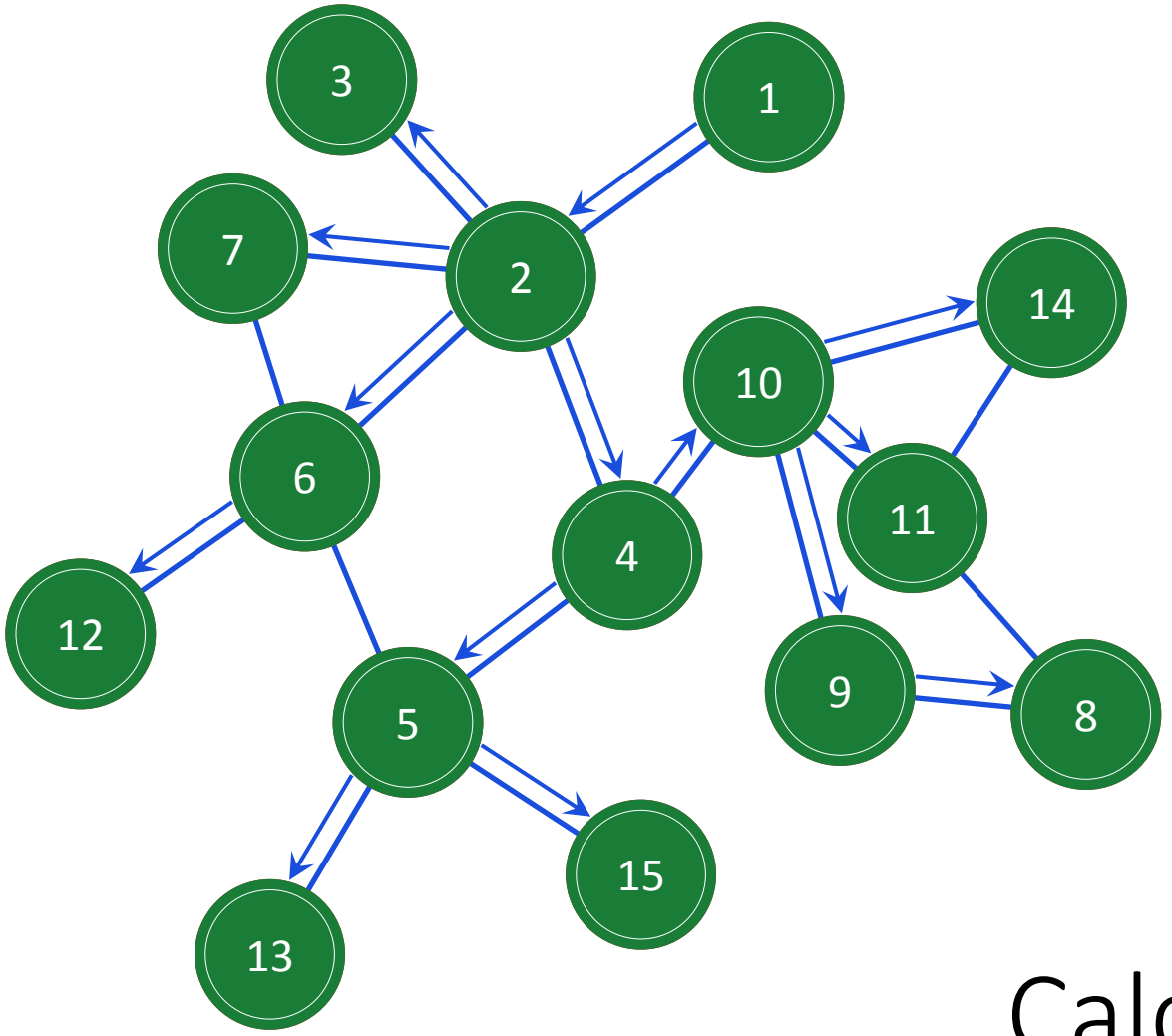
10	11	12	13	14	15
4	5	4	5	5	5

Calculate distance in number of edges

Algorithms – BFS with distance

```
int size[128];
int a[128][128];
int q[128];
int qh, qt;
int dist[128]; ];

void bfs(int v)
{
    qh = qt = 0;
    for (int i = 0; i < 128; i++)
        dist[i] = 0; 0;

    dist[v] = 1; 1;
    q[qh] = v; qh++;
 ...
```

```
    ...
    while (qh != qt)
    {
        v = q[qt]; qt++;
        for (int j = 0; j < size[v]; j++)
        {
            int b = a[v][j];
            if (!dist[b]) ])
            {
                dist[b] = dist[v] + 1;
                q[qh] = b; qh++;
            }
        }
    }
}
```




```
//int visited[128];
int parent[128];
```

1	2	3	4	5	6	7	8	9
1	1	2	2	4	2	2	9	10

10	11	12	13	14	15
4	10	6	5	10	5

Calculate shortest path

Algorithms – shortest path using BFS

```
int size[128];
int a[128][128];
int q[128];
int qh, qt;
int parent[128];

void bfs(int v)
{
    qh = qt = 0;
    for (int i = 0; i < 128; i++)
        parent[i] = 0;

    parent[v] = 1;;
    q[qh] = v; qh++;
 ...
```

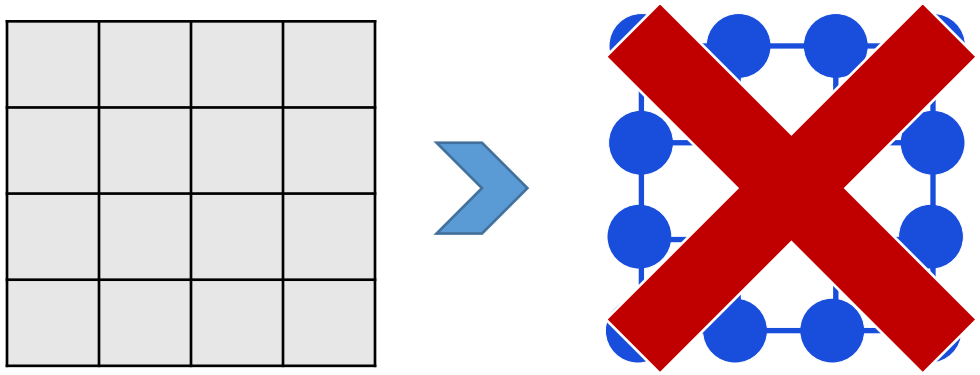
```
    ...
    while (qh != qt)
    {
       v = q[qt]; qt++;
       for (int j = 0; j < size[v]; j++)
       {
           int b = a[v][j];
           if (!parent[b])
           {
               parent[b] = v;;
               q[qh] = b; qh++;
           }
       }
   }
}
```

Algorithms – BFS usage

- By the nature BFS builds shortest path to all nodes from the given node. In order to build the path, we can track transitions in additional array when marking vertex as visited
- BFS and DFS can walk the graph based on condition. This allows to build the cluster – set of vertices, adjacent to each other and matched by some criteria. Example: phone booth task from January 2016 Advanced level test (ejudge contest 16, task “U”).
- BFS can calculate some function based on vertices and edges while walking the graph. Example: sum of all values associated with vertices.

Algorithms – graph in two-dimensional grid

If the graph is given as grid, we can convert it to graph and store as an Adjacency List.



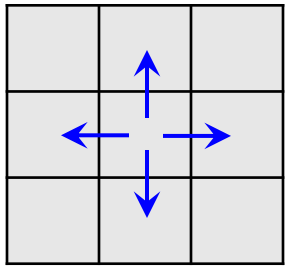
But why?

Use direct representation – two-dimensional array

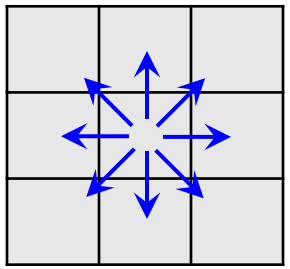
Algorithms – walking two-dimensional grid

Convenient way of walking the grid in contrast to adding numerous conditions and copy-pasting

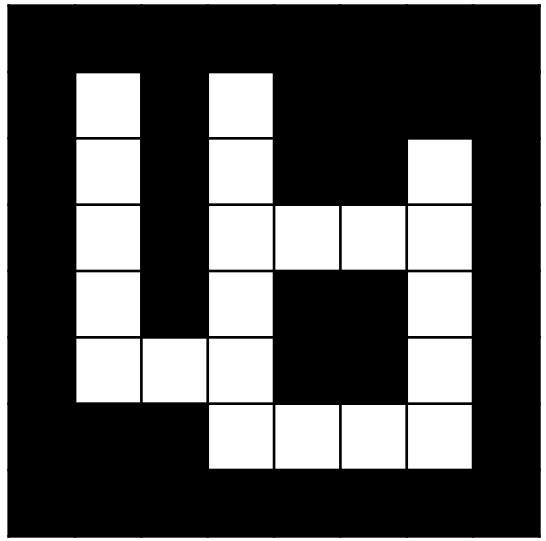
```
int dx[4] = {-1, 0, 1, 0};
int dy[4] = {0, -1, 0, 1};
for (k = 0; k < 4; ++k) process(i+dy[k], j+dx[k]);
```



```
int dx[8] = {-1, -1, 0, 1, 1, 1, 0, -1};
int dy[8] = {0, -1, -1, -1, 0, 1, 1, 1};
for (k = 0; k < 8; ++k) process(i+dy[k], j+dx[k]);
```



Algorithms – BFS in grid



Calculate the length of shortest path

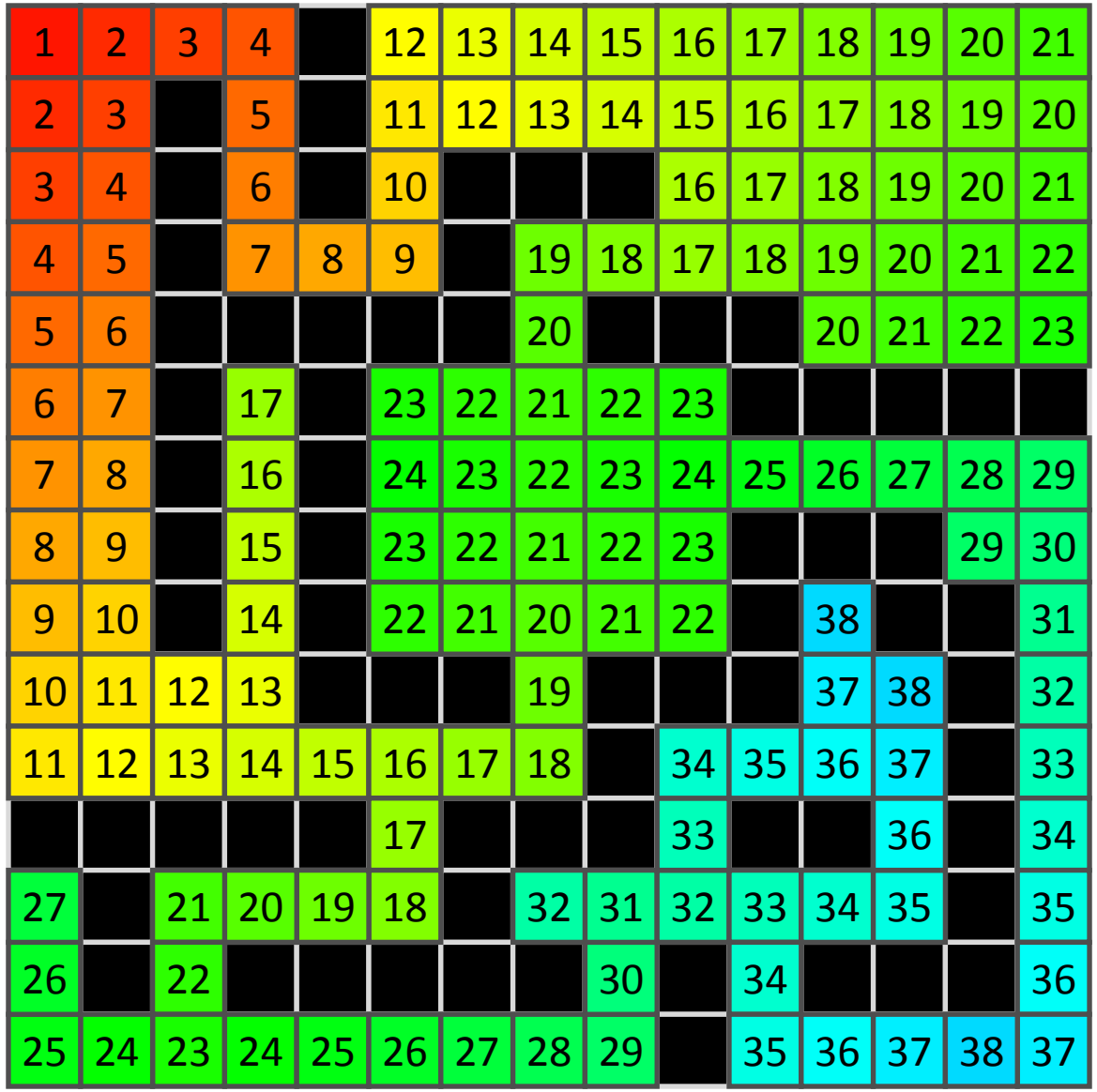
```
int a[128][128]; // input array
...

main() {

    for (int i = 0; i <= N+1; i++)
    {
        a[i][0] = 1;
        a[0][i] = 1;
        a[N+1][i] = 1;
        a[i][N+1] = 1;
    }
    ...
}
```

Algorithms – BFS in grid

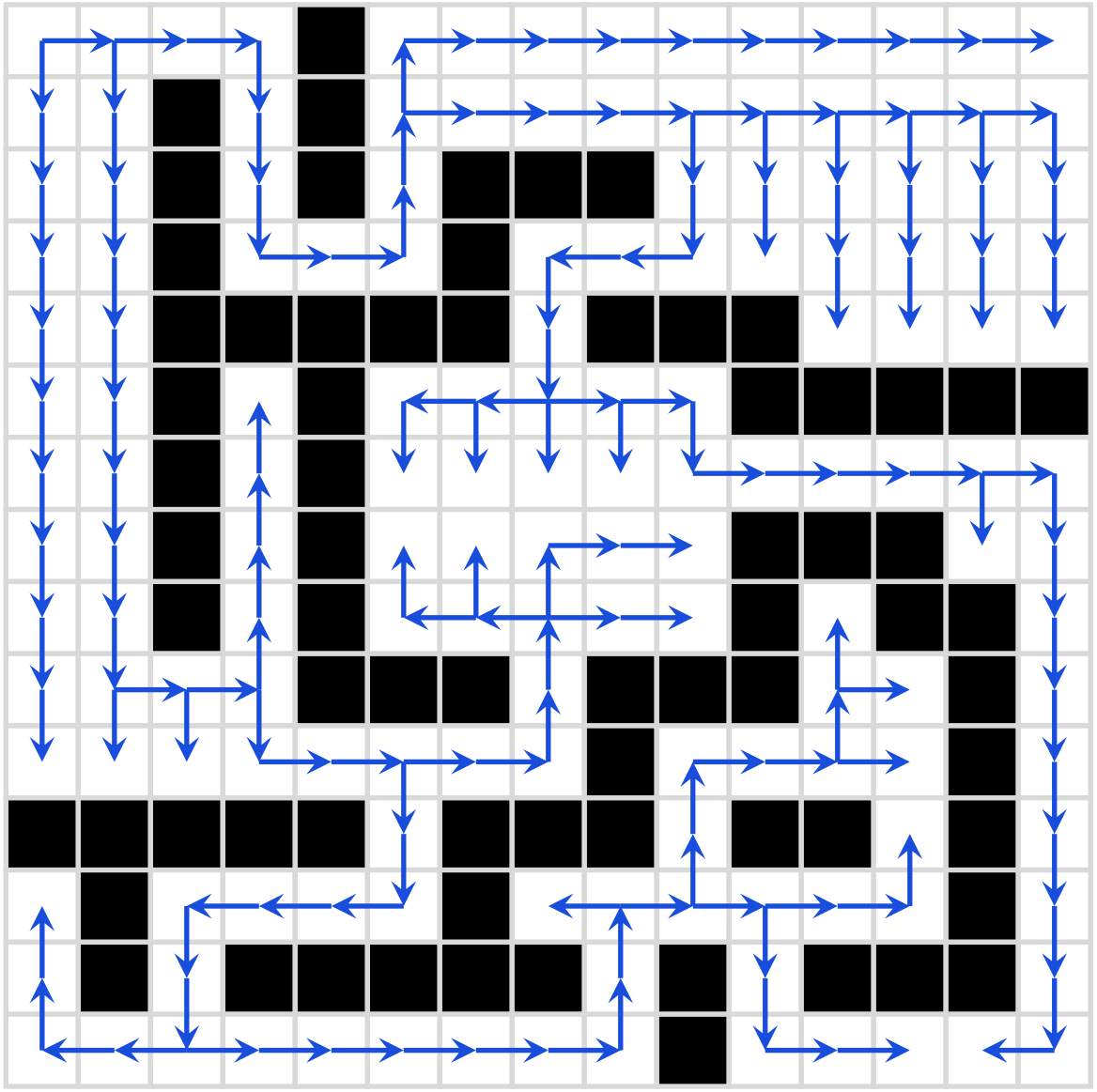
```
int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

int qx[128*128], qy[128*128];
int qh, qt;

int steps[128][128];
void bfs(int x, int y)
{
    qh = qt = 0;
    for (int i = 0; i < 128; i++)
    {
        for (int j = 0; j < 128; j++) steps[i][j] = 0;
    }

    steps[x][y] = 1;
    qx[qh] = x; qy[qh] = y; qh++;
...
```

Algorithms – BFS in grid

```
...
    while (qh != qt)
    {
        x = qx[qt]; y = qy[qt]; qt++;
        int d = steps[x][y];
        for (int k = 0; k < 4; k++)
        {
            int new_x = x + dx[k];
            int new_y = y + dy[k];
            if (a[new_x][new_y] != 0) continue;
            if (steps[new_x][new_y]) continue;
            steps[new_x][new_y] = d + 1;
            qx[qh] = new_x; qy[qh] = new_y; qh++;
        }
    }
...
```

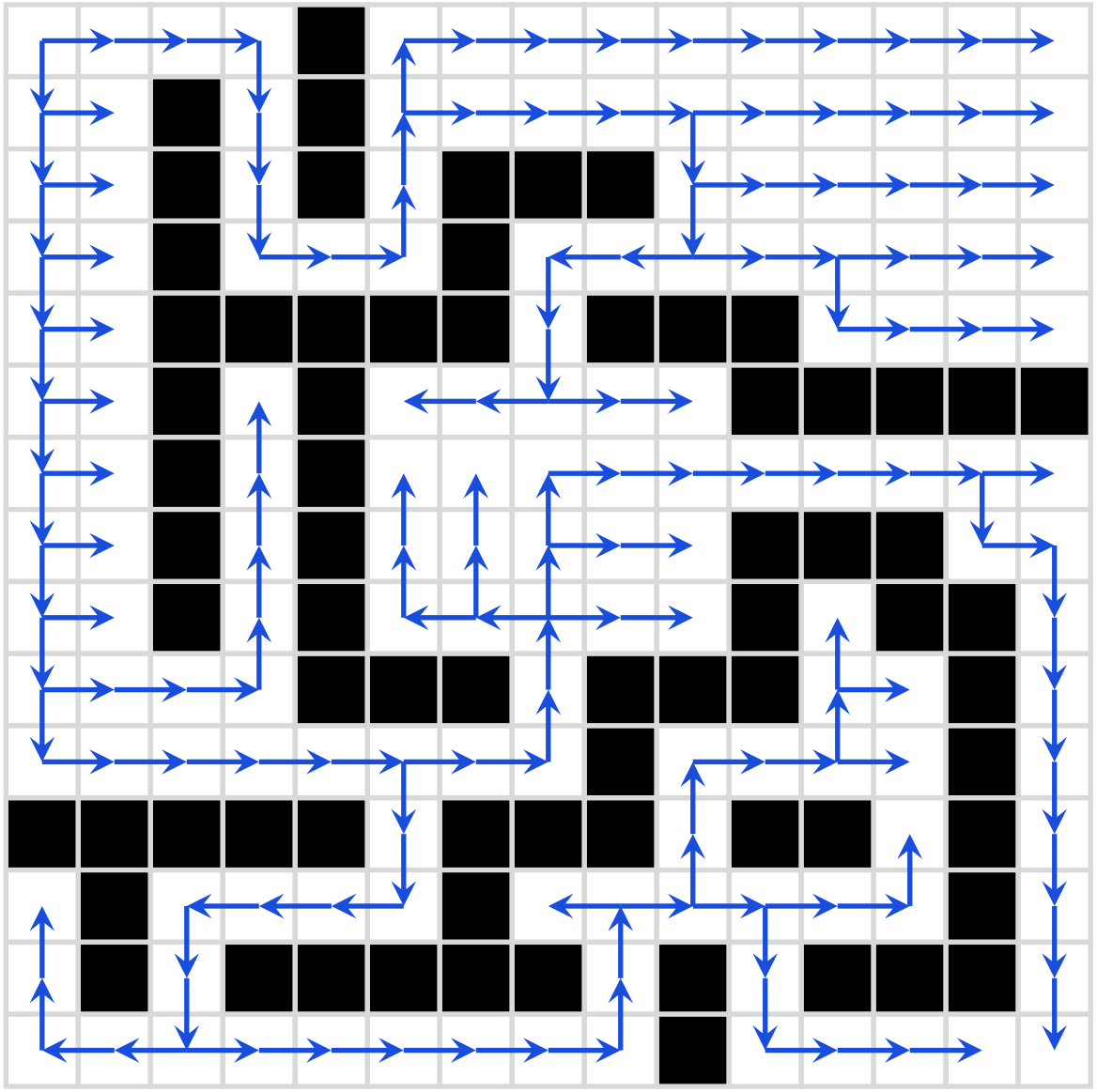


Distance in grid



Path in grid

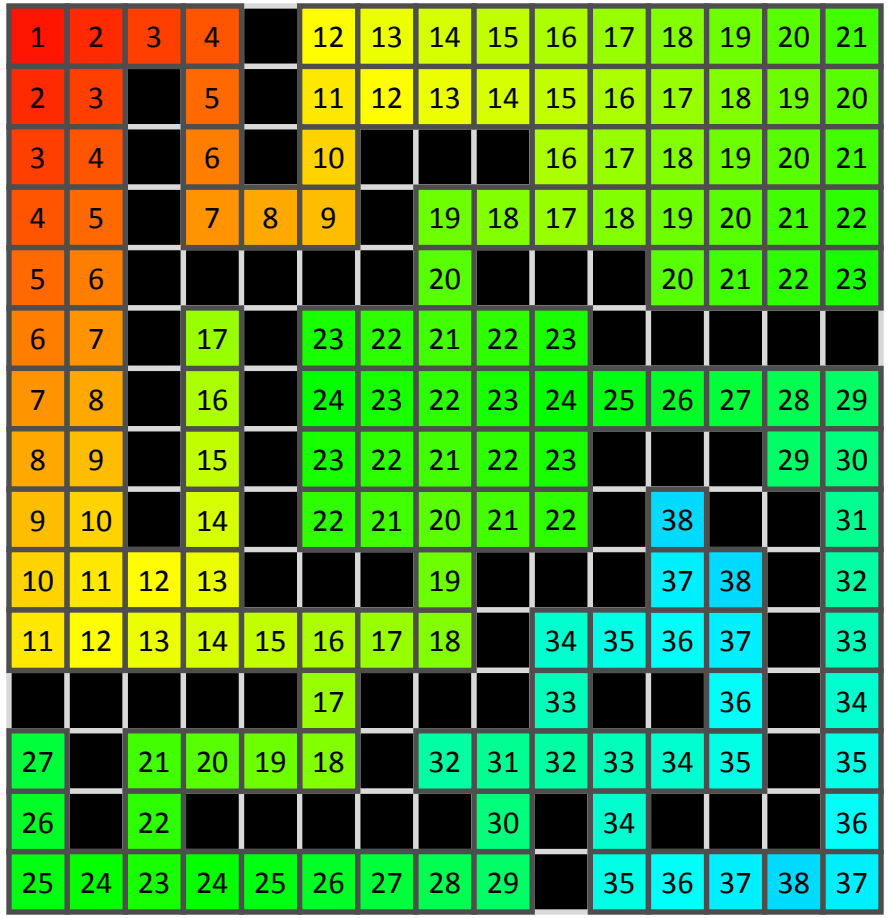
← ↑ → ↓



Path in grid

← ↑ ↓ →

Other algorithms

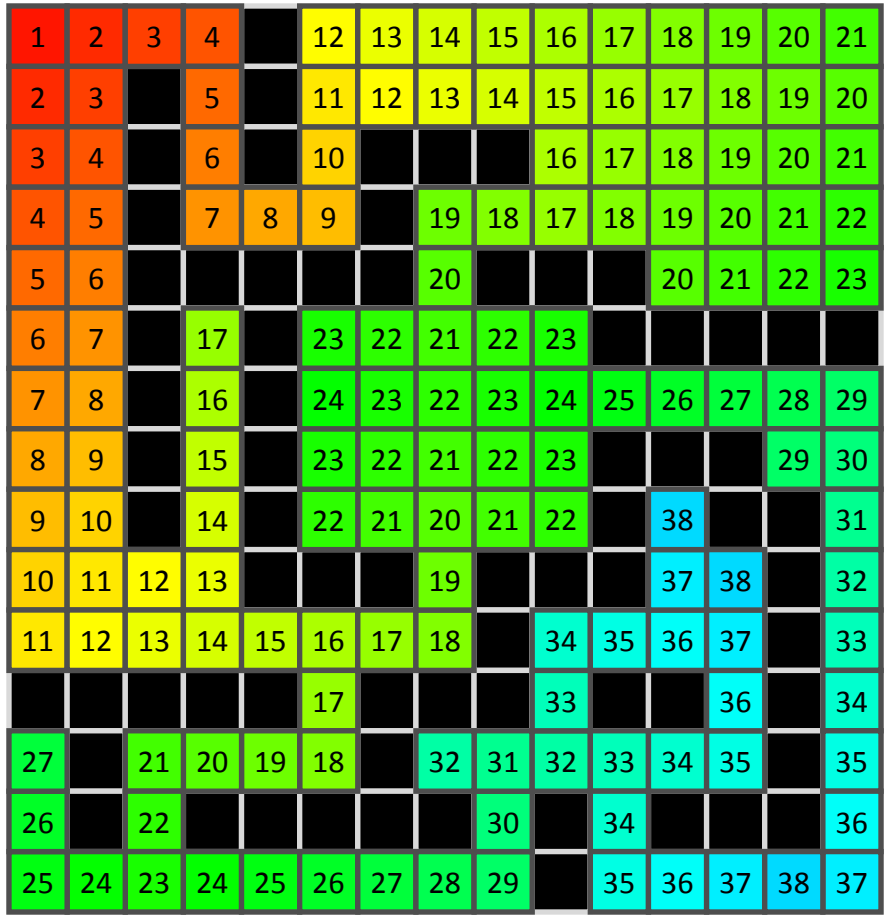


Lee algorithm (wave)

1. Repeat the next step N^2 times
2. For each of N^2 cells:
   - If the cell is empty, find its neighbor with smallest number and put this number+1 into the current cell

Don't use it! Complexity: O($\mathbf{N^4}$)

Other algorithms



BFS (compare result)

Use BFS! Complexity: O(N^2)

Advanced Algorithms

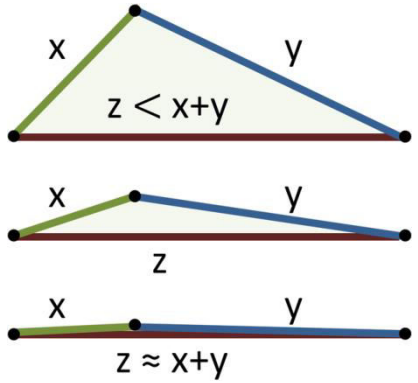
- Dynamic programming
- String searching algorithms
  - Knuth–Morris–Pratt
  - Boyer–Moore
  - Rabin–Karp
- Dijkstra's algorithm
- Maximum flow problem (Edmonds–Karp algorithm)
- Minimum spanning tree
- Lowest common ancestor

Are not needed for passing advanced level testing

Common mistakes

- Data types (integer overflow)
- Forgetting to submit solution
- Corner cases:
  - Most time consuming cases – check worst case before the submit
  - Smallest input data, largest input data (by size and by value),
  - Smallest answer, largest answer
  - “Degenerate” cases – qualitatively different from others

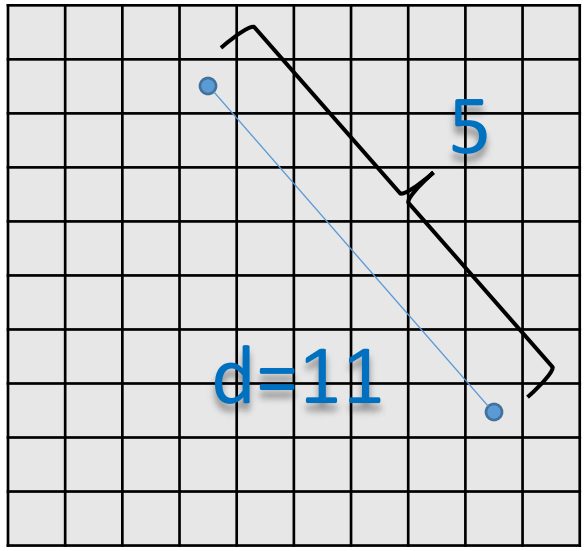
  e.g. triangle with collinear edges, circle with radius 0, etc
- Rules of thumb:
  - Test all cases, where some value or size is zero
  - Test any possibility of underflowing/overflowing



Debugging

- Printf debugging – easiest fastest way to debug for SW testing:
  - Print list of received arguments upon entering the recursive function
  - Print list of traversed elements in for cycles
  - Print input data before algorithm execution to make sure that everything's as expected (bad zeroization of structures, complicated data types etc)
- Additional named variables – it is easier to debug atomic operations on variables instead of convoluted expressions
- In case there is a mistake, but you can't find where:
  - **Debug parts of code separately!**
  - Random small data sample (and check manually by hand)
  - Exhaustive search for small data and automatic verification
  - Check corner cases (see "Common mistakes")

Starship (Summer 2015)



- Up to 100x100 matrix
- One entrance and one exit (two pairs of coords)
- Up to 5 portals (5 pairs of coords)
- Portal can be used in any direction
- Portal has its cost
- Distance is calculated as:

$$d = |x_1-x_2| + |y_1-y_2|$$

Task: find the length of shortest path from entrance to exit with our without using portals

Task examples