Backtracking

Generate permutations

E-OLYMP [2169. Permutations](#) Given a positive integer *n*, print all permutations of the integers from 1 to *n* in lexicographical order.

► In the problem you should generate all permutations of numbers from 1 to *n*. This can be done, for example, using the ***next_permutation*** function.

Use array m to generate permutations.

```
int m[10];
```

Read the value of *n*. Initialize the array m with the initial permutation 1 2 3…. *n* starting from the first index.

```
scanf("%d",&n);
for(i = 1; i <= n; i++) m[i] = i;
```

Using the ***next_permutation*** function, generate all permutations: from the lexicographically smallest to the lexicographically largest.

```
do
{
```

Print the next permutation on a separate line.

```
  for(i = 1; i <= n; i++)
    printf("%d ",m[i]);
  printf("\n");
} while(next_permutation(m+1,m+n+1));
```

E-OLYMP [1533. Anagram generation](#) You are to write a program that has to generate all possible words from a given set of letters.

Example: Given the word "*abc*", your program should – by exploring all different combination of the three letters – output the words "*abc*", "*acb*", "*bac*", "*bca*", "*cab*" and "*cba*".

In the word taken from the input file, some letters may appear more than once. For a given word, your program should not produce the same word more than once, and the words should be output in alphabetically ascending order.

► 1. What is the difference between *lexicographic* and *alphabetical* sorting?

2. Consider the string zAZaaZ. Sort the letters in alphabetical and lexicographic order.

3. What STL function can be used to generate permutations of all letters in a word?

4. The function ***sort***, by default, sorts the letters in a word in lexicographic order. How to implement with it an alphabetical sorting?

5. Implement a comparator for alphabetical sort `int lt(char a, char b)`, which is passed as the third argument to the ***sort*** function.

6. How to find the alphabetically smallest permutation of letters in the string *s*?

Sort the characters of the input string in ascending order. Use the built-in ***next_permutation*** function to generate all permutations. However, you should write your own function to compare the characters. In standard (lexicographic) comparison, any uppercase letter is less than any lowercase letter. That is, when sorting letters a, A, z, Z, r, R, we get the word ARZarz. In this problem you should sort (and generate permutations) in accordance with the alphabetical order AaBbCc… Zz, so it is necessary to obtain AaRrZz from the letters a, A, z, Z, r, R.

For the string **aAb** (1-st test case) the smallest permutation would be **Aab**, and the biggest would be **baA**.

The function ***lt*** will be used for sorting and generating permutations. It compares two characters according to alphabetical order AaBbCc ... Zz.

```
int lt(char a, char b)
{
  if (toupper(a) != toupper(b)) return (toupper(a) < toupper(b));
  return (a < b);
}
```

Read the input string, calculate its length, and sort the characters alphabetically.

```
scanf("%s", &s);len = strlen(s);
sort(s,s+len,lt);
```

Print the current anagram (permutation of characters) and generate the next one until its possible.

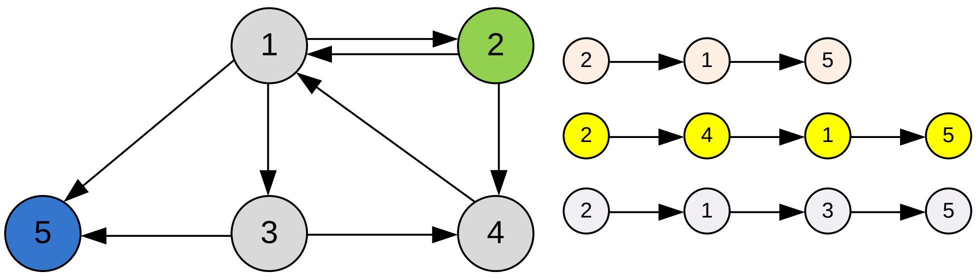
```
do {
  printf("%s\n",s);
} while(next_permutation(s,s+len,lt));
```

Graphs

E-OLYMP **122. Mountain routes** The mountain tourist resort consists of *n* hostels, connected with *k* mountain routes (other routes in mountains are dangerous). Any route between two hostels takes 1 day. The travel group, located in the hostel *a*, is going to go to the hostel *b* for no more than *d* days. How many different routes (without cycles) between *a* and *b* exist?

► There is a directed graph where you need to find the number of paths between two vertices. We'll solve the problem by exhaustive search. Starting from the vertex *a*, using the depth first search strategy, we'll go to the vertex *b*. Next, using backtracking, iterate over all possible paths from *a* to *b*, counting their number in the variable *res*. The length of all found paths must not exceed *d*.

In the sample given between the hostels 2 and 5 there are 3 acyclic paths of length at most 3.



Store the adjacency matrix of the graph in the array *g*.

```
#define MAX 51
int g[MAX][MAX], used[MAX];
```

Start the depth first search from the vertex *v*. It is known that the length of the path from *a* to *v* already equals to *depth*.

```
void dfs(int v, int depth)
{
  if (depth > d) return;
```

If we come to vertex *b*, then another route has been found. Increase the number of paths *res* by one and return back.

```
  if (v == b)
  {
    res++;
    return;
  }
```

Mark the vertex *v* as visited.

```
  used[v] = 1;
```

We are looking for a vertex *i*, where we can go from *v*.

```
  for(int i = 1; i <= n; i++)
    if (g[v][i] && !used[i]) dfs(i,depth+1);
```

Since the iteration of vertices is performed with backtracking, the vertex *v* should be set as not visited.

```
  used[v] = 0;
}
```

The main part of the program. Read the input data. Read the directed graph into the adjacency matrix *g*.

```
scanf("%d %d %d %d %d",&n,&k,&a,&b,&d);
memset(g,0,sizeof(g));
memset(used,0,sizeof(used));
```

```
for(i = 0; i < k; i++)
{
  scanf("%d %d",&a1,&a2);
  g[a1][a2] = 1;
}
```

Run the depth first search from the vertex *a*.

```
res = 0;
dfs(a,0);
```

Print the number of found routes.

```
printf("%d\n",res);
```

E-OLYMP **1540. 23 out of 5** Is it possible to find an arithmetic expression consisting of five given numbers that will yield the value 23. For this problem you can use only three arithmetic expressions: +, -, *. Assume that all these operations have the same priority and executed sequentially from left to right.

► Generate all permutations of input numbers. For each permutation between the numbers place all possible signs of arithmetic operations and calculate the resulting expressions. If at least one expression has the value of 23, then set the variable *flag* to 1 and end the processing of the current test.

Store the input five numbers in an integer array a. The variable *flag* takes the value of 1 if an expression with the value 23 is found, otherwise *flag* = 0.

```
int a[5], flag;
```

Function ***RunSum*** evaluates the value of an expression which operands are in array a. The signs of various operations are inserted between the operands and the resulting expressions are computed using the *backtracking* technique. If the value of one of expressions is 23, then function returns 1, otherwise 0.

```
int RunSum(int Sum, int index)
{
  if (index == 5)
    if (Sum == 23) return 1; else return 0;

  if (RunSum(Sum+a[index],index+1)) return 1;
  if (RunSum(Sum-a[index],index+1)) return 1;
  if (RunSum(Sum*a[index],index+1)) return 1;
  return 0;
}
```

The main part of the program. Read five numbers into array a. Start the procedure for generating all permutations. Since the ***next_permutation*** function generates permutations in lexicographic order, the smallest permutation must be the first. The smallest permutation is the one in which the numbers form a non-decreasing sequence. That is, before generating the permutations, the numbers in the array *a* should be sorted in non decreasing order (if this is not done, then not all permutations will be considered).

```
while(scanf("%d %d %d %d %d",
      &a[0],&a[1],&a[2],&a[3],&a[4]),a[0]+a[1]+a[2]+a[3]+a[4])
{
  sort(a,a+5);
  flag = 0;

  do{
```

For each permutation, run the ***RunSum*** function, which finds out whether it is possible to arrange the operation signs between the numbers of this permutation so as to obtain the value 23.

```
    if (found = RunSum(a[0],1)) break;
  } while(next_permutation(a,a+5));
```

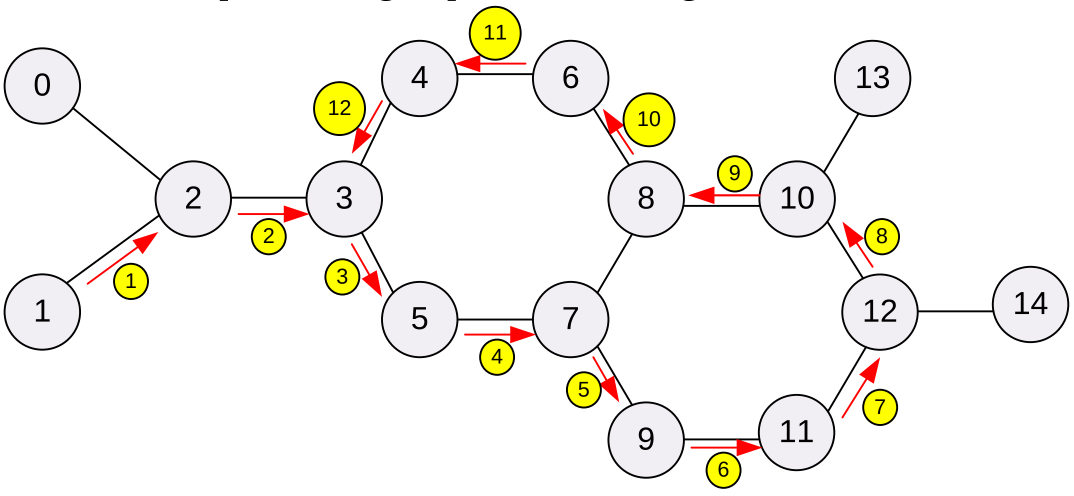
If the variable *flag* is set to 1, then print "Possible", otherwise print "Impossible".

```
  if (flag) printf("Possible\n"); else printf("Impossible\n");
  memset(a,0,sizeof(a));
}
```

E-OLYMP **10223. The Settlers of Catan** Undirected graph is given. Find the longest pah in it.

► Run the depth first search from each vertex. Generate all possible paths using backtracking. The limitation on the number of vertices ($n \le 25$) allows you to keep within the time limit. Find the length of the longest path.

In the second sample the longest path has the length 12.



Declare the adjacency matrix *mas* of the graph.

```
#define MAX 26
int mas[MAX][MAX];
```

Enter the vertex *i*. The current distance from the starting vertex to *i* is *depth*.

```
void run(int i, int depth)
{
```

The variable *best* maintains the value of the longest path.

```
  if (depth > best) best = depth;
```

Find into which vertex *j* can we go from *i*.

```
  for (int j = 0; j < n; j++)
    if (mas[i][j])
    {
```

Delete the edge (*i*, *j*) and continue search from the vertex *j*.

```
      mas[i][j] = mas[j][i] = 0;
      run(j, depth + 1);
```

Upon return from the function ***run*** restore the edge (*i*, *j*).

```
      mas[i][j] = mas[j][i] = 1;
    }
}
```

The main part of the program. Process several test cases.

```
while (scanf("%d %d", &n, &m), n + m)
{
  memset(mas, 0, sizeof(mas));
```

Read the input data. Construct the graph.

```
  for (i = 0; i < m; i++)
  {
    scanf("%d %d", &a, &b);
    mas[a][b] = mas[b][a] = 1;
  }
```

Run the depth first search from each vertex. Vertices are numbered from 0 to *n* – 1. In the variable *best* compute the maximum length of the path.

```
  best = 0;
  for (i = 0; i < n; i++)
    run(i, 0);
```

Print the answer.

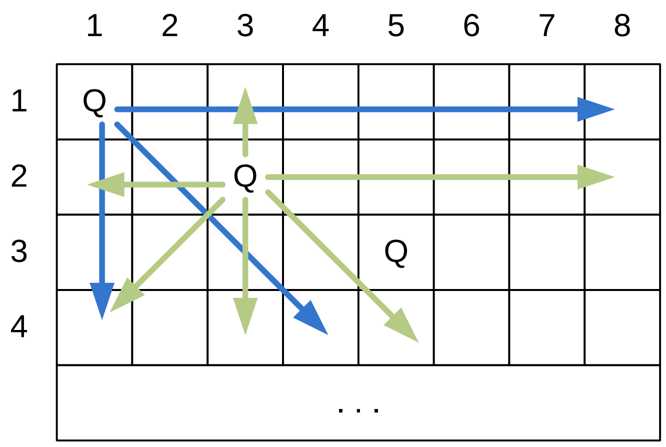
```
  printf("%d\n", best);
}
```

E-OLYMP [10236. N Queens - placement](#) Given a chess board having $n * n$ cells. You need to place n queens on the board in such a way that no queen attacks any other queen.

► To store the information about the arrangement of queens on the chessboard, we'll use a two-dimensional array *mat*:

- mat[i][j] = 1, if the queen is in position (i, j);
- mat[i][j] = 0, if there is no queen is in position (i, j);

Initially there are no queens on the board (mat[i][j] = 0). Take the first position (1, 1) and put the queen on it (mat[1][1] = 1). Look for the next position, for example (2, 3), that is not attacked by the first queen, and put the second queen there. Next, we look for a position that is not attacked by two placed queens. Such position, for example, can be (3, 5), where we put the third queen.



Iterate through the positions (i, j) further and look for one where you can put the next queen. When all the squares of the board have been examined, and all n queens have not been placed, start return back – backtracking. Try to change the position of the last queen by finding another suitable place for it. When all the positions of this queen are considered, change the position of the previous queen, and so on, continue the search with backtracking until all n queens have been placed.

In the array *mat*, we'll generate a chessboard.

```
int mat[11][11];
```

Function ***attacked*** checks if at least one queen from position (i, j) will be attacked.

```
int attacked(int x, int y)
{
  if (mat[x][y])return 1;
  for (int i = 1; i <= n; i++)
  {
    if (y - i >= 1 && mat[x][y - i]) return 1;
    if (y + i <= n && mat[x][y + i]) return 1;
    if (x - i >= 1 && mat[x - i][y]) return 1;
    if (x + i <= n && mat[x + i][y]) return 1;
    if (x - i >= 1 && y - i >= 1 && mat[x - i][y - i]) return 1;
    if (x + i <= n && y + i <= n && mat[x + i][y + i]) return 1;
    if (x + i <= n && y - i >= 1 && mat[x + i][y - i]) return 1;
    if (x - i >= 1 && y + i <= n && mat[x - i][y + i]) return 1;
  }
  return 0;
}
```

Function ***print*** prints the chessboard.

```
void print(void)
{
  for (int i = 1; i <= n; i++)
  {
    for (int j = 1; j <= n; j++)
      printf("%d ", mat[i][j]);
    printf("\n");
  }
}
```

Function ***solve*** places x more queens on the board so that they do not attack each other.

```
int solve(int x)
{
```

If $x = 0$, all n queens are placed. Print the state of the board.

```
  if (x == 0)
  {
    print();
    return 1;
  }
```

Iterate over the positions (i, j), check is it possible to put a queen in (i, j) so that it does not attack already placed ones.

```
  for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
  {
    if (attacked(i, j) == 1) continue;
```

Put the queen in position (i, j).

```
    mat[i][j] = 1;
```

Place $x - 1$ queens on the rest of the board.

```
    if (solve(x - 1)) return 1;
```

Remove the queen from position (i, j) and make a move backward (backtracking)

```
    mat[i][j] = 0;
  }
  return 0;
}
```

The main part of the program. Read the board size n.

```
scanf("%d", &n);
```

Arrange n queens. If the placement is possible, then it will be printed. Otherwise, print the message “Not possible”.

```
if (!solve(n)) puts("Not possible");
```

E-OLYMP **10242. Knight route** Given a $n * n$ board with the Knight placed in the first row and first column of an empty board. Moving according to the rules of chess knight, visit each square exactly once.

Print the order of each the cell in which they are visited.

► The knight’s moves are numbered from 1 to n^2. Declare a two dimensional array *sol*, where we’ll generate knight moves. Start from the point (0, 0), for which we set sol[0][0] = 1 (the first move of the knight). Next, iterate the positions where the chess knight can go. If there is a position where the knight can go, then make a move there and continue the search from this position. If it is impossible to make a move from the current position, then make a move backward (backtracking) and continue the search. The search ends when all numbers from 1 to n^2 have been placed on the chessboard.

The maximum size of the chessboard is 8. In the *sol* array generate the moves of the knight.

```
#define MAX 9
int sol[MAX][MAX];
```

The arrays *xMove* and *yMove* define the possible moves of the knight. If the knight is in the cell (x, y), then with one move it can go to the cell $(x + xMove[i], y + yMove[i])$, where $0 \le i \le 7$.

```
int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };
```

The function ***isSafe*** checks if cell (x, y) is free. That is, if the knight can go to cell (x, y). The answer is affirmative if the x and y coordinates are within the chessboard (the cells are numbered from 0 to $n – 1$) and the knight has not entered the cell (x, y) yet (sol[x][y] = -1).

```
int isSafe(int x, int y)
{
  return (x >= 0 && x < n && y >= 0 && y < n && sol[x][y] == -1);
}
```

The function ***printSolution*** prints a chessboard with knight moves.

```
void printSolution(void)
{
  for (int x = 0; x < n; x++)
  {
    for (int y = 0; y < n; y++)
```

```
      printf("%2d ", sol[x][y]);
    printf("\n");
  }
}
```

The function ***solveKTUtil*** makes a move number *movei* to the cell (*x*, *y*). Continue the search with backtracking from the cell (*x*, *y*) until n^2 moves are made.

```
int solveKTUtil(int x, int y, int movei)
{
```

The move number *movei* is made to the cell (*x*, *y*).

```
  sol[x][y] = movei;
```

If n^2 moves are made, then finish the search.

```
  if (movei == n * n) return 1;
```

Iterate through the cells where you can go from (*x*, *y*). From the cell (*x*, *y*) in one move the knight can go to ($x + xMove[i]$, $y + yMove[i]$), where $0 \le i \le 7$.

```
  for (int k = 0; k < 8; k++)
  {
    int next_x = x + xMove[k];
    int next_y = y + yMove[k];
```

If the knight did not visit the cell (*next_x*, *next_y*) yet, then recursively run the search from it. In (*next_x*, *next_y*) the move number *movei* + 1 is performed.

```
    if (isSafe(next_x, next_y))
    {
      if (solveKTUtil(next_x, next_y, movei + 1) == 1) return 1;
    }
  }
```

If all the moves of the knight are iterated, but it was not possible to make a move to the free square, make a move backward (backtracking), free the square (*x*, *y*) (sol[*x*][*y*] = -1).

```
  sol[x][y] = -1;
  return 0;
}
```

The function ***solve*** builds a knight's route using backtracking. The function returns 0 if the route cannot be built. If there are several solutions, the function generates one of them.

```
int solve()
{
```

Initialize the resulting matrix *sol*.

```
  for (int x = 0; x < n; x++)
```

```
    for (int y = 0; y < n; y++)
      sol[x][y] = -1;
```

The chess knight starts its path from the upper left corner. The function ***solveKTUtil*** starts path generation from cell (0, 0). The cell (0, 0) contains the number 1.

```
    if (solveKTUtil(0, 0, 1) == 0)
    {
      printf("Solution does not exist");
      return 0;
    }
    else
      printSolution();

    return 1;
  }
```

The main part of the program. Read the size of the chessboard *n*. Run the function ***solve*** – solving the problem.

```
scanf("%d", &n);
solve();
```