A Guide to NOI for Beginners (Draft)

By: Si Chenglei

Email: sichenglei1125@gmail.com

Reference Book: http://www.ituring.com.cn/book/1044

Practice Solutions: https://github.com/yogykwan/acm-challenge-workbook/tree/master/src

Foreword

This guide is designed specifically for those who already learned the basics of programming. I assume that you already know how to code in C++ and understand the concept of time complexity, as well as some other basic data structures like linked list, stack, queue, BST. (If not, there are lots of great tutorials/MOOCs online.)

We understand that it is hard to teach oneself competitive programming. So I made this guide and recorded a series of videos explaining it. This guide is meant for you to quickly get started on solving NOI problems and get familiar with some important algorithms. There is no guarantee that you can get a medal at NOI after reading this. But I am sure you can improve your knowledge and problem solving skills if you read this guide carefully and do lots of practice for every topic. Feel free to contact me if you find any mistakes or have doubts, I am always happy to discuss with you.

1. Time Complexity

One important thing in NOI is time complexity. There is always a time limit for the problems and if your algorithm is too slow, you will get Time Limit Exceeded (TLE).

Usually the time limit is 1 second. You can substitute the maximum possible number of data witin the given range to yur algorithm's time complexity to get an exstimated number of operations needed. For example, if your algorithm is O(n^2) and the data range is $n \leq 1000$, then the max operations needed is around 10^6. You can check whether you will get TLE by the checking the estimated number of operations according to:

- 10^6: no problem
- 10^7: should be ok
- 10^8: may not work

Now we will use a simple example to see how this works in practice.

E.g.1 Pick Numbers

You are given n different integers $k_1, \ldots, k_n$. You need to pick 4 numbers from them. You can pick the same number any times. If the sum of the 4 numbers you pick is m, you output YES, otherwise output NO.

$1 \leq n \leq 1000, 1 \leq m \leq 10^8, 1 \leq k_i \leq 10^8$

Solution 1: Brute Force

```cpp
#include <cstdio>
using namespace std;

const int MAX_N = 1002;

int main(){
    int n, m, k[MAX_N];

    scanf("%d %d", &n, &m);
    for(int i=0; i<n; i++){
        scanf("%d", &k[i]);
    }

    bool f = false;

    // brute force all possibilities
    for(int a=0; a<n; a++){
        for(int b=0; b<n; b++){
            for(int c=0; c<n; c++){
                for(int d=0; d<n; d++){
                    if(k[a] + k[b] + k[c] + k[d] == m){
                        f = true;
                    }
                }
            }
        }
    }

    if(f) printf("YES");
    else printf("NO");
}
```

The complexity for this solution is $O(n^4)$, so the estimated number of operations is 10^{12}, which will definitely get you TLE.

One way to improve it: when you have chosen the first three numbers, you also know what the last number should be in order for the sum to be m. Hence you can use binary search to fund the desired number.

Solution 2: Binary Search the last number

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;

const int MAX_N = 1002;

int n, m, k[MAX_N];

bool binary_search(int x){
    int l=0, r=n;

    while(r-l>=1){
        int i = (l+r)/2;
        if(k[i]==x) return true;
        else if(k[i]<x) l = i+1;
        else r = i;
    }

    return false;
}

void solve(){
    //must sort before BS
    sort(k, k+n);

    bool f = false;

    for(int a=0; a<n; a++){
        for(int b=0; b<n; b++){
            for(int c=0; c<n; c++){
                if(binary_search(m-k[a]-k[b]-k[c])){
                    f = true;
                }
            }
        }
    }

    if(f) printf("YES");
    else printf("NO");
}
```

The complexity is now improved to $O(n^3 \log n)$. But this is still too slow.

One way to improve: when we have chosen the first two numbers, we also know the sum of the other two numbers in order to get a total sum of m. Hence, we can binary search among all possible sums of two given numbers to find if the desired sum is present.

Solution 3: Binary Search the sum of the last two numbers

```
#include <cstdio>
#include <algorithm>
using namespace std;

const int MAX_N = 1002;

int n, m, k[MAX_N];

// sum of two numbers
int kk[MAX_N * MAX_N];

void solve(){
    for(int c=0; c<n; c++){
        for(int d=0; d<n; d++){
            kk[c*n + d] = k[c] + k[d];
        }
    }

    sort(kk, kk+n*n);

    bool f = false;
    for(int a=0; a<n; a++){
        for(int b=0; b<n; b++){
            // use STL BS
            if(binary_search(kk, kk+n*n, (m-k[a]-k[b]))){
                f = true;
            }
        }
    }

    if(f) printf("YES");
    else printf("NO");
}
```

Complexity: sorting n^2 numbers: O($n^2 \log n$) , nested loop with binary search: O($n^2 \log n$)

So overall complexity is: O($n^2 \log n$), which is acceptable.

2. Search

2.1 Recursion with Memoization

Naive recursion is often slow because it computes the same elements many times, which is a waste of time. For example, when calcuating the fibonacci number, we compute fib(8) and fib(9) to get fib(10). However, while computing fib(9), we need to compute fib(8) again.

One way to avoid this is to store all computed values in a table for future use. This technique is called memoization.

Example: Fibonacci number

```
int memo[MAX_N + 1] = {0};

int fib(int n){
    if(n<=1) return n;
    if(memo[n]!=0) return memo[n];
    return memo[n] = fib(n-1) + fib(n-2);
}
```

2.2 Stack

Stack is already implemented in STL.

To use Stack in STL:

```
#include <stack>
#include <cstdio>
using namespace std;

int main(){
    stack<int> s;
    s.push(2);
    s.push(3);
    printf("%d\n", s.top()); //3
    s.pop();
    printf("%d\n", s.top()); //2
}
```

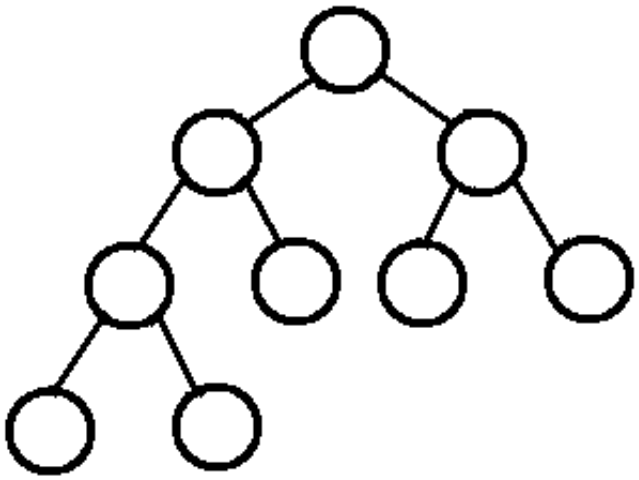
2.3 Queue

To use Queue in STL:

```cpp
#include <queue>
#include <cstdio>
using namespace std;

int main(){
    queue<int> que;
    que.push(1);
    que.push(2);
    printf("%d\n", que.front()); //1
    que.pop();
    printf("%d\n", que.front()); //2
}
```

2.4 Depth-first Search (DFS)

We use a binary tree to illustrate DFS.



DFS starts from the root and goes all the way down to the leftmost leaf node, then returns back to the previous layer, travels through the second leaf node, then returns back to the previous layer, and so on.

DFS is usually implemented by recursion.

E.g.1 Sum

You are given n integers $a_1, .., a_n$, determine if it is possible to choose some of them (each number can only be used once) so that their sum is k.

$1 \leq n \leq 20, -10^8 \leq a_i \leq 10^8, -10^8 \leq k \leq 10^8$

Since for each given number, we can choose to either take or not take, this is essentially searching through a binary tree.

```
const int MAX_N = 21;
int a[MAX_N];
int n, k;

bool dfs(int i, int sum){
    // leaf node, note n not (n-1)
    if(i==n) return sum == k;

    // not take a[i]
    if(dfs(i+1, sum)) return true;

    // take a[i]
    if(dfs(i+1, sum+a[i])) return true;

    return false;
}

void solve(){
    if(dfs(0, 0)) printf("YES");
    else printf("NO");
}
```

Total possible cases (number of leaf nodes) is 2^{n+1}, so complexity $O(2^n)$.

E.g.2 Lake Counting (POJ 2386)

Given a $N * M$ field, some areas some water after a rain ('W' : water, ' . ' : normal land). Connected areas with water (including diagonally adjacent) are counted as one puddle. Output the number of pubbles in the field.

$N, M \leq 100$

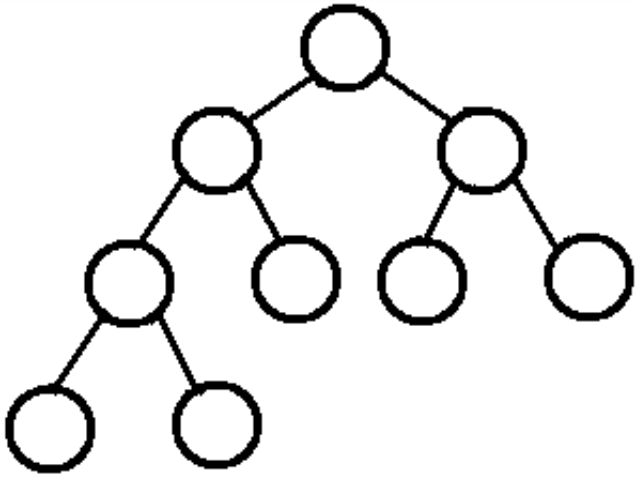
Starting from one area with water, we can use DFS to find all areas with water connected with this area. We count the number such connected puddles while setting already counted areas to normal to avoid repetition.

```
const int MAX_N = 101;
int N, M;
char field[MAX_N][MAX_N];

void dfs(int x, int y){
    // change this area to normal
    field[x][y] = '.';

    // check all 8 adjacent areas
    for(int dx=-1; dx<=1; dx++){
        for(int dy=-1; dy<=1; dy++){
            int nx = x + dx, ny = y + dy;
            if(nx>=0 && nx<N && ny>=0 && ny<M && field[nx][ny]=='W')
                dfs(nx, ny);
        }
    }
}

void solve(){
    int res = 0;
    for(int i=0; i<N; i++){
        for(int j=0; j<M; j++){
            if(field[i][j] == 'W'){
                dfs(i, j);
                res++;
            }
        }
    }

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

Since every area is only searched once (after once water is set to normal and won't be searched again), time complexity is $O(N * M)$.

2.5 Breadth-first Search (BFS)

We use a binary tree to illustrate BFS.



BFS searches from the nearest nodes to the farthest nodes. In the binary tree example, starting from the root, BFS first goes to the two nodes on the next layer which are the closest to it. Then it goes to the nodes on the third layer, and so on. BFS is usually implemented by queue.

E.g.1 Maze Runner

You are given a $N * M$ maze consisting of obstacles and normal lands. ('#' : obstacle, ' . ' : land, 'S': starting point, 'G': goal). Each step you can move left, right, up or down. Find the minimum number of steps needed from starting point to the goal.

$N, M \leq 100$

```
const int MAX_N = 101;
const int INF = 9999999;

typedef pair<int, int> P;

char maze[MAX_N][MAX_N];
int N, M;
int sx, sy;  //start pt
int gx, gy;  //goal pt

int d[MAX_N][MAX_N];

int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};

int bfs(){
    queue<P> que;

    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            d[i][j] = INF;
        }
    }

```

```
    que.push(P(sx, sy));
    d[sx][sy] = 0;

    while(que.size()){
        P cur = que.front();
        que.pop();

        if(cur.first==gx && cur.second==gy) break;

        for(int i=0; i<4; i++){
            int nx = cur.first + dx[i], ny = cur.second + dy[i];

            // available and not visited
            if(nx>=0 && nx<N && ny>=0 && ny<M && maze[nx][ny]!='#' && d[nx][ny]==INF){
                que.push(P(nx, ny));
                d[nx][ny] = d[cur.first][cur.second] + 1;
            }
        }
    }

    return d[gx][gy];
}


void solve(){
    int res = bfs();
    printf("%d\n", res);
}
```

Each point in the maze has entered the queue **at most** once. Hence complexity is $O(N * M)$.

2.6 Pruning and Backtracking

In DFS, if at a certain state we realize that this state will definitely not generate a correct answer, then we do not need to continue with this state any more, we can just seach the next possible state instead. This is called pruning.

Usually DFS is used to search for solution over a tree structure. Generally, this algorithm can be used to search over any problem space and it is called backtracking.

Example: Find all permutations of N numbers

```cpp
int total = 0;  //#permutations
const int N = 4;  //use 4 as an example
int numbers[N], used[N], res[N];

void permutate(int ith){
    if(ith==N){
        for(int i=0; i<N; i++){
            cout<<res[i];
        }
        cout<<endl;
        total++;
        return;
    }

    // find availble numbers
    for(int i=0; i<N; i++){
        if(!used[i]){
            res[ith] = nums[i];
            used[i] = 1;
            permutate(ith+1);
            //set free for future use
            used[i]=0;
        }
    }
}

int main(){
    for(int i=0; i<N; i++){
        cin>>nums[i];
    }
    memset(used, 0, sizeof(used));
    permutate(0);
    cout<<total;
}
```

For permutation, it might be easier to directly use next_permutation function. Note that you should use do while in this case, otherwise you will miss the original array case.

Example: next_permutation

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 4;
int nums[N] = {1, 2, 3, 4}, total=0;

int main(){
    do{
        total++;
        for(int i=0; i<N; i++){
            cout<<nums[i];
        }
        cout<<endl;
    }while(next_permutation(nums, nums+N));
    cout<<total;  //24
}
```

Practice (POJ)

DFS: 1979, 3009

BFS: 3669

Search: 2718, 3187, 3050

3. Greedy Algorithm