# ADS Project 4

# Red-Black Tree

---

**2022-4-14**

# Ch.1-Introduction

We've learned what Red-black Tree is. It has 5 properties:

1. Every node is either red or black.
2. The root is black.
3. All the leaves are NULL nodes and are colored black.
4. Each red node must have 2 black decends(may be NULL).
5. All simple paths from any node x to a decendant leaf have the same number of black nodes.

The question we try to solve is, how many diffrent kinds of red black tree are there when the number of nodes is given.

# Ch.2-Data Structure

A 3D array `long long dp[N][N][2]` is used, in which **N** is the total number of nodes. `dp[i][j][0/1]` stores the number of red-black tree with **i** node, whose black height equals to **j** and the root node is colored red/black perspectively.

Root node in red is against with the current definition of red-black tree but it works logically and we need that in the transformation equaltion of DP algorithm. You can take it as a process volume. Only `dp[N][j][0]` will be account for final result. $Answer = \sum_{j=0}^{N} dp[N][j][1]$.

# Ch.3-Algorithm

## 3.1-Describe the Optimal Solution

We can start with the red-black tree's unique properties. We know that every red-black tree has a black height, and 2 red-black trees having diferent black height differ even they have the same size.

Secondly, although the root of a red-black tree must be black, the root of the subtrees can be black or red, and it can affects their parents' color, and their parents' color can affect the black heights of the trees they in, so we should consider it, too.

Most importantly, the size of the red-black trees obviously affect the answer, we also should take it into the consideration.

Then we can design the optimal solution. We can set a 3-d array $dp[i][j][0/1]$ to store the number of the red-black trees that have $i$ vertices and have the black height of $j$. And when the 3rd dimension is 0, the root is red, while the 3rd dimension is 1, the root is black.

Therefore our answer to the scale N is: $Answer = \sum_{j=0}^{N} dp[N][j][1]$

## 3.2-State Transformation Equaltion

For $dp[i][j][0/1]$:

If the root of the tree is red, which is acceptable here, we can draw 3 conclusions:

1. The 2 subtrees both have a black height .

2. The 2 roots of the subtrees must be black.
3. 2 subtrees' size can be $(0, i-1), (1, i-2) \dots (i-1, 0)$.

Considering all above:

$$dp[i][j][0] = \sum_{k=0}^{i-1} dp[k][j][1] * dp[i-1-k][j][1] \tag{1}$$

If the root of the tree is black, similarly:

1. The 2 subtrees both have a black height $j-1$.
2. The 2 roots of the subtrees can be either black or red.
3. 2 subtrees' size can be $(0, i-1), (1, i-2) \dots (i-1, 0)$.

Also we can get:

$$dp[i][j][1] = \sum_{k=0}^{i-1} (dp[k][j-1][1] + dp[k][j-1][0]) * (d[i-1-k][j-1][1] + dp[i-1-k][j-1][0]) \tag{2}$$

For boundary conditions:

1. The tree is a black nulll eaf, $dp[0][0][1]$
2. The tree is a red node with 2 null leaves, $dp[1][1][0]$
3. All other boundary conditions in dp array are set to 0.

Hereby we can enumerate all numbers we need by a 3-nested loop:

```
for (int i = 2; i <= N; ++i)
//i==1 are set as boundary conditions
    for (int j = 1; j <= log_2(i+1); ++j)
    //red-black tree with black height j has at most 2^(j+1) nodes
        for (int k = 0; k<=i-1; ++k) {
            //enumerate state transformation equaltion here
        }
```

# 3.3-Accelerate by FFT

Fast Fourier transform(a.k.a. FFT) is an effective algorithm calculating curve products, or the product of polynomials.

Here we demonstrate a brief introduction of FFT. Better use Baidu or links in the appendix to better understand.

## 3.3.1 the point value expression of polynomial

For a polynomial $f(x) = \sum_{i=0}^{n} a_i x^i$, arbitrarily pick $n+1$ distinct $S = p_1, p_2, \dots, p_{n+1}$ point and calculate the value of $f(x)$, then we can call $A(x) = \{(p_1, f(p_1)), (p_2, f(p_2)), \dots, (p_{n+1}, f(p_{n+1}))\}$ the point value expression of polynomial.

Under the point value expression, the product calculation is simplified. Suppose for some $x$, $p(x) = y_1, q(x) = y_2$, then $p(x) * q(x) = y_1 y_2$.

So the time complexity of calculating two n-th order polynomials under point value expression is $O(n)$. However, only with this we can't accelerate the calculation of polynomial's product, since convert a polynomial from coefficient-expression to point value expression is $O(n^2)$ ($O(n)$ for calculating one $p(x)$). The trick is, it exists a set of $S$ making the conversion easier: $w_n^i$.

## 3.3.2 the process of FFT

The FFT contains three steps: conversion from coefficient to point value expression, calculating the product, and conversion from point value expression to coefficients. The first part is called DFT, and the third part is called IDFT.

Note that the $w_n^i$ has a useful pecularity:

$w_{2n}^{2m} = w_n^m$

$w_n^m = -w_n^{m+\frac{n}{2}}$

Let $A_0(x)$ be the even-order sum of the $A(x)$, $A_1(x)$ be the odd-order sum of the $A(x)$, namely

$A_0(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-1} x^{n/2}$

$A_1(x) = a_1 x^0 + a_3 x^1 + \ldots + a_n x^{n/2}$

$\because A(w_n^m) = a_0 w_n^0 + a_1 w_n^m + a_2 w_n^{2m} + \ldots + a_{n-1} w_n^{(n-1)m} + a_n w_n^{nm}$

$\therefore A(w_n^m) = A_0((w_n^m)^2) + w_n^m A_1((w_n^m)^2) = A_0(w_{n/2}^m) + w_n^m A_1(w_{n/2}^m)$,

$A_w n^{m+n/2} = A_0((w_n^m)^2) + w_n^{m+n/2} A_1((w_n^m)^2) = A_0(w_{n/2}^m) - w_n^m A_1(w_{n/2}^m)$

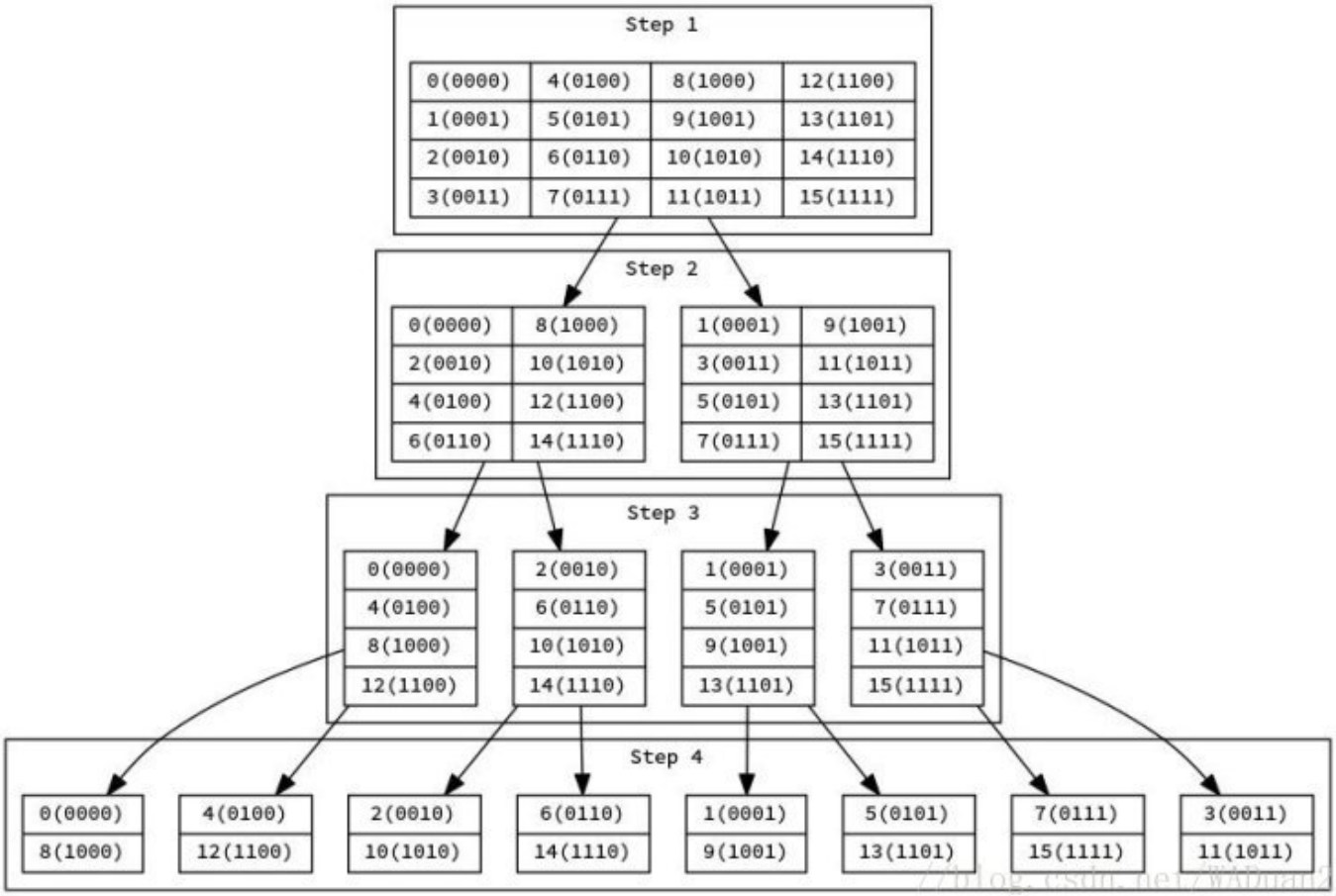
So we can calculate $A(x)$'s point value in $O(n)$ as long as we have $A_0(x)$ and $A_1(x)$.

This can be done by divide-and-conquer obviously.

The code of recursive FFT looks like this:

```
void FFT(Complex *a, int n)
{
  if(n==1) return;
  Complex *a0, *a1;
  for(i:=0; i < n; i+=2) a0[i/2] = a[i], a1[i/2] = a[i+1];
  FFT(a0, n/2), FFT(a1, n/2);
  Complex wn(cos(2pi/n), sin(2pi/n)), w(1,0);
  for(i = 0; i < n/2; ++i) a[i] = a0[i] + w*a1[i], a[i+n] = a0[i]-w*a1[i], w=w*wn;
  return;
}
```

To optimize it to iterative version, we should consider every values' original place and pre-calculate them.




Then we get the code in our final program.

### 3.3.3 calculate the modulo values

There's one thing left: what we need is the answer modulo $10^9 + 7$. Since the coefficients are about $10^9$, and $10^9 \times 10^9 \times n$ may exceeds both long long and the precision of long double, so simple FFT doesn't work.

Actually, if the modulo number is a prime looks like $2^k * p + 1$, we can use its original root replace $w_n^i$ and do FFT(also called NTT). However it's not the case for $10^9 + 7$!

So how to handle it? Our solution is to **divide the coefficient** to avoid result exceeding long long and then do FFT.

Let m be a number near $\sqrt{10^9 + 7}$ (we choose 32768).

Divide the coefficients of polynomials into $a + b$, where $a, b < m$

Then one polynomial can be divide into the sum of two polynomials.

Since $(am + b)(cm + d) = acm^2 + (ac + bd)m + cd$

we can calculate $ac, ad, bc, bd$ and do combinations to get the answer.

# Ch.4-Testing Result

# 4.1-Correctness

| N | result (mod 100000007) | N | result (mod 100000007) |
|---|---|---|---|
| 1 | 1 | 25 | 7495996 |
| 2 | 2 | 80 | 384709329 |
| 3 | 2 | 150 | 742395690 |
| 5 | 8 | 300 | 527048662 |
| 10 | 164 | 500 | 905984258 |

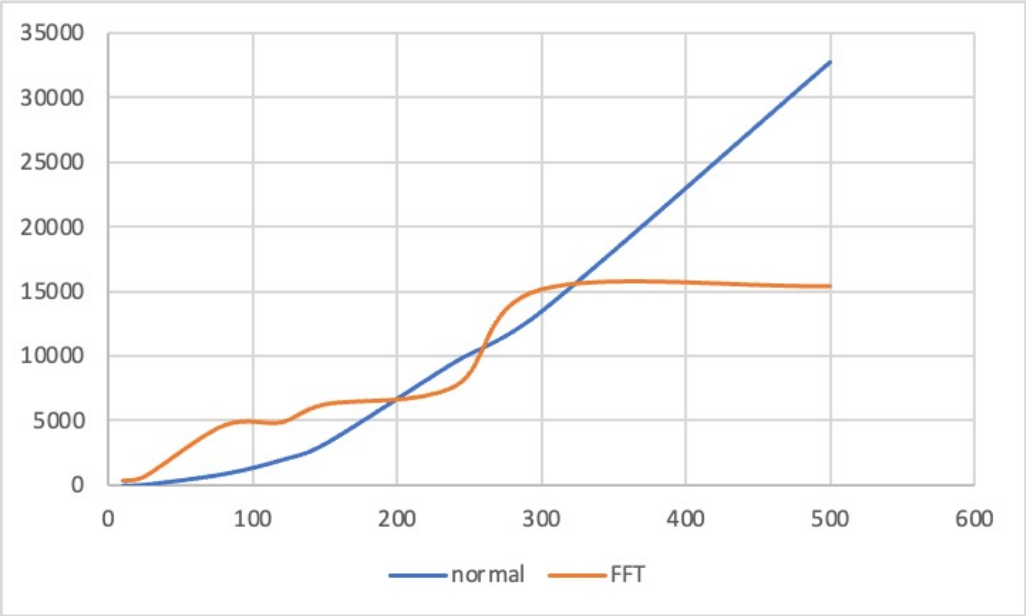
Both two algorithms have passed PTA testcase.

| 提交时间 | 状态 | 分数 | 题目 | 编译器 | 耗时 | 用户 |
|---|---|---|---|---|---|---|
| 2022/04/13 12:42:51 | 答案正确 | 35 | 编程题 | C++ (g++) | 12 ms | |

| 测试点 | 结果 | 分数 | 耗时 | 内存 |
|---|---|---|---|---|
| 0 | 答案正确 | 18 | 4 ms | 452 KB |
| 1 | 答案正确 | 7 | 4 ms | 440 KB |
| 2 | 答案正确 | 7 | 6 ms | 1344 KB |
| 3 | 答案正确 | 2 | 12 ms | 2368 KB |
| 4 | 答案正确 | 1 | 4 ms | 448 KB |

# 4.2-Efficiency

| N | Runtime of normal algorithm (us) | Runtime with FFT accelerate (ms) |
|---|---|---|
| 10 | 27 | 369 |
| 25 | 77 | 687 |
| 80 | 905 | 4622 |
| 150 | 3228 | 6245 |
| 300 | 13478 | 15114 |
| 500 | 32763 | 15361 |




It's obviously that the time complexity of normal algorithm is $O(N^2log(N))$ according to the 3-nested loop given above.

Based on the introduction of FFT in section 3.3, it takes $O(Nlog(N))$ to conversion a polynomial from coefficient to point value expression, $O(N)$ to compute product and $O(Nlog(N))$ to converse it back to coefficient form of polynomial. The time complexity is $O(Nlog(N))$.

The constant value of FFT algorithm is large so that it runs 14 times slower than the normal algorithm when N is equal to 10. It only performs better when N is around 250 or larger.

FFT requires to complete polynomial into $2^k$ terms where k is an integer before execute. It takes basically the same time when N is in range $2^k + 1$ to $2^{k+1}$. It explains why 300 and 500 take almostly the same time and its time curve seems like stairs.

# Appendix

## About FFT

A blog which explains FFT more detailingly:

- https://blog.csdn.net/oi_Konnyaku/article/details/84990404

## Declaration

We hereby declare that all the work done in this project is of our independent effort.

## Source Code

```
#include <iostream>
#include <cmath>
#include <ctime>

using namespace std;

```

```cpp
#define ll long long
#define ld double
#define For(i, x, y) for(ll i = (x);i <= (y);++i)

inline ll read() //the quick read function
{
    ll x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        if (ch == '-')f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}

inline void write(ll x)  //the quick write function
{
    if (x < 0) putchar('-'), x = -x;
    if (x > 9) write(x / 10);
    putchar(x % 10 + 48);
}


//the constant to be used
const ll N = 20505;
const ld PI = acosl(-1);
const ll mo = (ll) 1e9 + 7;

//the structure for the complex
struct Complex {
    ld r, i;

    Complex() {}

    Complex(ld _r, ld _i) : r(_r), i(_i) {}

    Complex conj() const { return Complex(r, -i); }//

    //reload the calculation for the complex
    Complex operator+(const Complex &p) const { return Complex(r + p.r, i + p.i);
}

    Complex operator-(const Complex &p) const { return Complex(r - p.r, i - p.i);
}

```

```
    Complex operator*(const Complex &p) const { return Complex(r * p.r - i * p.i,
r * p.i + i * p.r); }

    Complex operator/(const ld p) const { return Complex(r / p, i / p); }
};

inline ll add(ll p, ll q) { return (p += q) >= mo ? p - mo : p; }

inline ll sub(ll p, ll q) { return (p -= q) < 0 ? p + mo : p; }


/*class FFT
* the class for the fast fourier transformation(FFT)
* which can accelerate the convolution in the state transition in dp
*/
class FFT {
private:
    static const ll N = 131072;
    const ll P = 1e9 + 7;
    const ll base = 32768;
    Complex omega[N + 1], omegaInv[N + 1];
    Complex A[N + 1], B[N + 1];
    ll a0[N + 1], b0[N + 1], a1[N + 1], b1[N + 1];
    ll m;

    void init() {
        For(i, 0, N - 1) {
            omega[i] = Complex(cos(2 * PI / N * i), sin(2 * PI / N * i));
            omegaInv[i] = omega[i].conj();
        }
    }

    void reverse(Complex *a, ll n) {
        for (ll i = 0, j = 0; i < n; ++i) {
            if (i < j) swap(a[i], a[j]);
            for (ll l = n >> 1; (j ^= l) < l; l >>= 1) {}
        }
    }

    void transform(Complex *a, ll n, Complex *omega) {
        reverse(a, n);
        for (ll l = 2; l <= n; l <<= 1) {
            ll hl = l >> 1;
            for (Complex *x = a; x != a + n; x += l) {
                for (ll i = 0; i < hl; ++i) {
                    Complex t = omega[N / l * i] * x[i + hl];
                    x[i + hl] = x[i] - t;
                    x[i] = x[i] + t;
                }
```

```
            }
        }
    }

public:
    FFT() { init(); }

    int extend(int n) {
        int res = 1;
        while (res < n) res <<= 1;
        return m = res;
    }

    void dft(Complex *a, int n) { transform(a, n, omega); }

    void idft(Complex *a, int n) {
        transform(a, n, omegaInv);
        For(i, 0, n - 1) a[i] = a[i] / n;
    }

    void mul(ll a[], ll b[], ll c[]) {
        For(i, 0, m - 1) A[i] = Complex(a[i], b[i]);
        dft(A, m);
        For(i, 0, m - 1) {
            ll j = (m - i) & (m - 1);
            B[i] = (A[i] * A[i] - (A[j] * A[j]).conj()) * Complex(0, -0.25);
        }
        idft(B, m);
        For(i, 0, m - 1) c[i] = (ll) (B[i].r + 0.5) % P;
    }

    void mulmod(ll a[], ll b[], ll c[]) {
        ll i;
        For(i, 0, m - 1) a0[i] = a[i] >> 15, b0[i] = b[i] >> 15;
        for (mul(a0, b0, a0), i = 0; i < m; ++i) {
            c[i] = 1ll * a0[i] * base * base % P;
            a1[i] = a[i] & (base - 1), b1[i] = b[i] & (base - 1);
        }
        for (mul(a1, b1, a1), i = 0; i < m; ++i) {
            c[i] = add(a1[i], c[i]), a0[i] = add(a0[i], a1[i]);
            a1[i] = (a[i] >> 15) + (a[i] & (base - 1)), b1[i] = (b[i] >> 15) +
(b[i] & base - 1);
        }
        for (mul(a1, b1, a1), i = 0; i < m; ++i)
            c[i] = (1ll * base * sub(a1[i], a0[i]) + c[i]) % P;
    }
} fft;

ll n, m;
```

```
150
151 ll dp[N][20][2];//the 3d array for the dp
152 /*
153 * the dp[i][j][0] represents the number of red&black trees have i vetices, have
    the black height j, and its root vertex is red
154 * the dp[i][j][1] represents the number of red&black trees have i vetices, have
    the black height j, and its root vertex is black
155 */
156
157
158 ll Log[N], aa[N << 1], bb[N << 1], cc[N << 1];
159
160 int main() {
161     n = read();
162
163     //the initial condition
164     dp[0][0][1] = 1;//the null vertex has only 1 type
165     dp[1][0][0] = 1;//the single red vertex has only 1 type
166
167     clock_t start, finish;
168     start = clock();
169     For(loop, 0, 500) {
170         For(i, 2, n + 3) Log[i] = Log[i >> 1] + 1;
171         //to pre-calculate a table of log, so we needn't calc it in the loop and
    can decrease the time complexity
172
173         For(j, 1, Log[n + 1])
174             //our loop for dp, j is the black height, as a red&black tree with
    size n have the maximum black height of log(n+1)
175             //we only enumerate it to log(n+1)
176         {
177             For(i, 0, n - 1) aa[i] = bb[i] = add(dp[i][j - 1][0], dp[i][j - 1]
    [1]);
178             //the FFT accelerate part:
179             For(i, n, n * 2 - 1) aa[i] = bb[i] = 0;
180             m = fft.extend(n << 1);
181             fft.mulmod(aa, bb, cc);
182
183             For(i, 1, n) dp[i][j][1] = cc[i - 1];//,cerr<<cc[i]<<' ';cerr<<endl;
184             For(i, 0, n - 1) aa[i] = bb[i] = dp[i][j][1];
185             For(i, n, n * 2 - 1) aa[i] = bb[i] = 0;
186             m = fft.extend(n << 1);
187             fft.mulmod(aa, bb, cc);
188             For(i, 1, n) dp[i][j][0] = cc[i - 1];
189         }
190     }
191     finish = clock();
192     ll ans = 0;
193     For(i, 0, n) ans = add(ans, dp[n][i][1]);//,cerr<<dp[n][i][1]<<endl;
```

```
    cout << ans % 1000000007 << endl;
    cout << finish - start << endl;
}
```

```
#pragma GCC optimize("Ofast")

#include <iostream>
#include <ctime>

using namespace std;
#define ll long long
#define For(i, x, y) for(ll i = (x);i <= (y);++i)

inline ll read() {
    ll x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        if (ch == '-')f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}

inline void write(ll x) {
    if (x < 0) putchar('-'), x = -x;
    if (x > 9) write(x / 10);
    putchar(x % 10 + 48);
}

const ll mo = (ll) 1000000007;
const ll N = 505;

inline ll add(ll p, ll q) {
    return (p += q) >= mo ? p - mo : p;
}

ll dp[N][N][2], Log[N];

int main() {
    ll n = read();
    For(i, 2, n + 3) Log[i] = Log[i >> 1] + 1;
    dp[0][0][1] = 1;
    dp[1][0][0] = 1;
    dp[1][1][1] = 1;

```

```
    clock_t start, finish;
    start = clock();
    For(loop, 0, 500)
        For(i, 2, n)
            For(j, 1, Log[i + 3]) {
                For(k, 0, i - 1) {
                    dp[i][j][1] = add(dp[i][j][1], 1ll * (dp[k][j - 1][0] + dp[k][j - 1][1]) *
                                                       (dp[i - k - 1][j - 1][0] + dp[i - k - 1][j - 1][1]) % mo);
                    dp[i][j][0] = add(dp[i][j][0], 1ll * dp[k][j][1] * dp[i - k - 1][j][1] % mo);
                }
            }
    finish = clock();
    ll ans = 0;
    For(i, 0, n) ans = add(ans, dp[n][i][1]);//,cerr<<dp[n][i][1]<<endl;
    cout << ans % 1000000007 << endl;
    cout << finish - start << endl;
}
```