

Contents

1	Template	2
2	Search	2
2.1	Ternary Search	2
3	Algebra	3
3.1	All divisors	3
3.2	Primality test	3
3.3	Binary exponentiation	3
3.4	Greatest common divisor	4
3.4.1	Least common multiple	4
3.4.2	Extended Euclides Algorithm	4
3.5	Linear Diophantine Equations	4
3.5.1	Any solution	4
3.6	Integer Factorization	5
3.6.1	Pollard's Rho	5
3.7	Fast Fourier Transform	6
3.7.1	Polynomial Multiplication	7
4	Graphs	8
4.1	DFS	8
4.2	BFS	8
4.2.1	Shortest path on unweighted graph	8
4.3	Flood Fill	9
4.4	Topological Sort (Directed Acyclic Graph)	10
4.4.1	DFS Variation	10
4.4.2	Kahn's Algorithm	10
4.5	Bipartite Graph Check (Undirected Graph)	11
4.6	Cycle Check (Directed Graph)	11
4.7	Dijkstra	12
5	Math Formulas	13
5.1	Sum of an arithmetic progression	13
5.2	Permutation with repeated elements	13
5.3	Check if is geometric progression	13
5.4	Bitwise equations	13
5.5	Cube of Binomial	13
5.5.1	Sum of Cubes	13
5.5.2	Difference of Cubes	14
5.6	Binomial expansion	14

6	Facts	14
6.1	XOR	14
6.1.1	Self-inverse property	14
6.1.2	Identity element	14
6.1.3	Commutative	14
6.1.4	Associative	14

1 Template

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 using ll =          long long;
5 #define vll         vector<ll>
6 #define vvll        vector<vll>
7 #define pll         pair<ll, ll>
8 #define vp11        vector<pll>
9 #define vvpll       vector<vp11>
10 #define endl '\n'
11 #define all(xs)      xs.begin(), xs.end()
12 #define found(x, xs) (xs.find(x) != xs.end())

```

2 Search

2.1 Ternary Search

$O(\log n)$

Function $f(x)$ is unimodal on an interval $[l, r]$. Unimodal means: the function strictly increases first, reaches a maximum, and then strictly decreases OR the function strictly decreases first, reaches a minimum and then strictly increases

```

1 double ternary_search(double l, double r) {
2     double eps = 1e-9; // error limit
3     while(r - l > eps) {
4         double m1 = l + (r-l) / 3;
5         double m2 = r - (r-l) / 3;
6
7         double f1 = f(m1);
8         double f2 = f(m2);
9
10        if(f1 < f2)
11            l = m1;
12        else
13            r = m2;
14    }
15 }

```

```

16     return f(1);
17 }

```

3 Sequences

3.1 Max/Min subsegment

$O(n)$

```

1 ll kadane(const vll &a) {
2     ll n = a.size();
3     ll ans = a[0], ans_l = 0, ans_r = 0;
4     ll sum = 0, minus_pos = -1;
5
6     for (ll r = 0; r < n; ++r) {
7         sum += a[r];
8         if (sum > ans) {
9             ans = sum;
10            ans_l = minus_pos + 1;
11            ans_r = r;
12        }
13        if (sum < 0) {
14            sum = 0;
15            minus_pos = r;
16        }
17    }
18
19    return ans;
20 }
21

```

3.1.1 Max/Min submatrix

$O(nm^2)$

```

1 ll MSR(ll n, ll m, const vvll &a) {
2     ll ans = INFINITY;
3
4     for (ll i=0; i<m; i++) {
5         vll r(n+1, 0);
6
7         for (ll j=i; j<m; j++) {
8             for (ll k=0; k<n; k++)
9                 r[k] += a[k][j];
10
11            ans = max(ans, kadane(n, r));
12        }
13    }
14 }

```

```

14
15     return ans;
16 }

```

4 Algebra

4.1 All divisors

$O(\sqrt{n})$

```

1  vll divisors(ll n) {
2      vll divs;
3      for (ll i = 1; 1LL * i * i <= n; i++) {
4          if (n % i == 0) {
5              divs.push_back(i);
6              if (i != n / i) {
7                  divs.push_back(n / i);
8              }
9          }
10     }
11
12     return divs;
13 }

```

4.2 Primality test

$O(\sqrt{n})$

```

1  bool isPrime(ll n)
2  {
3      if (n != 2 && n % 2 == 0)
4          return false;
5
6      for (ll d = 3; d * d <= n; d += 2)
7      {
8          if (n % d == 0)
9              return false;
10     }
11
12     return n >= 2;
13 }

```

4.3 Binary exponentiation

$O(\log n)$

```

1 ll binpow(ll a, ll b) {
2     ll res = 1;
3     while (b > 0) {
4         if (b & 1)
5             res = res * a;
6         a = a * a;
7         b >>= 1;
8     }
9     return res;
10 }

```

4.4 Greatest common divisor

$O(\log \min(a, b))$

```

1 ll gcd (ll a, ll b) {
2     while (b) {
3         a %= b;
4         swap(a, b);
5     }
6     return a;
7 }

```

4.4.1 Least common multiple

```

1 ll lcm(ll a, ll b) {
2     return a / gcd(a, b) * b;
3 }

```

4.4.2 Extended Euclides Algorithm

```

1 ll gcd(ll a, ll b, ll& x, ll& y) {
2     if (b == 0) {
3         x = 1;
4         y = 0;
5         return a;
6     }
7     ll x1, y1;
8     ll d = gcd(b, a % b, x1, y1);
9     x = y1;
10    y = x1 - y1 * (a / b);
11    return d;
12 }

```

4.5 Linear Diophantine Equations

$O(\log \min(a, b))$

4.5.1 Any solution

```
1 bool find_any_solution(ll a, ll b, ll c, ll &x0, ll &y0, ll
   &g) {
2     g = gcd(abs(a), abs(b), x0, y0);
3     if (c % g) {
4         return false;
5     }
6
7     x0 *= c / g;
8     y0 *= c / g;
9     if (a < 0) x0 = -x0;
10    if (b < 0) y0 = -y0;
11    return true;
12 }
```

4.6 Integer Factorization

4.6.1 Pollard's Rho

$O(\sqrt[3]{n} \log n)$

```
1 /**
2  * @param a first multiplier
3  * @param b second multiplier
4  * @param mod
5  * @return a * b mod n (without overflow)
6  * @brief Multiplies two numbers >= 10^18
7  * Time Complexity: O(log b)
8  */
9 ll mult(ll a, ll b, ll mod) {
10     ll result = 0;
11     while (b) {
12         if (b & 1)
13             result = (result + a) % mod;
14         a = (a + a) % mod;
15         b >>= 1;
16     }
17     return result;
18 }
19
20 /**
21  * @param x first multiplier
22  * @param c second multiplier
23  * @param mod
```

```

24  * @return f(x) = x^2 + c mod (mod)
25  * @brief Polynomial function chosen for pollard's rho
26  * Time Complexity: O(1)
27  */
28  ll f(ll x, ll c, ll mod) {
29      return (mult(x, x, mod) + c) % mod;
30  }
31
32  /**
33   * @param n number that we want to find a factor p
34   * @param x0 number where we will start
35   * @param c constant in polynomial function
36   * @return fac
37   * @brief Pollard's Rho algorithm (works only for composite
38   *         numbers)
39   * if(g==n) try other starting values
40   * Time Complexity: O(n^(1/4) log n)
41  */
42  ll rho(ll n, ll x0=2, ll c=1) {
43      ll x = x0;
44      ll y = x0;
45      ll g = 1;
46      while (g == 1) {
47          x = f(x, c, n);
48          y = f(y, c, n);
49          y = f(y, c, n);
50          g = gcd(abs(x - y), n);
51      }
52      return g;
53  }

```

4.7 Fast Fourier Transform

$O(n \log n)$

```

1  using cd = complex<double>;
2  const double PI = acos(-1);
3
4  /**
5   * @param a vector that we want to transform
6   * @param invert inverse fft or not
7   * @brief apply fft or inverse fft to a vector
8   * Time Complexity: O(n log n)
9   */
10 void fft(vector<cd> &a, bool invert) {
11     ll n = a.size();
12     if (n == 1)
13         return;
14

```

```

15     vector<cd> a0(n / 2), a1(n / 2);
16     for (ll i = 0; 2 * i < n; i++) {
17         a0[i] = a[2*i];
18         a1[i] = a[2*i+1];
19     }
20     fft(a0, invert);
21     fft(a1, invert);
22
23     double ang = 2 * PI / n * (invert ? -1 : 1);
24     cd w(1), wn(cos(ang), sin(ang));
25     for (ll i = 0; 2 * i < n; i++) {
26         a[i] = a0[i] + w * a1[i];
27         a[i + n/2] = a0[i] - w * a1[i];
28         if (invert) {
29             a[i] /= 2;
30             a[i + n/2] /= 2;
31         }
32         w *= wn;
33     }
34 }

```

4.7.1 Polynomial Multiplication

```

1  /**
2   * @param a first polynomial coefficients
3   * @param b second polynomial coefficients
4   * @return product of two polynomials
5   * @brief Multiplies two polynomials
6   * Time Complexity:  $O(n \log n)$ 
7   */
8  vll multiply(vll const& a, vll const& b) {
9      vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end()
10         );
11      ll n = 1;
12      while (n < a.size() + b.size())
13          n <<= 1;
14      fa.resize(n);
15      fb.resize(n);
16
17      fft(fa, false);
18      fft(fb, false);
19      for (ll i = 0; i < n; i++)
20          fa[i] *= fb[i];
21      fft(fa, true);
22
23      vll result(n, 0);
24      for (ll i = 0; i < n; i++) {
25          result[i] += round(fa[i].real());
26      }
27  }

```



```

25         if(result[i] >= 10) {
26             result[i+1] += result[i] / 10;
27             result[i] %= 10;
28         }
29     }
30     return result;
31 }

```

5 Graphs

5.1 DFS

$O(n + m)$

```

1 void dfs(ll at, ll n ,vll adj[], bool visited[]) {
2     if(visited[at])
3         return;
4
5     visited[at] = true;
6
7     vll neighbours = adj[at];
8     for(auto nex: neighbours)
9         dfs(nex.first, n, adj, visited);
10 }

```

5.2 BFS

$O(n + m)$

```

1 void bfs(ll s, ll n, vll adj[]) {
2     bool visited[n] = {0};
3     visited[s] = true;
4
5     queue<ll> q;
6     q.push(s);
7     while (!q.empty())
8     {
9         vll neighbours = adj[q.front()];
10        for(auto nex: neighbours) {
11            if(!visited[nex]) {
12                visited[nex]=true;
13                q.push(nex);
14            }
15        }
16        cout << q.front() << '\n';
17        q.pop();
18    }
19 }

```

5.2.1 Shortest path on unweighted graph

$O(n + m)$

```
1  vll solve(ll s, ll n, vll adj[]) {
2      bool visited[n] = {0};
3      visited[s] = true;
4
5      queue<ll> q;
6      q.push(s);
7      vll prev(n, -1);
8      while (!q.empty())
9      {
10         vll neighbours = adj[q.front()];
11         for(auto nex: neighbours) {
12             if(!visited[nex]) {
13                 visited[nex]=true;
14                 q.push(nex);
15                 prev[nex] = q.front();
16             }
17         }
18         q.pop();
19     }
20
21     return prev;
22 }
23
24 vll reconstructPath(ll s, ll e, vll prev) {
25     vll path;
26     for(ll i=e; i!=-1; i=prev[i])
27         path.push_back(i);
28
29     reverse(path.begin(), path.end());
30
31     if(path[0]==s)
32         return path;
33     else {
34         vll place;
35         return place;
36     }
37 }
38
39 vll bfs(ll s, ll e, ll n, vll adj[]) {
40     vll prev = solve(s, n, adj);
41
42     return reconstructPath(s, e, prev);
43 }
```

5.3 Flood Fill

$O(n + m)$

```
1 int dir_y[] = {};  
2 int dir_x[] = {};  
3  
4 int ff(int i, int j, char c1, char c2) {  
5     if ((i < 0) || (i >= n)) return 0;  
6     if ((j < 0) || (j >= m)) return 0;  
7     if (grid[i][j] != c1) return 0;  
8  
9     int ans = 1;  
10    grid[i][j] = c2;  
11  
12    for (int d = 0; d < 8; ++d)  
13        ans += floodfill(i+dir_y[d], j+dir_x[d], c1, c2);  
14  
15    return ans;  
16 }  
  
;
```

5.4 Topological Sort (Directed Acyclic Graph)

5.4.1 DFS Variation

$O(n + m)$

```
1 void dfs(ll at, ll n, vpll adj[], bool visited[], vll &ts) {  
2     if(visited[at])  
3         return;  
4  
5     visited[at] = true;  
6  
7     vpll neighbours = adj[at];  
8     for(auto nex: neighbours)  
9         dfs(nex.first, n, adj, visited);  
10    ts.push_back(at);  
11 }  
  
// Only change
```

5.4.2 Kahn's Algorithm

```
1 priority_queue<ll, vll, greater<ll>> pq;  
2 for(ll at=0; at<n; at++)  
3     if(in_degree[at] == 0)  
4         pq.push(at);  
5  
6 while(!pq.empty()) {  
7     // Push all sources of  
8     // connected components in graph  
9     ll u = pq.top(); pq.pop();  
10    for(auto v: adj[u])  
11        in_degree[v]--;  
12    if(in_degree[v] == 0)  
13        pq.push(v);  
14 }  
15
```

```

7      ll at = pq.top(); pq.pop();
8      vll neighbors = adj[at];
9      for(auto nex: neighbors) {
10         in_degree[nex]--;
11         if(in_degree[nex]>0) continue;
12         pq.push(nex);
13     }
14 }

```

5.5 Bipartite Graph Check (Undirected Graph)

$O(n + m)$

```

1 bool isBipartite(ll s, ll n, vll adj[]) {
2     queue<ll> q;
3     q.push(s);
4     vll color(n, -1); color[s]=0;
5     bool flag = true;
6     while (!q.empty())
7     {
8         vll neighbours = adj[q.front()];
9         for(auto nex: neighbours) {
10             if(color[nex] == -1) {
11                 color[nex] = 1-(color[q.front()]);
12                 q.push(nex);
13             }
14             else if(color[nex] == color[q.front()]) {
15                 flag = false;
16                 break;
17             }
18         }
19         q.pop();
20     }
21
22     return flag;
23 }

```

5.6 Cycle Check (Directed Graph)

$O(n + m)$

```

1 enum { UNVISITED = -1, VISITED = -2, EXPLORED=-3};
2
3 void cycleCheck(ll at, ll n ,vll adj[], int visited[], ll
4     dfs_parent[]) {
5     visited[at] = EXPLORED;
6
7     vll neighbours = adj[at];

```

```

7   for(auto nex: neighbours) {
8       if(visited[nex] == UNVISITED) {
9           // Tree edges (part of the DFS spanning tree)
10          dfs_parent[nex] = at;
11          cycleCheck(nex, n, adj, visited);
12      }
13      else if(visited[nex] == EXPLORED) {
14          if(nex == dfs_parent[at]) {
15              // Trivial cycle
16              // Do something
17          }
18          else {
19              // Non trivial cycle - Back Edge ((u, v)
20              // such that v is the ancestor of node u but
21              // is not part of the DFS tree)
22              // Do something
23          }
24      }
25      else if(visited[nex] == VISITED) {
26          // Forward/Cross edge ((u, v) such that v is a
27          // descendant but not part of the DFS tree)
28          // Do something
29      }
30  }
31  visited[at] = VISITED;
32  }

```

5.7 Dijkstra

$O(n \log n + m \log n)$

```

1  void dijkstra(ll s, vll & d, vll & p) {
2      d.assign(n, LLONG_MAX);
3      p.assign(n, -1);
4
5      d[s] = 0;
6      priority_queue<pll, vpll, greater<pll>> q;
7      q.push({0, s});
8      while (!q.empty()) {
9          ll v = q.top().second;
10         ll d_v = q.top().first;
11         q.pop();
12         if (d_v != d[v])
13             continue;
14
15         for (auto edge : adj[v]) {

```

```

16         ll to = edge.first;
17         ll len = edge.second;
18
19         if (d[v] + len < d[to]) {
20             d[to] = d[v] + len;
21             p[to] = v;
22             q.push({d[to], to});
23         }
24     }
25 }
26

```

6 Math Formulas

6.1 Sum of an arithmetic progression

$$S_n = \frac{n}{2}(a_1 + a_n)$$

6.2 Permutation with repeated elements

$$P_n = \frac{n!}{n_1!n_2!\dots n_k!}$$

6.3 Check if is geometric progression

$$a_i^2 = a_{i-1}a_{i+1}$$

6.4 Bitwise equations

$$a|b = a \oplus b + a \& b$$

$$a \oplus (a \& b) = (a|b) \oplus b$$

$$(a \& b) \oplus (a|b) = a \oplus b$$

$$a + b = a|b + a \& b$$

$$a + b = a \oplus b + 2(a \& b)$$

$$a - b = (a \oplus (a \& b)) - ((a|b) \oplus a)$$

$$a - b = ((a|b) \oplus b) - ((a|b) \oplus a)$$

$$a - b = (a \oplus (a \& b)) - (b \oplus (a \& b))$$

$$a - b = ((a|b) \oplus b) - (b \oplus (a \& b))$$

6.5 Cube of Binomial

$$(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

$$(a - b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$$

6.5.1 Sum of Cubes

$$a^3 + b^3 = (a + b)(a^2 - ab + b^2)$$

6.5.2 Difference of Cubes

$$a^3 - b^3 = (a - b)(a^2 + ab + b^2)$$

6.6 Binomial expansion

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$
$$(a + b)^n = \sum_{k=0}^n \binom{n}{k} a^k b^{n-k}$$

7 Facts

7.1 XOR

7.1.1 Self-inverse property

To cancel a XOR, you can XOR again the same value because $a \oplus a = 0$, so $(value \oplus a) \oplus a = value$

7.1.2 Identity element

$$a \oplus 0 = a$$

7.1.3 Commutative

$$a \oplus b = b \oplus a$$

7.1.4 Associative

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$