BASIC TEMPLATE & FAST I/O

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define pb push_back
#define mp make_pair
#define all(x) (x).begin(), (x).end()
#define F first
#define S second
#define sz(x) (int)(x).size()

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    return 0;
}
```

STL CONTAINERS

1. **VECTOR** (Dynamic Array)

```cpp
vector<int> v;              // Declaration
vector<int> v(n);           // Size n, all 0
vector<int> v(n, 5);        // Size n, all 5
vector<vector<int>> v2d;    // 2D vector
```

Key Functions:

- `v.push_back(x)` - Add element at end | O(1)
- `v.pop_back()` - Remove last element | O(1)
- `v.size()` - Return size | O(1)
- `v.empty()` - Check if empty | O(1)
- `v.clear()` - Remove all elements | O(n)
- `v[i]` or `v.at(i)` - Access element | O(1)
- `v.front()` - First element | O(1)
- `v.back()` - Last element | O(1)
- `v.insert(v.begin()+i, x)` - Insert at position | O(n)
- `v.erase(v.begin()+i)` - Delete at position | O(n)
- `v.resize(n)` - Change size | O(n)

2. **SET** (Ordered, Unique Elements)

```
set<int> s;                 // Sorted, no duplicates
multiset<int> ms;           // Sorted, allows duplicates
unordered_set<int> us;      // No order, no duplicates, O(1) avg
```

Key Functions:

- `s.insert(x)` - Add element | O(log n)
- `s.erase(x)` - Remove element | O(log n)
- `s.find(x)` - Find element (returns iterator) | O(log n)
- `s.count(x)` - Check if exists (0 or 1) | O(log n)
- `s.size()` - Return size | O(1)
- `s.clear()` - Remove all | O(n)
- `s.lower_bound(x)` - First element ≥ x | O(log n)
- `s.upper_bound(x)` - First element > x | O(log n)
- `*s.begin()` - Minimum element | O(1)
- `*s.rbegin()` - Maximum element | O(1)

When to use:

- Need sorted unique elements
- Fast insertion/deletion with ordering
- Range queries (lower_bound/upper_bound)

3. **MAP** (Key-Value Pairs)

```
map<int, int> m;            // Sorted by key
unordered_map<int, int> um; // No order, O(1) avg access
```

Key Functions:

- `m[key] = value` - Insert/update | O(log n)
- `m.insert({key, value})` - Insert pair | O(log n)
- `m.erase(key)` - Remove key | O(log n)
- `m.find(key)` - Find key (returns iterator) | O(log n)
- `m.count(key)` - Check if key exists | O(log n)
- `m.size()` - Return size | O(1)
- `m.clear()` - Remove all | O(n)

Iteration:

```
for(auto it : m) {
    cout << it.first << " " << it.second << "\n";
}
```

When to use:

- Frequency counting
- Mapping relationships
- Coordinate compression

4. **QUEUE** (FIFO)

```
queue<int> q;
```

Key Functions:

- `q.push(x)` - Add to back | O(1)
- `q.pop()` - Remove from front | O(1)
- `q.front()` - Access front element | O(1)
- `q.back()` - Access back element | O(1)
- `q.size()` - Return size | O(1)
- `q.empty()` - Check if empty | O(1)

When to use: BFS, level-order traversal

5. **STACK** (LIFO)

```
stack<int> st;
```

Key Functions:

- `st.push(x)` - Add to top | O(1)
- `st.pop()` - Remove from top | O(1)
- `st.top()` - Access top element | O(1)
- `st.size()` - Return size | O(1)
- `st.empty()` - Check if empty | O(1)

When to use: DFS, parentheses matching, expression evaluation

6. **DEQUE** (Double-ended Queue)

```
deque<int> dq;
```

Key Functions:

- `dq.push_back(x)` - Add to back | O(1)
- `dq.push_front(x)` - Add to front | O(1)
- `dq.pop_back()` - Remove from back | O(1)
- `dq.pop_front()` - Remove from front | O(1)
- `dq.front()` - Access front | O(1)
- `dq.back()` - Access back | O(1)

- `dq[i]` - Random access | O(1)

When to use: Sliding window problems

7. **PRIORITY QUEUE** (Heap)

```
priority_queue<int> pq;              // Max heap
priority_queue<int, vector<int>, greater<int>> pq; // Min heap
```

Key Functions:

- `pq.push(x)` - Insert element | O(log n)
- `pq.pop()` - Remove top | O(log n)
- `pq.top()` - Access top (max/min) | O(1)
- `pq.size()` - Return size | O(1)
- `pq.empty()` - Check if empty | O(1)

When to use: Dijkstra, greedy problems, k-th largest

8. **PAIR**

```
pair<int, int> p = {1, 2};
pair<int, int> p = make_pair(1, 2);
```

Access: `p.first`, `p.second` **Auto sorting:** First by first element, then by second

---

🔧 IMPORTANT ALGORITHMS

1. **SORTING**

```
sort(v.begin(), v.end());              // Ascending
sort(v.begin(), v.end(), greater<int>()); // Descending
sort(arr, arr+n);                      // Array sorting

// Custom comparator
bool cmp(int a, int b) { return a > b; }
sort(v.begin(), v.end(), cmp);


// Sort pairs by second element
sort(v.begin(), v.end(), [](pair<int,int> a, pair<int,int> b) {
    return a.second < b.second;
});
```

Time: O(n log n)

2. **BINARY SEARCH**

```
// On sorted array/vector
bool found = binary_search(v.begin(), v.end(), x); // Returns true/false

// Lower bound: first element >= x
auto it = lower_bound(v.begin(), v.end(), x);
int pos = it - v.begin();

// Upper bound: first element > x
auto it = upper_bound(v.begin(), v.end(), x);

// Count occurrences
int cnt = upper_bound(all(v), x) - lower_bound(all(v), x);

// Manual binary search
int l = 0, r = n-1, ans = -1;
while(l <= r) {
    int mid = l + (r-l)/2;
    if(check(mid)) {
        ans = mid;
        r = mid - 1;  // or l = mid + 1
    } else {
        l = mid + 1;  // or r = mid - 1
    }
}
```

Time: O(log n)

3. **REVERSE**

```
reverse(v.begin(), v.end());
reverse(s.begin(), s.end());  // String
```

4. **MIN/MAX ELEMENT**

```
int minVal = *min_element(v.begin(), v.end());
int maxVal = *max_element(v.begin(), v.end());


// Index of min/max
int minIdx = min_element(v.begin(), v.end()) - v.begin();
```

5. **ACCUMULATE (Sum)**

```
int sum = accumulate(v.begin(), v.end(), 0);
ll sum = accumulate(v.begin(), v.end(), 0LL); // For long long
```

6. **UNIQUE** (Remove Duplicates)

```
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

7. **NEXT/PREV PERMUTATION**

```
vector<int> v = {1, 2, 3};
do {
    // Process permutation
} while(next_permutation(v.begin(), v.end()));
```

8. **COUNT**

```
int cnt = count(v.begin(), v.end(), x); // Count occurrences of x
```

9. **FILL**

```
fill(arr, arr+n, 0);           // Fill array
fill(v.begin(), v.end(), -1);  // Fill vector
```

10. **SWAP**

```
swap(a, b);
swap(v[i], v[j]);
```

NUMBER THEORY

1. **GCD & LCM**

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}


int lcm(int a, int b) {
    return (a / gcd(a, b)) * b;  // Avoid overflow
}


// Built-in (C++17)
int g = __gcd(a, b);
```

2. **PRIME CHECK**

```
bool isPrime(int n) {
    if(n <= 1) return false;
    if(n <= 3) return true;
    if(n % 2 == 0 || n % 3 == 0) return false;
    for(int i = 5; i * i <= n; i += 6) {
        if(n % i == 0 || n % (i + 2) == 0)
            return false;
    }
    return true;
}
```

Time: O(√n)

3. **SIEVE OF ERATOSTHENES** (All primes up to n)

```
vector<bool> sieve(int n) {
    vector<bool> prime(n+1, true);
    prime[0] = prime[1] = false;
    for(int i = 2; i * i <= n; i++) {
        if(prime[i]) {
            for(int j = i*i; j <= n; j += i)
                prime[j] = false;
        }
    }
    return prime;
}
```

Time: O(n log n)

4. **PRIME FACTORIZATION**

```
vector<int> primeFactors(int n) {
    vector<int> factors;
    for(int i = 2; i * i <= n; i++) {
        while(n % i == 0) {
            factors.push_back(i);
            n /= i;
        }
    }
    if(n > 1) factors.push_back(n);
    return factors;
}
```

5. **COUNT DIVISORS**

```
int countDivisors(int n) {
    int cnt = 0;
    for(int i = 1; i * i <= n; i++) {
        if(n % i == 0) {
            cnt++;
            if(i != n/i) cnt++;
        }
    }
    return cnt;
}
```

6. **MODULAR ARITHMETIC**

```
const int MOD = 1e9 + 7;

// Modular addition
int add(int a, int b) { return (a + b) % MOD; }

// Modular multiplication
int mul(int a, int b) { return (1LL * a * b) % MOD; }

// Modular power
int power(int a, int b) {
    int res = 1;
    while(b > 0) {
        if(b & 1) res = mul(res, a);
        a = mul(a, a);
        b >>= 1;
    }
    return res;
}
```

```
// Modular inverse (when MOD is prime)
int modInv(int a) { return power(a, MOD - 2); }

// Modular division
int divide(int a, int b) { return mul(a, modInv(b)); }
```

7. **FACTORIAL**

```
ll factorial(ll n) {
    ll ans = 1;
    for (ll i = 1; i <= n; i++) {
        ans = (ans * i) % mod;
    }
    return ans;
}

int main() {
    ll n;
    cin >> n;
    cout << factorial(n) << "\n";
}
```

8. **FACTORIAL & nCr**

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
const int MAXN = 1e6 + 5;
const ll mod = 1e9 + 7;

ll fact[MAXN], invFact[MAXN];

// (a * b) % mod
ll mul(ll a, ll b) {
    return (a % mod) * (b % mod) % mod;
}

// Binary exponentiation
ll binpow(ll a, ll p) {
    ll r = 1;
    while (p) {
        if (p & 1) r = mul(r, a);
        a = mul(a, a);
        p >>= 1;
    }
    return r;
}
```

```
// Modular inverse using Fermat's little theorem (mod must be prime)
ll modInv(ll x) {
    return binpow(x, mod - 2);
}

void precompute() {
    fact[0] = 1;

    for (int i = 1; i < MAXN; i++)
        fact[i] = mul(fact[i - 1], i);

    invFact[MAXN - 1] = modInv(fact[MAXN - 1]);

    for (int i = MAXN - 2; i >= 0; i--)
        invFact[i] = mul(invFact[i + 1], i + 1);
}

ll nCr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return mul(fact[n], mul(invFact[r], invFact[n - r]));
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    precompute();

    // Example usage:
    cout << nCr(5, 2) << "\n";  // Output = 10

    return 0;
}
```

PROBLEM SOLVING TECHNIQUES

1. **PREFIX SUM**

```
// 1D Prefix Sum
vector<int> prefix(n+1, 0);

for(int i = 1; i <= n; i++) {
    prefix[i] = prefix[i-1] + arr[i-1];
}
// Sum of [l, r] = prefix[r+1] - prefix[l]

// 2D Prefix Sum
prefix[i][j] = arr[i-1][j-1] + prefix[i-1][j] + prefix[i][j-1] - prefix[i-1][j-1];
```

```
// Sum of rectangle = prefix[r2][c2] - prefix[r1-1][c2] - prefix[r2][c1-1] +
prefix[r1-1][c1-1]
```

2. **TWO POINTERS**

```
// Example: Find pair with sum = target
int l = 0, r = n-1;
while(l < r) {
    int sum = arr[l] + arr[r];
    if(sum == target) {
        // Found
        break;
    } else if(sum < target) {
        l++;
    } else {
        r--;
    }
}
```

3. **SLIDING WINDOW**

```
// Maximum sum subarray of size k
int maxSum = 0, windowSum = 0;
for(int i = 0; i < k; i++) windowSum += arr[i];
maxSum = windowSum;

for(int i = k; i < n; i++) {
    windowSum += arr[i] - arr[i-k];
    maxSum = max(maxSum, windowSum);
}
```

4. **FREQUENCY COUNTING**

```
map<int, int> freq;
for(int x : v) freq[x]++;

// Or using array (when values are small)
int freq[100005] = {0};
for(int x : v) freq[x]++;
```

5. **DIFFERENCE ARRAY** (Range Updates)

```
// Update [l, r] by x
diff[l] += x;
diff[r+1] -= x;

// Build actual array
for(int i = 1; i < n; i++)
    diff[i] += diff[i-1];
```

6. **KADANE'S ALGORITHM** (Max Subarray Sum)

```
int maxSum = arr[0], currSum = arr[0];
for(int i = 1; i < n; i++) {
    currSum = max(arr[i], currSum + arr[i]);
    maxSum = max(maxSum, currSum);
}
```

7. **BINARY SEARCH ON ANSWER**

```
// Find minimum x such that check(x) is true
int l = 0, r = 1e9, ans = -1;
while(l <= r) {
    int mid = l + (r-l)/2;
    if(check(mid)) {
        ans = mid;
        r = mid - 1;  // Try smaller
    } else {
        l = mid + 1;
    }
}
```

8. **GREEDY** (Common Patterns)

- Sort and process
- Always pick min/max
- Exchange argument
- Optimal substructure

MATH FORMULAS & TRICKS

1. **ARITHMETIC PROGRESSION**

```
Sum = n/2 * (2a + (n-1)d)
Sum = n/2 * (first + last)
nth term = a + (n-1)d
```

2. **GEOMETRIC PROGRESSION**

```
Sum = a * (r^n - 1) / (r - 1)
Infinite sum (|r| < 1) = a / (1 - r)
```

3. **SUM FORMULAS**

```
// Sum of first n natural numbers
sum = n * (n + 1) / 2

// Sum of squares
sum = n * (n + 1) * (2n + 1) / 6

// Sum of cubes
sum = (n * (n + 1) / 2)^2
```

4. **FAST EXPONENTIATION**

```
ll power(ll a, ll b) {
    ll res = 1;
    while(b > 0) {
        if(b & 1) res *= a;
        a *= a;
        b >>= 1;
    }
    return res;
}
```

5. **DIGIT SUM**

```
int digitSum(int n) {
    int sum = 0;
    while(n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
```

```
}
```

6. **TRAILING ZEROS IN FACTORIAL**

```
int trailingZeros(int n) {
    int cnt = 0;
    for(int i = 5; n/i > 0; i *= 5)
        cnt += n/i;
    return cnt;
}
```

7. **NUMBER OF DIGITS**

```
int digits = floor(log10(n)) + 1;
// Or: to_string(n).length()
```

BIT MANIPULATION

Basic Operations

```
// Check if i-th bit is set
bool isSet = (n >> i) & 1;
bool isSet = (n & (1 << i)) != 0;


// Set i-th bit
n |= (1 << i);


// Clear i-th bit
n &= ~(1 << i);

// Toggle i-th bit
n ^= (1 << i);

// Count set bits
int cnt = __builtin_popcount(n);      // int
int cnt = __builtin_popcountll(n);    // long long

// Lowest set bit
int lowBit = n & (-n);

// Check if power of 2
```

```
bool isPower2 = (n > 0) && ((n & (n-1)) == 0);

// Remove last set bit
n = n & (n-1);
```

XOR Properties

```
a ^ a = 0
a ^ 0 = a
a ^ b ^ b = a
XOR is associative and commutative
```

Useful Tricks

```
// Swap without temp
a ^= b ^= a ^= b;

// Check if opposite signs
bool oppSigns = ((a ^ b) < 0);

// Multiply/divide by 2
n << 1  // n * 2
n >> 1  // n / 2

// Check odd/even
if(n & 1) // odd
```

STRING ALGORITHMS

1. **STRING FUNCTIONS**

```
string s = "hello";

s.length() or s.size()     // Length
s.substr(pos, len)         // Substring
s.find("ll")               // Find substring (returns pos or string::npos)
s.rfind("l")               // Find from right
s += "world"               // Concatenate
s.push_back('!')           // Add char at end
s.pop_back()               // Remove last char
s.erase(pos, len)          // Erase substring
s.insert(pos, "hi")        // Insert at position
```

```
// Convert
stoi(s)                    // String to int
stoll(s)                   // String to long long
to_string(n)               // Number to string


// Character checks
isalpha(c), isdigit(c), islower(c), isupper(c)
tolower(c), toupper(c)
```

2. **PALINDROME CHECK**

```
bool isPalindrome(string s) {
    int l = 0, r = s.length()-1;
    while(l < r) {
        if(s[l++] != s[r--]) return false;
    }
    return true;
}
```

3. **ANAGRAM CHECK**

```
bool isAnagram(string s1, string s2) {
    sort(s1.begin(), s1.end());
    sort(s2.begin(), s2.end());
    return s1 == s2;
}
```

4. **FREQUENCY MAP**

```
map<char, int> freq;
for(char c : s) freq[c]++;
```

QUICK TIPS & TRICKS

Input/Output

```

```

```
// Read until EOF
int x;
while(cin >> x) { }

// Read line
string line;
getline(cin, line);

// Precision
cout << fixed << setprecision(10) << ans;
```

Debugging Tricks

```
#define debug(x) cerr << #x << " = " << x << endl;
debug(variable);

// Print vector
for(int x : v) cerr << x << " ";
cerr << "\n";
```

Constants

```
const int INF = 1e9;
const ll LLINF = 1e18;
const int MOD = 1e9 + 7;
const double EPS = 1e-9;
```

Extra

1. STL Shortcuts & Utilities

`is_sorted` / `is_sorted_until`

Check if a vector is already sorted to avoid unnecessary sorting.

```
vector<int> v = {1,2,3,5,4};

// Check if sorted
if(is_sorted(v.begin(), v.end()))
```

```
    cout << "Already sorted\n";
else
    cout << "Not sorted\n";

// Find first unsorted element
auto it = is_sorted_until(v.begin(), v.end());
cout << "First unsorted value: " << *it << "\n";
```

any_of, all_of, none_of

Check conditions on ranges in a single line.

```
vector<int> v = {1,2,3,4,5};

if(any_of(v.begin(), v.end(), [](int x){ return x < 0; }))
    cout << "Some negatives\n";

if(all_of(v.begin(), v.end(), [](int x){ return x > 0; }))
    cout << "All positive\n";

if(none_of(v.begin(), v.end(), [](int x){ return x == 0; }))
    cout << "No zeros\n";
```

clamp

Restrict a value to a range [low, high].

```
int val = 120;
int x = clamp(val, 0, 100); // x = 100
```

iota

Fill a vector with consecutive numbers.

```
vector<int> v(10);
iota(v.begin(), v.end(), 1); // {1,2,3,...,10}
```

rotate

Rotate a range efficiently.

```
vector<int> v = {1,2,3,4,5};
rotate(v.begin(), v.begin()+2, v.end()); // {3,4,5,1,2}
```

`partial_sort` / `nth_element`

Get k-th smallest/largest element without full sort.

```
vector<int> v = {7,2,1,6,8,5,3,4};
nth_element(v.begin(), v.begin()+3, v.end());
cout << "4th smallest: " << v[3] << "\n";
partial_sort(v.begin(), v.begin()+4, v.end()); // Sort first 4 smallest elements
only
```

2. Bit Manipulation & Masks

`bitset`

Fast operations on bits.

```
bitset<8> b("10101010");
cout << b.count() << "\n";
b.flip(0);
b.set();
b.reset();
```

Iterate all subsets of a bitmask

```
int mask = 0b111;
for(int sub = mask; sub; sub = (sub-1) & mask) {
    cout << bitset<3>(sub) << "\n";
}
```

`__builtin` shortcuts

```
int x = 40;
cout << __builtin_popcount(x) << "\n";
cout << __builtin_ctz(x) << "\n";
cout << __builtin_clz(x) << "\n";
cout << __builtin_parity(x) << "\n";
```

3. Pairs & Tuples

`tie` for unpacking pairs

```
pair<int,int> p = {5,10};
int a,b;
```

```
tie(a,b) = p; // a=5, b=10
```

Tuple auto-lexicographical comparison

```
tuple<int,int,int> t1={1,2,3}, t2={1,3,2};
if(t1 < t2) cout << "t1 comes first\n";
```

4. Math / Utility Tricks

Fast ceil of integer division

```
int a=7, b=3;
int res = (a+b-1)/b; // ceil(a/b) = 3
```

Multiple min / max

```
int m = min({a,b,c,d});
int M = max({a,b,c,d});
```

Sign of number

```
int sign = (x > 0) - (x < 0); // 1,0,-1
```

5. Vector / STL Hacks

Erase-remove idiom

Remove elements from a vector efficiently.

```
vector<int> v = {1,2,3,2,4,2};
v.erase(remove(v.begin(), v.end(), 2), v.end());
```

`distance` – iterator → index

```
auto it = find(v.begin(), v.end(), 4);
int idx = distance(v.begin(), it);
```

`swap` for pairs, arrays, tuples

```
pair<int,int> p1={1,2}, p2={3,4};
swap(p1,p2); // p1={3,4}, p2={1,2}
```

Lambda for small utilities

```
auto sq = [](int x){ return x*x; };
cout << sq(5); // 25
```