**＜C++復習＞**

**○最近よくやるミス**

・intした後にcinで入力するのを忘れる→後になると意外に気づきにくい

・for(int i = x; i <= n; i+=x)とするときにx=0だと無限ループしてしまう。意外に気づきにくい

・int ans;としたが計算している途中にオーバーフローすることがあるとWAになる。提出してから少しだけWAになるのなら全部ll ans;にしてみるとよいかも

**○エラーの出ない恐ろしいミス**

・配列時のアクセスに「a.at(i)」ではなく「a[i]」を用いると、iが配列の要素数を超えてもエラーにならない

・数値型がオーバーフローすると最大値になってしまう

int型　　　　→

long long型　→

long long型でも意外に大きくないからオーバーフローに注意する

例）積のオーバーフロー回避

if (ans\*a[i] > pow(10,18)) return 0;

ans \*= a[i]

と書くとの時点でオーバーフローになるから割り算にする

if (a[i] > pow(10,18)/ans) return 0;

ans \*= a[i]

・小数点の闇、誤差

double型で2.51と入力したときに実は「2.50999999999999971…」と誤差が生じている。これに対してをかけると誤差によって一の位がおかしいことになる

→いったん整数に直す

例）小数を整数に直し、ちょっと足して、後から戻す(bは小数第二位まで)

long long a;

double b;

long long b2 = b\*100 + 0.5;

long long ans = a \* b2;

a /= 100;

・long doubleやdoubleは10^6を超えると1e+06という表記になってしまう。

→coutではなくprintf(“%.10f”, ans);とするとちゃんと100000と書いてくれる

・a\*b = Nの組み合わせを全部試すときは√N回で十分（i=1からi\*i<=nまで）だが、for分の中でll iとしないとオーバーフローしちゃう。

**○最近知ったびっくりしたこと**

・string型の足し算ができる!

string s = “abc”;

s += ‘d’; // abcd

s += “efg”; // abcdefg

・string型も配列と同じようにreverse、sortができる！

reverse(s.begin(),s.end()); // gfedcba

・queueも配列と同じようにfor(auto p : que)だったり、reverseできたりする

・配列の最大値にアクセスするときにはpriority\_queueを使うと計算量O(logN)でいける！

**○注意事項**

・べき乗の計算をするときにpow(x, n)とするのは計算量が多くなるからあまり良くないら。テイラー展開をしてるとかで大変らしい。新しい変数を用意してあげたほうが早い

例)ｘ^n

int xn = 1;

rep(i,n) xn \*= x;

・ビットを使うときに「1<<n」と書くと2^nになるがint型である。long long型にしたいときは「1ll<<n」とかく

・nCkを計算するのはめっちゃ大変で計算量が半端ない。ただし、「1,2,…,n」の中からk通りならいける方法がある。k通り選ぶときの最小の和は1+2+…+k、最大の和はn+(n-1)+…(n-k+1)となり、この間の値は全てとれるので「(最大の和) - (最小の和) + 1」で求められる。

・かけ算と割り算があるときは割り算を先にやると良い。かけ算でオーバーフローするかも

・reverseは計算量O(配列の長さ)かかる

・配列の後ろに追加するのは計算量O(1)だが、前に追加するのはO(配列の長さ)かかってしまう。dequeを用意して変数名.push\_front()すれば計算量O(1)で済む。

**○技**

・素因数分解

ｎを素因数分解するときは√nまででよい。√nのときに1ならこれ以降に素数はないし、1でないなら素数が1つのみあるということがわかる。理由は、もし√n以降に素因数が2つ以上あるならそれらをかけただけでｎを超えるから。

vector<pair<ll,int>> fs; // pair型の配列

for(ll i=2; i\*i<=n; i++){ // √nを超えたら終わり、intだとi\*iでオーバーフローする

int kosuu=0;

while(n%i==0){

kosuu++;

n/=i;

}

fs.emplace\_back(i, kosuu); // pair型配列の末尾に（素因数, 個数）を追加

}

if(n!=1) fs.emplace\_back(n, 1); // √nを超えたときに1でないなら追加

・割り切れる、割り切れないの問題で愚直にやるとオーバーフローする場合は、配列の中で一番大きい値の長さの新しい配列を作り、倍数のカウントをするとよい。例えば、3が現れたら配列の3，6，9，12…番目に＋1する。計算量はNlogNとなる。

vi a(n);

sort(a.rbegin(),a.rend());

int len = a[0]+1;

vi baisuu(len); // 倍数をカウントする配列

for(int x : a){

for(int i = x; i <= len; i += x); yakusuu[i]++; // a[i]おきに＋1する

}

・配列の部分配列を扱う

→もし隣同士で大小比較などで済むのなら部分配列は作る必要はない。1つの変数に突っ込み足したり引いたりしてなんとかすると良い。また、実際に書いてみることで比較する部分が配列の中の一部分だったりもする

・円環の問題

→円環を切り開いて、少し足した配列を作ると良い。2周、3周したほうが良いこともある。こうすると原点をまたぐときの場合分けをせずに済む！

・複数の配列をまたいで大きい順（小さい順）に取り出したい

→新しい配列に「push\_back」して詰め込んで新しい配列をsortすれば、すっきりとできる

・とてつもなくでかい数の倍数判定

→大きくなる前にmodをとればよい。modはans %= n;とすればmod nになる

・ｎ個の値から2つを選ぶのをそれぞれやると計算量O(N^2)でオーバーする。そこで、mapを用意して一つの変数を突っ込めば計算量O(N)でいける。もう一つの変数を回して前者の個数分答えに加算する。

例）和と差が等しいのは何個あるか

for(int i=1; i<=n; i++){

int sa = i – a[i]; // 回す差の変数

ans += mp[sa]; // 差と等しい和があったらその分答えに加算する

int wa = i + a[i]; // メモする和の変数

mp[wa] ++; // mapに突っ込む

}

**○基本**

・「#include <bits/stdc++.h>」「using namespace std;」はとりあえず毎回最初に書く。コピペ。

・main関数は「int main(){}」の中に書く（ほぼ全部ここ）

・コピペ用

#include <bits/stdc++.h>

using namespace std;

#define rep(i, n) for (int i = 0; i < (int)(n); i++)

using ll = long long;

using vi = vector<int>;

using vvi = vector<vi>;

int main(){

}

・行の最後には「;」

・改行→endl

・文頭に「//」をつけるとコメントになる。出力したくないときにも使う。

・「tabキー」でインデント（空欄）

・実行順序は上から

・条件式の結果は真のとき1に、偽のとき0になる

・1をtrueで、0をfalseで表す（条件式の中で直接1や0を使うこともできる）

・プログラムを途中で終わらせたい→「return 0」

・行単位での入力

string s, t;

getline(cin, s);

getline(cin, t);

**○便利機能**

・int型はくらいまでしか扱えない。それ以上にすると最大値2147483647になる

→long long型にする

・有効数字の決め方（double型）→ cout << fixed << setprecision(桁数)をいれる

double y=3.1415926535;

cout << fixed << setprecision(10);

cout << y << endl;

・実行時間の測り方：強いものだけを考える

定数　＜　N　＜　logN　＜　　＜　NlogN　＜

【型を変えるときの記述方法】

|  |  |  |
| --- | --- | --- |
| 型変換 | 書き方 | 例 |
| int型→ double型  double型→int型 | (変える後の型)変数 | int a;  double b = (double)a; |
| int型→string型  double型→string型 | to\_string(変数) | int num =100;  string s = to\_string(num) |
| string型→int型 | stoi(変数) | string s = “100”;  int a = stoi(s); |
| string型→double型 | stod(変数) | string s = “3.14”;  double b = stod(s); |

・auto

→初期化を伴って変数を宣言する場合や範囲for文において、型の部分にautoと書くことによって型を勝手に決めてくれる

string s = “hello”

aut a = s; //string型

vector<int> c = {1, 2, 3};

auto d = c; // vector<int>型

例）pair型配列の回し方

pair 変数名;

for(auto p: 変数名){}

forの中で使うときはp.firstやp.secondとする。pは変数名[i]と等価。こうするとpair型配列の長さがわからないときに最後まで勝手に回してくれるから便利。

・型エイリアスを使って、長い型名を省略できる（using 真型名 = 旧型名;）

例）long long → ll

using long long = ll;

例）vector<int> → vi , vector<vector<int>> → vvi

using vi = vector<int>;

using vvi = vector<vi>;

・配列を回すときは範囲for文がめちゃんこ便利

→配列の要素を1文字で表せ、便利だしわかりやすい

vector<int>vec(N);

for(int x : vec){

cout << x << endl;

}

・部分文字列→string型の一部を取り出す「変数名.substr(左, サイズ)」

サイズを省略すると末尾まで指定になる

例

string s = “abcdefg”

string former = s.substr(0,3); //abc

string latter = s.substr(3); //efg

・アルファベットが小文字か大文字かの判定はislower()、isupper()でするとよい

()の中のアルファベットがそれぞれ小文字、大文字なら trueを返し、他にはfalseを返す。

・数字かどうかの判別にはisdigit()がある。数字ならtrue、違うならfalseを返す。

・配列を大きい順（降順）に並べる「sort(v.rbegin(),v.rend())」

・最大公約数（GCD）は「gcd(a, b)」とする。実装はユークリッドの互除法を使っている

・最小公倍数（LCM）は「lcm(a,b)」とする。または、「a\*b/gcd(a, b)」とする。

・3つの最大公約数を求めるときは「gcd(a, gcd(b, c))」とする

・2変数を交換したいときは「swap(a,b);」

【set】

・setは重複する要素を数えない集合である。入れた時点で昇順にsortされる。

・宣言は「set<型>名前;」

・要素の追加は「名前.insert(追加するもの)」

・要素のサイズ→「名前.size()」

・setの要素のサイズは重複していない要素の個数

・「名前.count(要素)」で要素が入っているかを確認できる。入っていたら1を返し、入ってなかったら0を返す。

例）いろんな文字列をsetの中に入れる

set<string> a;

rep (i, n){

string s;

cin >> s;

a.insert(s);

}

**○計算量の減らし方**

・for文2つをそのまま使うとO(N^2)となるから、違う視点から配列をとると良い。

例）約数の個数

1, 2, ･･･ , nとそれぞれの約数の個数を調べるのではなく、横から見るイメージでそれぞれの約数の個数を配列にとり、約数1をもつものにインクリメント、約数2をもつものにインクリメント、･･･とするとO(NlogN)になる！2つめのfor文の書き方が特殊。

for(int i = 1; i <= n; i++){

for(int j = i; j <= n; j += i) yakusuu[i] ++;

}

**○2分探索**

・2分探索は「ソート済み配列の中から目的のものを高速に探索する」ものである。本来だとforの中にforを書くので計算量O(NM)だが、2分探索だとO(NlogM)となる。

・ソート済み配列は言い換えると、単調増加の配列。単調増加なら2分探索でいける！

・leftとrightとmidを用意して半分のうち狙った値はどちらにあるかを繰り返す。

・下のテンプレをそのままいつでも使うことができる。

テンプレ（keyより小さい最大の値を配列aから探す）

int left = -1, right = a.size();

while(right – left > 1){

int mid = (right + left) / 2;

if(a[mid] > key) right = mid; // keyより大きい最小の値を探すなら>=にする

else left = mid;

}

・入力ではそれぞれの時間が与えられているかもしれないが、配列にはそれぞれ和をとっておくことで2分探索をする条件が整う

vi a(n+1);

a[0] = 0;

int sum = 0;

for(int i = 1; i <= n; i++){

int tmp;

cin >> tmp;

sum += tmp;

a[i] = sum;

}

**○しゃくとり法**

・2分探索のように「条件を満たす区間から最小（最大）の長さを求める」または「条件を満たす区間を数え上げる」ときに使える

・leftを端に固定しrightを端から1つずつ右にずらし、条件を満たさなくなったら仕方なくleftを右に動かす。この動きが尺取り虫みたい

・2つのポインタ（left, right）を動かすのが特徴

・計算量は2分探索より小さくO(N+M)

テンプレ

int right = 0;

for (int left = 0; left < n; left ++){

while(right < n && 「rightを1つ右にしたときに条件を満たす」){

実際にrightを右に1つ進める。例) sum += a[right];

right ++;

　　}

　　whileを抜けた時点でrightは条件を満たす最大なのでなにかする;

　　leftを右にずらす

　　例） if(right = left) right ++;

　　　　else sum -= a[left];

}

**○幅優先探索（BFS）**

・無向グラフの問題で、ある場所からの最短距離を全ての頂点について求める

・確率が等しいときに使える（各辺の重みが等しい）

・計算量はO(辺＋頂点) 。軽い！！

・無向グラフを2次元配列で、距離をdist配列（-1で初期化して-1のときは未訪問と考える）で、訪問予定の待ち行列をqueueでやる。

・スタートの頂点から考えるのでqueueに突っ込み、distを0とする。その頂点に隣接している頂点が未訪問であることを確認してから距離を＋1して、queueに突っ込み後で隣接する頂点を調べていく。という流れ

例）頂点0から各頂点への最短距離

int n, m;

cin >> n >> m;

vector<vector<int>> graph(n); // 頂点が０からではないときはn+1とする

rep(i,n){

int a,b;

cin >> a >> b;

graph[a].push\_back(b); // 無向グラフを作る

graph[b].push\_back(a); // 無向グラフを作る

}

vector<int> dist(n, -1); // 距離を-1で初期化、頂点が０からでないならn+1とする

queue<int> que;

dist[0] = 0; // スタートの頂点の距離を0にする

que.push(0); // スタートの頂点をqueに突っ込む

while(!que.empty()){ // 全ての頂点が訪問済なら終了

int v = que.front(); // 先頭のqueを調べる

que.pop(); // 先頭のqueを消す

for(int nv : graph[v]){ // 頂点に隣接している全ての頂点を取り出す

if(dist[nv] != -1) continue; // 訪問済なら打ち切る

dist[nv] = dist[v] + 1; // 隣接している方が1遠いはず

que.push(nv); // 次に調べるためにqueに突っ込む

}

}

**○順列**

・順列とは配列に順序をつけて並び替えるものである。

・全部異なるのならn!通りある。計算量がめっちゃ大きくn=10でもオーバーする

・next\_permutation(v.begin(),v.end()); とすると配列を辞書順に次の配列に変換する。変換できないときは-1を返す。

・実際に使うときは事前にソートをしてからdo-whileを使う。ソートしないと途中で終わってしまう

例、{1,2,3,4}の順列を書き出す

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

sort(v.begin(),v.end()); // まずソートする

do{

rep(i,4) cout << v[i];

cout << endl;

}while(next\_permutation(v.begin(),v.end())); // 次の順列にする。最後までいったら抜ける

**○よくわからないこと**

・参照

・再帰関数

**○いろいろな数式**

|  |  |
| --- | --- |
| 三角関数 | sin()、cos()、tan() |
| 逆三角関数 | asin()、acos()、atan() |
| 双曲線関数 | sinh()、cosh()、tanh() |
| 底eの指数関数 | exp() |
| 底10の対数関数 | log10() |
| 平方根 | sqrt() |
| 立方根 | cbrt() |
| 絶対値 | abs() |
| 累乗 | pow(a,b) |

例）平方根と三角関数と累乗

cout << sqrt(9) << sin(PI\*0.5) << pow(10,3) << endl;

→3、1、1000

・三角関数などπが必要なときは「#define PI acos(-1)」を加える。使うときは「PI」とする

**○他の記述方法**

・配列や文字列にアクセスする際の「変数名.at(数字)」は「変数名[数字]」とも書ける

しかし後者だとエラーメッセージが出ないことに注意

【printfでの出力】

・printf(“変数の混じった文字列”,変数1,変数2);

・printfを使う場合、変数のところにフォーマット指定子をいれる

・改行は「＼ｎ」

| **型** | **フォーマット指定子** |
| --- | --- |
| int | %d |
| double | %lf |
| char | %c |

例)ゆうとん3

int a = 3;

printf(“ゆうとん　%d＼n”);

・フォーマット指定子を「%03d」とすると3桁表記となる（桁数に満たないときは0埋め）

【scanfでの入力】

・scanf(“入力欄”,＆変数1, ＆変数2)

・入力欄には文字列も入れられる

**〇競技プログラミング用知識**

・Xがｎで割り切れるか？

→X％ｎが０となればよい

・Xをaで割ったときの余りで分けるのはよく使う（数学でいうmod）

if(X%a == 0){処理}

if(X%a == 1){処理}

if(X%a == 2){処理}

・bool型の配列はめっちゃ便利。重ねて調べるときも重複せずに済む。for型のループ内の変数のスコープも気にせずに済む

vector<bool>ans(N,true); //N個、すべてtrue、bool型の配列

・プログラムの中で「/」をしたときにおかしいことになることが多い

1. 小数点以下切り捨ての関係でWAになる

→intではなくdoubleにする

1. 0で割ってしまいREになる

・AtCoderコンテストでの実行時間の制約は2秒→2×回の計算までしかできない

・訪れたところのメモの仕方

→要素数０の配列を用意して、訪れたらpush\_backする

・同じところに訪れたかの判定

→要素すべてをtrueにしたbool型、もしくは要素すべてを-1にした配列を利用する。訪れたらその要素を変えて、もう一度訪れた時にわかるようにしておく

・pareやtupleで1個目は辞書順だけど、2個目は大きい順に並べたいとき（ふつうなら数字は小さい順になっちゃう）

→2個目の変数に「-」（マイナス）をつける

・2次元の表を作り直すときは要素すべてfalseのbool型の配列を2つ用意するといい。条件を満たせばtrueにしていって、trueの場所だけ表を出力すれば良い

1. 縦横両方の条件を満たすときのみ出力するなら→if(bool1 && bool2){}
2. 少なくとも一方の条件を満たすときのみ出力するなら→if(bool1 || bool2){}
3. 行や列を消したいとき→if(bool1){}とif(bool2){}の入れる場所を工夫する

・座標の取り方（座標上で長さを計算できる）

→x座標とy座標を別にとる

例）任意の2つの座標の距離

double x1, y1, x2, y2;

cin >> x1 >> y1 >> x2 >> y2;

double dx = x1 - x2, dy = y1 -y2;　//x座標とy座標それぞれの差をとる

double ans = sqrt(dx\*dx + dy\*dy); //2乗の平方根（距離）

cout << ans << endl;

・指定されたもの以外を扱いたいときにはbool型の配列でフラグを立ててループの中の最初で終わらせるとよい。「if(hantei[i]) continue;」

・数字の配列が与えられ、普通にやったら計算量がオーバーしてしまうときは配列のかわりにそれぞれの数字のカウンタ（vector）を用意すると良い。配列の総和は「カウンタの個数×カウンタの番号」を回せば良い

例）配列の総和

vector<int>count(100001);

rep(i,n) {

int a;

cin >> a; // 入力の受け取り

count[a] ++; // カウンタに入れる

}

int sum=0;

rep(i,100001) sum += i\*count[i]; // 総和の計算

イメージ図

配列　21342232

カウンタ 1：1個

　　　　　2：4個

　　　　　3：2個

　　　　　4：1個

・割り算で切り上げるには割る数に割られる数-1を足せばよい

例）1000で割り算をして切り上げる

ans = (n + 1000 - 1) / 1000

**○bool型**

・この型の変数にはtrueまたはfalseだけが入る

・bool型でtrue のときにif型を使うなら「if(bool型変数名)」だけでよい

bool a = true

bool b = false

bool c = true

if (a) {cout << "At";}

else {cout << "Yo";}

if (!a && b) {cout << "Bo";}

else if (!b || c) {cout << "Co";}

if (a && b && c) {cout << "foo!";}

else if (true && false) {cout << "yeah!";}

else if (!a || c) {cout << "der";}

→「AtCoder」と出力

**○出力**

・出力したい→cout <<、出力したいとこを””で囲む。

・"Hello, world!"とendlというデータを、<<でcoutに送っていくイメージ

・ 「 cout << "Hello, world!" << endl;」→Hello, world!

**○入力**

・入力したい→cin >>（入力したものをその先に送るイメージ）

int a;  （変数aの宣言）

cin >> a; 　（入力をaに適応）

cout << a \* 10 << endl;　（入力したものに10かける）

・入力したもの同士の演算

int A,B;

cin >> A >> B;

cout << A+B <<endl;

**○エラー**

・全角スペースがあるとコンパイルエラー（CE）になる

・「;」忘れもCE

・エラーメッセージの中でエラーの発生場所を教えてくれる

→./Main.cpp:5:19: error:だと5行目19文字目

→「^」で直接教えてくれる

・string型変数.at(i) == “t” はだめ。char型だから’t’

・「＝」の個数に注意！定義や初期化は「＝」、条件は「＝＝」

・cout の中に配列変数名を入れるのはだめ

→とりだすなら.at(i)

・cout の中にpair変数名を入れてもだめ

→tie(a,b) = で分解してそれぞれ出す　cout << a << “ “ << b << endl;

**○演算**

・足し算cout << 1 + 1 << endl;

・引き算cout << 3 - 4 << endl;

・かけ算cout << 2 \* 3 << endl;

・割り算cout << 7 / 3 << endl; （有効数字一桁なので答えは2）

・％→割り算のあまり（例、5%2→1）

| **演算子** | **意味** |
| --- | --- |
| x == y | xとyは等しい |
| x != y | xとyは等しくない |
| x > y | xはyより大きい |
| x < y | xはyより小さい |
| x >= y | xはy以上 |
| x <= y | xはy以下 |

【注意】「＝＝」のつもりで「＝」とするミスが多い

・演算子（＋－＊/）の入力を反映させる方法

→文字列を変数化して、入力に対応させ、if文でそれぞれに対応させる

string op;

int x;

cin >> op >> x;

if(op=="+"){

A += x; }

if(op=="-"){

A -= x;}

if(op=="\*"){

A \*= x;}

if(op=="/" && x != 0){

A /= x;}

if(op=="/" && x == 0){

cout << "error" << endl;

break;}

| **演算子** | **意味** | **真になる時** |
| --- | --- | --- |
| !(条件式) | 条件式の結果の反転 | 条件式が偽 |
| 条件式1 && 条件式2 | 条件式1が真 かつ 条件式2が真 | 条件式1と条件式2のどちらも真 |
| 条件式1 || 条件式2 | 条件式1が真 または 条件式2が真 | 条件式1と条件式2の少なくとも片方が真 |

・変数ｘの値を変えたいとき（方程式ではない）

**x = x + y**は**x += y**のように短く書ける（**+,-,\*,/,%**で使える）

**x += 1**は**x++**と書ける（1だけ足したい→インクリメント）

**x -= 1**は**x--**と書ける（1だけ引きたい→デクリメント）

以降のxはすべて変更した後の値になる

**○変数**

・変数はメモ

| **型** | **書き込むデータの種類** |
| --- | --- |
| int | 整数 |
| double | 小数 |
| string | 文字列（1行単位） |
| char | 文字（１文字のみ） |

・変数の宣言→int name;

name = 10;

またはint name = 10;でもよい

・例）string name=hello

・複数の宣言→int a=5,b=10;

・同ブロックの中で同じ型を二回宣言してはだめ。intと二回書くな。

・name=なら何回やってもいい。同じnameなら書くたびに更新される

・｛｝の中を**ブロック**という

・あるブロックの中で宣言した変数は、それより内側のブロックでしか使えない。その範囲のことをスコープという

#include <bits/stdc++.h>

using namespace std;

int main() {

int x = 5; // xのスコープはこの行からmain関数のブロックの終わりまで

if (x == 5) {

int y = 10; // yのスコープはこの行からif文のブロックの終わりまで

cout << x + y << endl; }

cout << x << endl;

cout << y << endl;}

上記の記述だとyのスコープがifの中だけなので最終行のyに対応しなくなりCEになる

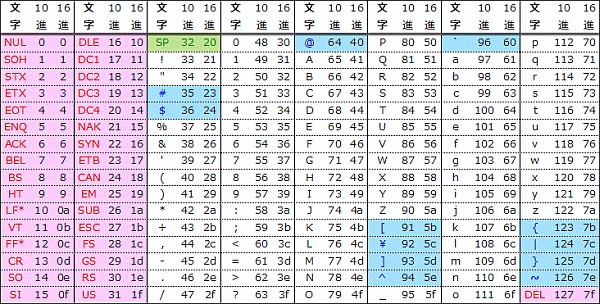
・異なるブロックなら同じ名前の変数を宣言できる

・外にある変数なら使える

・外と中に同じ名前の変数が宣言されたら、中のほうが優先される

**○文字コード（アスキーコード）の扱い**

・char型の文字にはそれぞれ文字コードが存在する。



・’1’の文字コードは1ではなく49。’A’の文字コードは65

・辞書順でアルファベットをn個進めたいときは「s.at(i) = ( s.at(i) – ‘A’ +n )%26 + ‘A’とする

【何をしているか】

Zの次がAとしたい。Zをこえたときに戻るために26で割ったあまりを使う

1. s.at(i)はchar型
2. 「-‘A’」をすることで文字コードをいったん0からにし、進めたい分ｎをたす
3. 26を超えたときに0に戻すために26で割ったあまりを使う
4. 「-‘A’」した分を戻すため「+’A’」する

・1桁ではない整数から1つずつ数字を取り出したいとき→「-‘0’」する

char型（string型で取り出すとき）からint形に変換するときに使う。int型として使うならそのままでいいが、string型に突っ込むときは＋’0’をしないとおかしいことになる

string s = “0120”;

vector<int> v (s.size());

rep(i,s.size()) a.at(i) = s.at(i) – ‘0’;

**○pairとtuple**

【pair】

・pair型は2つの値の組を表す

・「pair<値1の型, 値2の型> 変数名;」で宣言する

・「変数名.first」で1番目の値、「変数名.second」で2番目の値にアクセス

・「make\_pair(値1, 値2)」でpairを生成

・「tie(変数1, 変数2) = pair型の変数;」でpairを分解

・cout の中にpair変数名を入れてもだめ

→tie(a,b) = で分解してそれぞれ出す　cout << a << “ “ << b << endl;

例）pair（a,b）を（c,d）にして出力

int a,b;

pair<int,int> p;

int c,d;

tie(c,d) = p;

cout << c << “ “ << d << endl;

【pairの配列】

・宣言はvector<pair<型、型>> 変数名 (個数);

・入力はrepマクロの中に　変数名.at(i) = make\_pair(値1, 値2)

・アクセスはrepマクロの中で　変数名.at(i).first;　または　変数名.at(i).second;

・末尾に要素を追加するときは　変数名.emplace\_back(a, b);

【tuple】

・tuple型は複数個の値の組を表す

・「tuple<値1の型, 値2の型, 値3の型, (...)> 変数名;」で宣言

・「make\_tuple(値1, 値2, 値3, (...))」でtupleを生成

・「tie(変数1, 変数2, 変数3, (...)) = tuple型の変数;」でtupleを分解

・「get<K>(tuple型の変数)」でK(定数)番目にアクセス

Kは定数。変数だとCEになる

【共通】

・型が同じpairやtuple同士で比較すると、1番目の値から比較され、1番目の値が等しければ、2番目の値の比較が行われる（pairの配列でよく使う）

例）pairの配列をsortする(aに注目して小さい順に並べる)

vector<pair<int,int>> ab (N);

rep(i,N){

int a,b;

cin >> a >> b;

ab.at(i) = make\_pair(a,b);

}

sort(ab.begin(),ab.end());

・pairやtupleを分解する際に要らない要素を捨てたい場合、ignoreをtieの引数に渡すことで、対応する位置の値を捨てられる

pair<int, int> p(3, 5);

int right;

tie(ignore, right) = p; // 2番目の値だけ取り出す

**〇文字列と文字**

・「文字列変数.size()」で文字列の長さを取得できる

・「文字列変数.at(i)」でi文字目にアクセスできる

【注意】

変数.at(i)で取得するデータはchar型なので囲むときは「‘」で！！

・iを添え字といい、添え字は０から始まる

・char型を表すときには「’’」で挟む

char c = ‘a’

・入力した文字の中にOが何個あるかを数える方法

string str;

cin >> str;

int count = 0;

for (int i = 0; i < str.size(); i++) {

if (str.at(i) == 'O') {

count++;

}

**○if文**

・if (条件式) {処理}

→条件が正しいときのみ作用する

→処理が1行で書けるなら｛｝を省略可能

・ifの後ろにelse

if (条件式1) {処理1}

else {処理2}

→それ以外の時に作用する

・else if

if (条件式1) {処理1}

else if (条件式2) {処理2}

→条件式1が偽 かつ 条件式2が真 のときに作用する

【注意】

・if(a =b=c)はだめ！！

・if(a=b && b=c)ならOK

**○ループの基本**

・ループの書き方がわからないときはとりあえず何個か書いてみて、パターンを見つけ、ループで書き直す

・ループ構文の使い分け

1. 配列の全ての要素に対する処理を行なう場合 → 範囲for文
2. 単純な繰り返し→repマクロ
3. それ以外で一定回数繰り返し処理する場合 → for文
4. それ以外の場合（繰り返す条件がi<Nでないとき） → while文

**○ループ（repマクロ）**

・最初の2行に「#define rep(i, n) for (int i = 0; i < (int)(n); i++)」をいれ、宣言する

・指定した回数繰り返す　rep(i,回数)｛処理｝

→処理が1行で書けるなら｛｝は省略可能

・カウンタ変数iは0から始まり、「回数－1」で終わる

・回数を入力する変数にして応用できる

int N;

cin >> N;

int sum = 0; // 合計点を表す変数

int x; // 入力を受け取る変数

rep(i, N) {

cin >> x;

sum += x;}

cout << sum << endl;

　と記述すれば合計点を計算できる（sum==0とすることに注意）

・ループを抜ける→break;(ifと一緒に使いやすい)

if(条件)｛break;｝

・後の処理を飛ばして次のループへ→continue(ifと一緒に使いやすい)

if(条件)｛continue;｝

**○ループ（while文）**

・while文を使うと繰り返し処理ができる。条件式が真の時に処理を繰り返す

・一般的にカウンタ変数は0から初めてNより小さいときまでにする

int N;

cin >> N;

int i = 0;

while (i < N) {

処理

　i++;}

**○ループ（for文）**

・for文はwhile文の上位互換のようなもの

・for (初期化; 条件式; 更新) {処理}

・ループを作るときには一般的に下記の通りにする  
for (int i = 0; i < N; i++) {処理}

・iが1ずつ増えながらN回処理を繰り返す

・while文よりもスコープが狭くなる。forの{}の中だけ

【注意】

・forの（）の中にintを忘れやすい

・forの（）の中の区切りは「,」ではなく「;」

・for文の外では変数iが使えない（while文なら使える）

**○ループ（範囲for文）**

・範囲for文は配列のすべての要素について同じ処理を行いたいときにfor文を省略できる

・回す際に配列の要素を1文字で指定できるのは楽だし、わかりやすい

for (配列の要素の型 変数名 : 配列変数) {各要素に対する処理}

例）配列の各要素を出力

vector<int> a = {1, 3, 2, 5};

for (int x : a) {

cout << x << endl;//「1 3 2 5」と出力

}

【注意】

forの（）の中は「；」ではなく「：」

**○多重ループの使いどころ**

・総当たりで調べたいときに超よく使う

・ｍ個の要素とｎ個の要素それぞれ一つずつ取り出し、条件に合うかを確かめたいときに使う

例）ｍ個の要素とｎ個の要素が何個同じか

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

if (A.at(i) == B.at(j)) {

count++;

}

}

}

【注意】

・多重ループの中から抜けたいときはループを抜けるかどうかを持つ変数(フラグ変数)を用意してフラグ変数の値に応じてループを抜けるように書く

・添え字の「i」と「j」を間違いやすいから注意

・添え字を数字として使うとき、いつものようにi<nと書くとミスるかも。i+1では？

**○配列**

・配列は文字だけでなく、数字も扱える。for文と組み合わせることで初期化することなく複数の変数を扱える

・「vector<型>配列変数名;」で配列変数を宣言できる

・配列変数名 = { 要素1, 要素2, ... };で配列変数に値を代入できる

・「vector<型> 配列変数名(要素数) 」で要素数を指定できる

・配列変数.at(i)でi番目の要素にアクセスできる

・配列変数.size()で配列の要素数を取得できる

・vector<int> vec(3);はvector<int> vec = {0, 0, 0}とほとんど同じ意味（初期化）

・vector<int> vec(3, 5);と書いた場合、配列変数vecは{5, 5, 5}で初期化される

・配列でN個の入力を受け取るときは、要素数Nの配列変数を宣言してから、for文の中で.atを使って1ずつ受け取る

vector<int> vec(N);

for (int i = 0; i < N; i++) {

cin >> vec.at(i);

}

・string型は1行文を格納するので、配列を使えば2次元的に入力を扱える

・要素を増やしたいときは「配列変数.push\_back(x)」で後ろにxを追加できる

・要素を消したいときは「配列変数.pop\_back()」で末尾の要素を削除できる

・ほかの配列の書き方

→int data[3]; // Cの配列

→array<int, 3> data; // arrayによる配列

・ｘ以上の最小の要素を求める

→まずsortして、\*lower\_boundをする

「値」以上の最小の値

\*lower\_bound (配列.begin(), 配列.end(), 値)

例）12以上の最小の値は？

vector<int> a = {0,10,13,14,20};

sort(a.begin(),a.end());

cout << lower\_bound(a.begin(),a.end(), 12)

→13

・ｘより大きい最小の要素を求める

→\*upper\_boundを使う(他は上と同様)

・配列同士が同じかどうかの判定はpythonと同じように変数そのままでかける

vector a = {0,1,2}, b = {4,5,6};

if(a != b) cout << “different” << endl;

**○多次元配列**

・2次元配列は2次元の表を扱うときに便利

・基本的に最初が縦

vector<vector<型>> 変数名(縦の要素数, vector<型>(横の要素数, 初期値))

・初期値は省略可能

・変数名.at(i).at(j)でi行目j列目へアクセスできる

・変数名.size()で縦の大きさを取得できる

・変数名.at(0).size()で横の大きさを取得できる

例）2次元配列の初期化

vector<vector<int>> data = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12},

}；

・入力には多重ループを使う

例）縦3＊横4の表の入力

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 4; j++) {

cin >> data.at(i).at(j);

}

}

**○関数の呼び出し**

・関数に代入する値を**引数**という

・関数の計算結果を**返り値**、または**戻り値**という

| **関数** | **min(a, b)** | **max(a, b)** | **swap(a, b)** |
| --- | --- | --- | --- |
| 機能 | aとbのうち小さい方の値を返す | aとbのうち大きい方の値を返す | 変数aと変数bの値を交換する |

| **関数** | **sort(vec.begin(), vec.end())** | **reverse(vec.begin(), vec.end())** |
| --- | --- | --- |
| 機能 | 配列変数vecをソートする（要素を小さい順に並び替える） | 配列変数vecの要素の並びを逆にする |

・min関数、max関数

int a = 10, b = 5;

int ans = min(a, b); // min関数

cout << ans << endl;

→5

・swap関数（交換）

int a = 10, b = 5;

swap(a, b);

cout << a << endl; // 5

cout << b << endl; // 10

・sort関数（配列を小さい順に並べる）

vector<int> vec = {2, 5, 2, 1};

sort(vec.begin(), vec.end()); // {1, 2, 2, 5}

・reverse関数（配列を逆に並べる）

vector<int> vec = {1, 5, 3};

reverse(vec.begin(), vec.end()); // {3, 5, 1}

・sort関数を使ってからreverse関数を使うことで大きい順に並べられる

**○関数の呼び出し2**

【map】

・mapを用いると「特定の値(key)に、ある値(value)が紐付いている」ようなデータを扱える

・「変数[key]++」とすると、値が登録されていないときには初期化して1にしてくれて、登録済みならインクリメントするのでめちゃ便利！

・map<keyの型, valueの型> 名前;

例）テスト

map<string,int> score;

| **操作** | **記法** | **例** |
| --- | --- | --- |
| 値の追加 | 変数[key] = value; | score[“yuton”] = 100; score[“ryon”] = 90; |
| 値の削除 | 変数.erase(key); | score.erase(“ryon”); |
| 値へのアクセス | 変数.at(key)または変数[key] | score.at(“yuton”); // 100 |
| 所属判定 | 変数.count(key) |  |
| 要素数の取得 | 変数.size() | score.size(); // 1 |

・所属判定「変数.count(key)」で行い、すでに所属しているならtrueを返す

→同じものが何個あるかを数えられる

・所属判定はif文とともに使う

if(score.at(“yuton”)){cout << score.at(“yuton”) << endl;}

else{}

・mapの中でループさせたいときにはkey,valueをpairにする。順番はkeyが小さい順となり、keyのアクセスには「p.first」、valueのアクセスには「p.second」を使う。

for (auto p : score){

string key = p.first;

int value = p.second;

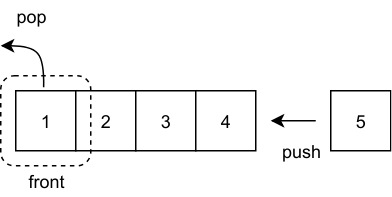
// keyとvalueに置き換えたのでそれらを使い、処理を書く

}

【queue】

・「値を1つずつ追加し、追加した順で値を取り出す」ような処理を行うデータ構造をキューや待ち行列という（名前の通り、先頭の人は一人ずつ消えていく）

・イメージ↓



| **操作** | **記法** |
| --- | --- |
| 要素の追加 | 変数.push(値); |
| 先頭の要素へのアクセス | 変数.front() |
| 先頭の要素を削除 | 変数.pop(); |
| 要素数の取得 | 変数.size() |
| queueが空かの判別 | 変数.empty() |

・queueの宣言→queue<型> 名前;

・queueが空かの判別で空ならtrueを返す

例）queueに全部追加→１つずつ出力→出力したら消去→空になったらおわり

queue<int> q;

q.push(10);

q.push(3);

q.push(6);

q.push(1);

while (q.empty() = false) {

cout << q.front() << endl; // 先頭の値を出力

q.pop(); // 先頭の値を削除

}

【priority\_queue】

・「それまでに追加した要素のうち、最も大きいものを取り出す」という処理を行うときには、優先度付きキューというデータ構造を使う

・最大の値を取り出すには計算量O(N)かかるが、priority\_queueだとO(logN)ですむ！

・topにアクセスはできるけどそのまま操作することはできない(read only)から、いったん違う変数にしてpopして、操作後の変数をpushするとよい

例）topを半分にする操作

int v = q.top(); q.pop(); // popを忘れやすいから1行で書いてしまうと良い

v /= 2;

q.push(v);

・queueの使い方とほぼ変わらないが、アクセスと消去が先頭のものではなく、最大のものになる

| **操作** | **記法** |
| --- | --- |
| 要素の追加 | 変数.push(値); |
| 最大の要素へのアクセス | 変数.top() |
| 最大の要素を削除 | 変数.pop(); |
| 要素数の取得 | 変数.size() |
| queueが空かの判別 | 変数.empty() |

【deque】

・先頭への値の追加や削除をしたいときに有用。計算量O(1)ですむ

例）dequeの使い方

deque<int> que; // 宣言

que.push\_back(2); // 末尾に値を追加

que.push\_front(1); // 先頭に値を追加

que.front(); // 先頭にアクセス

que.back(); // 末尾にアクセス

que[i]; // i番目にアクセス

que.pop\_front(); // 先頭を削除

que.pop\_back(); // 末尾を削除

que.empty(); // 中身が空なら1(true)を返す

【他】

・他にもいろいろあるから、適宜「[STLのコンテナ](https://atcoder.jp/contests/apg4b/tasks/APG4b_aa)」を参照

**○関数の作成**

・自分で関数を作るときはmain関数の前に作る

・返り値の型 関数名(引数1の型 引数1の名前, 引数2の型 引数2の名前, ...) {処理}

・返り値の指定はreturn文で行う「return 返り値;」

・return文までいくと関数の処理は終了する

・関数を呼び出すときは関数名(引数１,引数２,・・・)

・返り値のある関数でreturn文を忘れると適当な値が返る

・例）min関数を自作

int min(int x, int y) {

if (x < y) {return x;}

else {return y;}

}

int main() {

int answer = min(10, 5);

cout << answer << endl; // 5

}

・返り値のない関数は最初（返り値の型）をvoid型にする

例）合計点を出力する関数

void sum(int a, int b, int c){//返り値なしでa,b,cを引数とする関数を定義

int goukei = a+b+c;

cout << goukei << endl;

return;//返り値なし

}

int main(){

int x,y,z;

cin >> x >> y >> z;

sum(x,y,z);//sum関数にx,y,zを代入して使う

}

・引数の型、引数の数が異なる場合は同じ名前の関数を定義できる

・配列を関数の中に組み込むときは、配列の方にのっとって書く

・配列を返り値にするとき

vector<int>input(int N){//引数N、返り値が配列で「input」という名の関数

vector<int>vec(N);

rep(i,N){

cin >> vec.at(i);

}

return vec;//配列を返す

}

関数を呼び出すときはvector<int> A = input(N);//引数Nの配列Aを作る

・配列を引数にするとき

int sum(vector<int> score){//引数が配列score、返り値が整数で「sum」という名の関数

int s =0;

rep(i,score.size()){

s += score.at(i);

}

return s;//sを返す

}

関数を呼び出すときはsum(配列名)

**○再帰関数(何も理解してない)**

・関数の中で同じ関数を呼び出すことを再帰関数という

・ベースステップ：再帰呼び出しをしないで完了する処理（if文で条件分岐）

・再起ステップ：再帰呼出しを行い、その結果を用いて行う処理

・連鎖に終わりがないと無限ループになってしまう（必ずベースケースに到達させる）

・再帰関数の書き方

1,「引数」「返り値」「処理内容」を決める

2，再帰ステップの実装

3，ベースケースの実装

例）1からｎまでの総和を求める

1，引数int n, 返り値 総和, 処理内容 総和を計算する

int sum (int n) {}

2，再起ステップ：「ｎまでの総和」＝「ｎ－1までの総和」＋1

int s = sum (n-1) + 1

return s + n ;

3,ベースケース：ｎが0のときの総和は0なので再帰呼び出しの必要なし

if (n == 0) return 0;

最終的にできた再帰関数

int sum(int n) {

if (n == 0) return 0; //ベースケース

int s = sum(n - 1); //再起ステップ

return s + n;

}

**○参照(なんかよくわからない･･･)**

・参照を使うことで無駄なコピーを避け、実行時間が大幅に減る

・参照とは既存の変数の別名のことで、参照の値を変えると元々の変数も変わる

・元々の変数のことを参照先という

・参照先の型 &参照の名前 = 参照先;

int a = 0;

int &b = a;

b = 1;// a = 1となる

・関数の引数での参照に本領を発揮する（参照渡し）

→無駄なコピーがなくなり、実行時間が大幅に減るらしい

・返り値の型　変数名(引数1の型　&引数1,　引数2の型　＆引数2,　･･･)

int min (int &x, int &y){}

・配列を引数,返り値なしとする関数の参照

void sum (vector<int> &vec ){}

**○ビット演算**

・ビットとは2進数のこと。全探索ができるようになる

・ビットは集合を扱うときに使う

・ビットの位置は配列と逆で右から0，1，2・・・と続く

・ビットを使うときはbitset<ビット数> 変数名;　「変数名（ビット列）」を加えてもよい

・2つの集合に共通している要素を取り出す→AND演算

・2つの集合のうち少なくとも一方に存在する要素を取り出す→OR演算

・ある集合に含まれない要素取り出す→NOT演算

・シフト→左右にビットをずらす。はみ出たビットは捨てられ、足りないとこは0になる

・ビット演算子の末尾に「＝」をつけないと代入にはならない

・ビット演算子は優先順位が低いから（）で囲め！

| **ビット演算** | **bitsetの演算子** | **使い方** | **意味** |
| --- | --- | --- | --- |
| AND演算 | & | 変数1 & 変数2 | 各ビットについて、両方1なら1 |
| OR演算 | | | 変数1 | 変数2 | 少なくとも一方が1なら1 |
| XOR演算 | ^ | 変数1 ^ 変数2 | どちらか一方のみ1なら1 |
| NOT演算 | ~ | ~変数 | ビットを反転させる |
| 左シフト | << | 変数 << 数 | 指定した分左にずらす |
| 右シフト | >> | 変数 >> 数 | 指定した分右にずらす |

・ビットの値を変更→変数名.set(位置,値);

・ビットの値が1か調べる（if文でよく使う）→変数名.test(位置);

→ビットの値が1ならtrue、0ならfalseを返す

例）if(b.test(5)){cout << “5番目は1です” << endl;}

| **操作** | **書き方** |
| --- | --- |
| 全てのビットを1に | 変数.set(); |
| 特定のビットを1に | 変数.set(1にする位置); |
| 特定のビットを変更 | 変数.set(位置, 値); |
| 全てのビットを0にする | 変数.reset(); |
| 特定のビットを0にする | 変数.reset(0にする位置); |
| 全てのビットを反転する | 変数.flip(); |
| 特定のビットを反転する | 変数.flip(反転する位置); |
| 特定のビットが1になっているかを調べる | 変数.test(調べる位置); |
| 全てのビットが1になっているかを判定する | 変数.all() |
| いずれかのビットが1になっているかを判定する | 変数.any() |
| 1のビットの個数を数える | 変数.count() |

**○ビット全探索**

000,001,010,011,100,101,110,111

・ビット全探索を使うと組み合わせをすべて列挙できる

・bitsetの長さには変数は使えない！！

・「1 << N」は2＾Nを表す。for文の中によく使われる

・下記のビット全探索のフォーマット通りに書く！！bitsetをfor文の中にいれ初期化する

for (int tmp = 0; tmp < (1 << ビット数); tmp++) {

bitset<ビット数> s(tmp);

処理

}

例）3ビットをすべて列挙

for (int tmp = 0; tmp < (1 << 3); tmp++){

bitset<3> b (tmp);

cout << b << endl;

}

・bitsetを使わないビット全探索の仕方

rep(is, 1<<h)rep(js, 1<<w){ // 2^h回×2^w回まわす（2進数）

int black=0;

rep(i,h)rep(j,w){

if(is>>i&1)continue; // isをi個右にずらし、一番右が1かどうかを判定

if(js&(1<<j))continue; // jsの右からｊ番目が1かをどうかを判定

if(c[i][j] == '#')black++;

}

if(black==k)ans++;

}

is, js自体は10進数だけど、is>>i&1の時点で2進数に変換される

if分の中の判定の「is>>i&1」は「is&(1<<i)」ともかける