Contents

1 基本

1.1 複雜度

複雜度是定性描述該演算法執行成本（時間/空間）函式，用來分析資料結構和演算法（DSA)。

1.1.1 常用函数

Big O 用來表示一個複雜度的上界，定義為 $f(n) \in O(g(n)) \quad iff \quad \exists\ c,\ N \in R^+,\ \forall n \geq N$ 有 $|f(n)| \leq |cg(n)|$，例如 $f(n) = 5n^2 + 4n + 1$，我們會注重最高項 $5n^2$，且我們會 5 是常數，得出 $f(n) \in O(n^2)$

Big Ω 用來表示一個複雜度的下界，對於任意的 $f(n) \in O(g(n))$，都有 $g(n) \in \Omega(f(n))$。

Big Θ 要同時滿足 Big O 和 Big Ω

Big O 是我們比較常用的，其他兩個可能再一些地方會用到

1.1.2 常見複雜度

$O(1) < O(log\ n) < O(n) < O(n\ log\ n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$ 另外還有一個在並查集常見，即 $O(\alpha(n))$，近似於 $O(1)$，可直接當作 $O(1)$

1.1.3 時間/空間複雜度

時間複雜度，和運算有關，*/% 會比 +-還要久，而複雜度得項次會跟迴圈有關，初階競賽只會在意你的項次，只要不要太大基本都會過，進階些比賽，有可能出現常數過大，導致複雜度合理卻還是吃 TLE 的情況，這時候需要利用”壓常數”技巧，降低時間，讓程式 AC。
空間複雜度，則是跟你宣告的變數記憶體總和有關，比時間複雜度容易估計，在樹狀的資料結構，往往需要搭配動態記憶體，才不會因為開太多空間而吃了 MLE。
題外話，如果你在你的 array 不是開在全域內，開了 10 的 5,6 次，在執行時跑出 RE，那你有以下兩種解決方式

1. 把 array 移至全域
2. 加上 static，表示靜態變數

```
1 static int a[100000];
```

1.2 函式

函式為程式裡的運算單元，可以接受資料，並回傳指定值。main 是 C/C++ 程式的入口函式，接受命令列的參數，正常情況會回傳 0 代表正常運作。
以下為其語法

```
1 return_type function_name(parameter list){
2     Do something...
3     return data;// void need not return;
4 }
```

範例

```
1 int sum(int x, int y){
2     int sum=0;
3     sum+=x;
4     sum+=y;
5     reutnr x+y;
6 }
```

函式有個特性為自呼叫，也就是自己的區域可以呼叫自己，但要有終止條件，不然會陷入無限遞迴，同時也要避免遞迴過深，造成 stack overflow。

```
1  int ans;
2  void f(int i){
3      if(i==1){
4          ans=1;
5          return;
6      }
7      f(i-1);
8      ans*=i;
9      return;
10 }
```

函式有很多用處，一個為模組化，意即相同的部分 (最多只差一些參數)，寫成一個函式，除了簡潔，在除錯也比較方便。一個是利用自呼叫特性實作遞迴，遞迴可將問題拆解成同類的子問題而解決問題。
常見遞迴使用

1. 分治
2. dp 中的 top-down
3. 圖/樹的搜索

1.3 指標

指標是紀錄記憶體的位址的變數，不管是基礎型態或自定義型態皆可用指標，指標的可以讓你直接對記憶體操作。而指標對學習者是一到難度高的門檻，但在程式競賽中，是不可或缺的。
指標在程競中會用到的地方是”動態記憶體配置”，這在比較進階的資料型態會比較常出現。

```
int *p=new int;
(*p)=1;
delete p;
```

```
int a=5;
int *p=&a;
(*p)++;
cout<<a<<'\n';
```

1.4 參考

參考型態代表一個變數的別名，可直接取得變數的位址，並間接透過參考型態別名來操作物件，作用類似於指標，但卻不必使用指標語法，也就是不必使‘*‘運算子來提取值。

```
const int N=100;
int a[N][N];
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        int &x=a[i][j];
        x=i+j;
    }
}
```

參考型態可用在取代太長的變數（如：‘a[x][y][z]‘），容易維護。另一個是當函式要傳入可修改的值，可取代指標。

1.5 傳值

函式傳入的參數，可以是一般、指標或是參考型態，以下以 Swap 來介紹

1.5.1 call by value

傳入的變數為一般型態，會”複製”一份到函式，原本的變數不會有任何改變。

```
void swap(int x,int y){
    cout<<x<<' '<<y<<'\n';// 1 2
    int t=x;
    x=y;
    y=t;
    cout<<x<<' '<<y<<'\n';// 2 1
}
int main(){
    int a=1,b=2;
    cout<<a<<' '<<b<<'\n';// 1 2
    swap(a,b);
    cout<<a<<' '<<b<<'\n';// 1 2
}
```

1.5.2 call by address/value of pointer

傳入的變數為指標型態，函式內的變數改變，是對記憶體操作，所以原本的數字也會跟著改變。

```
void swap(int *x,int *y){
    cout<<*x<<' '<<*y<<'\n';// 1 2
    int t=*x;
    *x=*y;
    *y=t;
```

```
    cout<<*x<<' '<<*y<<'\n';// 2 1
}
int main(){
    int a=1,b=2;
    cout<<a<<' '<<b<<'\n';// 1 2
    swap(&a,&b);
    cout<<a<<' '<<b<<'\n';// 2 1
}
```

1.5.3 call by reference

傳入的變數為參考型態，函數內的變數是原本變數的分身，所以函數內變數改變時，原本變數也會跟者改變。

```
void swap(int &x,int &y){
    cout<<x<<' '<<y<<'\n';// 1 2
    int t=x;
    x=y;
    y=t;
    cout<<x<<' '<<y<<'\n';// 2 1
}
int main(){
    int a=1,b=2;
    cout<<a<<' '<<b<<'\n';// 1 2
    swap(a,b);
    cout<<a<<' '<<b<<'\n';// 2 1
}
```

1.6 struct

structc 是讓 coder 能將原本獨立的資料包在一起。例如：三維空間由 x 座標、y 座標、z 座標組成。
語法：

1. 型態 (type) 可以是一般或是指標型態
2. 也可以寫函式或重載運算子

```
struct struct_name{
    type1 name1;
    type2 name2;
    ...
}; //<-notice
```

以下的例子為平面上的點。

```
struct Plane{
    int x,y;
    Plane(){};
    Plane(int _x,int _y):x(_x),y(_y){}
    Plane add(const Plane &rhs)const{
        return Plane(x+rhs.x,y+rhs.y);
    }
    bool operator<(const Plane &rhs)const{
        if(x!=rhs.x)return x<rhs.x;
        return y<rhs.y;
    }
    ~Plane(){};
}
int main(){
    Plane p1;
    Plane p2(1,2);
    Plane p3=Plane();
    Plane p4=Plane(0,0);
}
```

1.6.1 建構子 (constructor)、解構子 (destructor)

建構子和 strcut name 同名，是用來初始化 struct 裡的資料，如果不寫的話，會有預設建構子，裡面的資料都是亂數。根據請況可多載，然而，如果你寫了運算子，一定要寫一個不帶任何參數的運算子，否則的話，像第 14 行這樣只有宣告，沒加其他東西的的程式碼就不會通過。
解構子的名字形式為 $\sim strcutname$，是在變數離開作用域時運作，不寫的話也是會有預設解構子，在程式比賽中這樣就已足夠。

1.6.2 重載運算子

c++ 原有的型態都根據需要，定義了各種運算子，但 struct T如果有需要的話，須自己定義。而在競賽中，常需要作排序而需要小於運算子 ('<')。

1.7 algorithm

1.7.1 sort

這個函式傳入兩個變數，代表容器 (array 或是 vector) 的頭尾 $[a,b)$，這裡的 b 不會排序，用來指示為結尾，例如要排序 a 陣列的第 0 到第 5 個元素。

```
1 sort(a,a+6);
```

此函數的複雜度圍 $O(n\log n)$，n 為排序的個數

1.7.2 min/max

min 和 max 原本在 c 定義在 math.h 內，c++ 將它移入 algorithm 中

```
1 cout<<min(5,2)<<'\n';// 2
2 cout<<max(5,2)<<'\n';// 5
```

1.7.3 lower_bound/upper_bound

這兩個函式會在" 有序序列" 中尋找值，前兩個值放的是容器 (array 或是 vector) 的頭尾 [a,b)，第三的是比較的值 val。

```
1 lower_bound(a, a+n, k); //最左邊 ≥ k 的位置
2 upper_bound(a, a+n, k); //最左邊 > k 的位置
3 upper_bound(a, a+n, k) - 1; //最右邊 ≤ k 的位置
4 lower_bound(a, a+n, k) - 1; //最右邊 < k 的位置
5 [lower_bound, upper_bound) //等於 k 的範圍
6 equal_range(a, a+n, k);
```

如果要將位置傳換成數字，直接減起始位置就可，下面是一個範例 (from cplusplus)：

```
1  #include <iostream>     // std::cout
2  #include <algorithm>    // std::lower_bound, std::upper_bound, std::sort
3  #include <vector>       // std::vector
4
5  int main () {
6    int myints[] = {10,20,30,30,20,10,10,20};
7    std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
8
9    std::sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30
10
11   std::vector<int>::iterator low,up;
12   low=std::lower_bound (v.begin(), v.end(), 20); //          ^
13   up= std::upper_bound (v.begin(), v.end(), 20); //                    ^
14
15   std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
16   std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
17
18   return 0;
19 }
20 /*
21 lower_bound at position 3
22 upper_bound at position 6
23 */
```

1.7.4 next_permutation/prev_permutation

這兩個函式會幫你的陣列轉為後/前一個字典序，如果沒有後/前一個字典序，這個函式會回傳 false，也不會有任何改變，以下為範例 (from cplus-plus)：

```cpp
// 字典序wiki https://zh.wikipedia.org/wiki/%E5%AD%97%E5%85%B8%E5%BA%8F
#include <iostream>     // std::cout
#include <algorithm>    // std::next_permutation, std::sort

int main () {
  int myints[] = {1,2,3};

  std::sort (myints,myints+3);

  std::cout << "The 3! possible permutations with 3 elements:\n";
  do {
    std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
  } while ( std::next_permutation(myints,myints+3) );

  std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n'
    ;

  return 0;
}
/*
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
After loop: 1 2 3
*/
```

1.8 cmath

1.8.1 atan/atan2

atan/atan2 函數是將斜率轉為弧度，如果要在轉為角度就以 180 度除以 PI 就好，而 atan 直接傳入斜率，atan2 則是座標，atan2 可以處理 x=0 的狀況，比 atan 好用。

```cpp
int main () {
  double PI=acos(-1.0);
  cout<<PI<<'\n'; // 3.14159
  cout<<atan(1)*180/PI<<'\n'; // 45
  cout<<atan(-1)*180/PI<<'\n'; // -45
  cout<<atan2(1,1)*180/PI<<'\n'; // 45
  cout<<atan2(-1,1)*180/PI<<'\n'; // -45
  return 0;
}
```

1.8.2 log/log2/log10

這些都是常用對數函數分別以 e,2,10 為底

```cpp
int main(){
  cout<<log(10)<<'\n'; // 2.30259
  cout<<log2(10)<<'\n'; // 3.32193
  cout<<log10(10)<<'\n'; // 1
  cout<<log(8)/log(2)<<'\n'; // 3
}
```

1.8.3 pow

此函數會回傳以 base 為底的 exponent 次方，若 $\geq 10^6$，就會輸出科學記號。

```cpp
int main(){
  cout<<pow(10,5)<<'\n'; // 100000
  cout<<pow(10,6)<<'\n'; // 1e+06
}
```

1.8.4 sqrt

此函數會回傳 x 的根號次方

```cpp
int main(){
  cout<<sqrt(1)<<'\n'; // 1
  cout<<sqrt(2)<<'\n'; // 1.41421
  cout<<sqrt(3.0)<<'\n'; // 1.73205
}
```

1.9 iomanip

1.9.1 setw

這個函式會將傳入的整數設定輸出寬度後輸出。

```cpp
int main(){
  cout<<setw(5)<<100<<'\n';  //"  100"
  cout<<setw(5)<<1234<<'\n'; //" 1234"
  cout<<setw(2)<<100<<'\n';  //  "100"
  return 0;
}
```

1.9.2 setprecision

這個函式設定輸出到小數點後幾位。

```cpp
int main () {
  double f =3.14159;
  cout << setprecision(5) << f << '\n';// 3.1416
  cout << setprecision(9) << f << '\n';// 3.14159
  cout << fixed;
  cout << setprecision(5) << f << '\n';// 3.14159
  cout << setprecision(9) << f << '\n';// 3.141590000
  return 0;
}
```

1.10 extra syntax

1.10.1 break, continue, return

1. break：跳出迴圈
2. continue：這輪不做，到下一輪
3. return：跳出函式，並回傳值

在 break, continue, return 後的 else 是無用的，因為如果這三種指令執行了，後面的東西就根本不會執行到。

1.10.2 const

const 用途在於宣告這個變數式不能更動的，這適合用來宣告常數。

```
1  const int N=5;
2  int main(){
3      cout<<N<<'\n'; // 5
4      N++; // error
5  }
```

1.10.3 static

如果一個變數被宣告為 static，那麼他只會被宣告一次，直到整個程式結束才被刪除。

```
1  #include <iostream>
2  using namespace std;
3
4  void count();
5
6  int main() {
7
8      for(int i = 0; i < 10; i++)
9          count();
10
11     return 0;
12 }
13
14 void count() {
15     static int c = 1;
16     cout << c << endl;
17     c++;
18 }
19 /*
20 1
21 2
22 3
23 4
24 5
25 6
26 7
27 8
28 9
29 10
30  */
```

1.10.4 define

不知道各位有沒有用過 excel 中的巨集，他可以幫你做重複性高的動作，C++ 中的 define 可以幫你做類似的事。

```
1  #define one 1
2  #define ten(x) 10*x
3  #define sum(x,y) 2*x+y
4
5  int main(){
6      cout<<1<<'\n';
7      cout<<ten(5)<<'\n';
8      cout<<sum(1,1)<<'\n';
9      cout<<sum(1,1)*ten(2)<<'\n';
10 }
```

由上面範例可見，define 可以取代程式中出現的特定字元，還可以帶參數，為了要讓使用 define 後的結果是正確的，請將取代後的字元括號起來，否則會輸出非預期的結果如上面範例第 9 行

```cpp
#define one 1
#define ten(x) 10*x
#define sum(x,y) 2*x+y

int main(){
    cout<<1<<'\n';
    cout<<ten(5)<<'\n';
    cout<<sum(1,1)<<'\n';
    cout<<sum(1,1)*ten(2)<<'\n';
}
```

1.10.5 typedef

typedef 可以為型態取別名，在之後用到要宣告該型態的時候，可以打該型態之別名，減省時間。C++11 開始可以用 using 來達到相同的事。

```cpp
typedef long long LL;
using LL = long long; //C++11
```

1.10.6 auto

C++11 開始，新增了一個關鍵字叫 auto，auto 可以自動判別變數型態，但必須給他初始值，否則他無法判別型態，C++14 開始，可用在 function 回傳值

```cpp
auto x;//error
auto y=1;
auto f(){return 1;}//from c++14
```

1.10.7 range_based for

C++11 開始有另外一種 for 是 range_based，他只要給兩個參數，一個變數指定資料型態並提供遍歷，另一個為要遍歷的範圍，例子如下。

```cpp
int sum = 0;
int arr[5] = {1,2,3,4,5};
for(int i=0;i<5;i++){
    int x = a[i];
    sum += x;
}
for(int x:arr){//can use auto x:arr
    sum += x;
}
```

1.11 lambda

方便地定義匿名函式

1.11.1 lambda-introducer

也叫 Capture clause，宣告外部變數 (在可視範圍 (scope) 內) 傳入此函式內的方法。

1. ‘[]‘: 只有兩個中括號，完全不抓取外部的變數。
2. ‘[=]‘：所有的變數都以傳值 (call by value) 的方式抓取
3. ‘[&]‘：所有的變數都以傳參考 (call by reference) 的方式抓取
4. ‘[x,&y]‘：x 變數使用傳值，y 變數使用傳參考
5. ‘[=,&y]‘：除了 y 變數使用傳參考以外。其餘的變數皆使用傳值的方式
6. ‘[&,x]‘：除了 x 變數使用傳值以外，其餘的變數皆使用傳參考的方式

1.11.2 lambda declarator

也叫參數清單，傳入此函式對應資料。

1.11.3 mutable specification

指定以傳值方式抓取進來的外部變數，如果用不到可省略。與一般函數的傳入參數之異

1. 不可指定參數的預設值。
2. 不可使用可變長度的參數列表。
3. 參數列表不可以包含沒有命名的參數。

1.11.4 例外狀況規格

指定該函示會丟出的例外，其使用的方法跟一般函數的例外指定方式一樣，如果用不到可省略。

1.11.5 傳回值型別

指定 lambda expression 傳回型別，如果 lambda expression 所定義的函數很單純，只有包含一個傳回陳述式（statement）或是根本沒有傳回值的話，可省略 (optional)

1.11.6 compound-statement

亦稱為 Lambda 主體 (lambda body)，跟一般的函數內容一樣。

最後來看個 lambda 範例，結束這一章節。

```
std::sort(s.begin(), s.end(), [](int a, int b) {
    return a > b;
});
```

```
std::vector<int> someList;
int total = 0;
std::for_each(someList.begin(), someList.end(),[&total](int x) {
    total += x;
});
std::cout << total;
```

2 基礎資料結構

2.1 什麼是 STL?

標準函式庫 (Standard Template Libiary)，C++ 內建的資料結構。

2.2 型態模板

當你要使用容器時，你必須要告訴 C++ 說，你的資料型態是什麼，型態模板的用途就是在於此。
用法：$C < T > name$
而容器內部東西不會只有一個，像 map 就需要兩種型態。
$map < T1, T2 > name$
有時候參數不須寫滿，不寫滿的地方的值為預設值。

2.3 迭代器

如果你想在容器中遍歷，你可能想用下標運算子 ‘[]‘，但不是所有容器都像陣列，都有支援下標運算子，所以 C++ 為每個容器都提供一個資料型態叫”迭代器”，你可以把迭代器當成一種指標，假設有一個迭代器 it，加上星號 ‘*‘可以存取 IT 所指向的內容，依據迭代器的強到弱可分為三種：

1. 隨機存取 (Random Access)：可與整數做 +-法、遞增及遞減
2. 雙向（Bidirectional）迭代器：遞增及遞減
3. 單向（Forward）迭代器：只能遞增

根據用法可分為兩種：

1. 輸入（Input）迭代器：讀取迭代器指向的內容，，所有的迭代器都可以當作輸入迭代器。
2. 輸出（Output）迭代器：更改迭代器指向的內容時，除了常數（const）迭代器（也就是規定不能更動迭代器指向的內容）以外，所有的迭代器都可以當作輸出迭代器。

C++ 在許多容器中提供正向和逆向迭代器，前者由前往後，後著由後往前，宣告時分別為 $C::iterator$ 及 $C::reverse\_iterator$，每種迭代器分別有一對迭代器代表頭尾，如下表，注意 end 系列指向該容器最後一項的後一項，不要對他做人和取值或修改。

	正向	逆向
可改值	C.begin() C.end()	C.rbegin() C.rend()
不可改值	C.cbegin() C.cend()	C.crbegin() C.crend()

2.4 stack 堆疊

有兩個端口，其中一個封閉，另一個端口負責插入、刪除的資料結構

```
struct stack{
    int st[N],top;
    Stack():top(0){}
    int size(){
        return top;
    }
    void push(int x){
        st[++top]=x;
    }
    int top(){
        assert(top>0)
        return st[top];
    }
    void pop(){
        if(top)--top;
    }
}
```

1. 標頭檔：<stack>
2. 建構式：stack <T> s
3. s.push(T a)：插入頂端元素，複雜度 $O(1)$
4. s.pop()：刪除頂端元素，複雜度 $O(1)$
5. s.top()：回傳頂端元素，複雜度 $O(1)$
6. s.size()：回傳元素個數，複雜度 $O(1)$
7. s.empty()：回傳是否為空，複雜度 $O(1)$

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

int main(){
    stack<int>st;
    st.push(1);
```

```
    cout<<st.top()<<'\n';// 1
    st.push(2);
    cout<<st.top()<<'\n';// 2
    st.push(3);
    cout<<st.top()<<'\n';// 3
    st.pop();
    cout<<st.top()<<'\n';// 2
}
```

2.5 queue 佇列

有兩個端口，一個負責插入，另一個端口負責刪除的資料結構

```
struct Queue{
    int q[N],head,tail;
    Queue():head(0),tail(0){}
    int size(){
        return tail-head;
    }
    void push(int x){
        q[tail++]=x;
    }
    int front(){
        return q[head];
    }
    void pop(){
        head++;
    }
}
```

1. 標頭檔：<queue>
2. 建構式：queue <T> q
3. q.push(T a)：插入尾端元素，複雜度 $O(1)$
4. q.pop()：刪除頂端元素，複雜度 $O(1)$
5. q.front()：回傳頂端元素，複雜度 $O(1)$
6. q.size()：回傳元素個數，複雜度 $O(1)$
7. q.empty()：回傳是否為空，複雜度 $O(1)$

```
#include <iostream>
#include <queue>
using namespace std;

int main(){
    queue<int>st;
    st.push(1);
    cout<<st.front()<<'\n';// 1
    st.push(2);
    cout<<st.front()<<'\n';// 1
    st.push(3);
    cout<<st.front()<<'\n';// 1
    st.pop();
    cout<<st.front()<<'\n';// 2
}
```

2.6 deque 雙向佇列

有兩個端口，皆負責刪除、插入的資料結構

1. 標頭檔：<deque>
2. 建構式：deque <T> dq
3. dq.push_front(T a),dq.push_back(T a)：插入頂端/尾端元素，複雜度 $O(1)$
4. dq.pop_front(),dq.pop_back()：刪除頂端/尾端元素，複雜度 $O(1)$
5. dq.front(),dq.back()：回傳頂端/尾端元素，複雜度 $O(1)$
6. dq.size()：回傳元素個數，複雜度 $O(1)$
7. dq.empty()：回傳是否為空，複雜度 $O(1)$

2.7 list

陣列如果要從中間插入一個元素，需要將其後面所有元素搬移一格，需耗費 $O(n)$，連結串列 (linklist) 能只花 $O(1)$ 完成插入。

```
struct Node{
    int v;
    Node *next=nullptr;
}
```

```
struct Node{
    int v;
    Node *next=nullptr,*prev=nullptr;
}
```

但有時候我們也可以利用 idnex 取代指標來實作 linklist，兩種做法各有自己的優缺點。
C++ 提供 list 函式庫實作雙向串列

1. 標頭檔：‘<list>‘
2. 建構式：‘list <T> L‘
3. L.size()：回傳元素個數，複雜度 $O(1)$
4. L.empty()：回傳是否為空，複雜度 $O(1)$
5. L.push_front(T a),L.push_back(T a)：插入頂端/尾端元素，複雜度 $O(1)$
6. L.pop_front(),L.pop_back()：刪除頂端/尾端元素，複雜度 $O(1)$
7. L.insert(iterator it,size_type n,T a): 在 it 指的那項的前面插入 n 個 a 並回傳指向 a 的迭代器。複雜度 $O(n)$。
8. L.erase(iterator first,iterator last): 把 $[first,last)$ 指到的東西全部刪掉，回傳 last。複雜度與砍掉的數量呈線性關係，如果沒有指定 last，那會自動視為只刪除 first 那項。
9. L.splice(iterator it,list& x,iterator first,iterator last):first 和 last 是 x 的迭代器。此函式會把 $[first,last)$ 指到的東西從 x 中剪下並加到 it 所指的那項的前面。x 會因為這項函式而改變。若未指定 last，那只會將 first 所指的東西移到 it 前方。複雜度與轉移個數呈線性關係。

```
// adapt from cppreference
#include <iostream>
#include <list>

int main(){
    std::list<char> letters {'o', 'm', 'g', 'w', 't', 'f'};

    if (!letters.empty()) {
        cout << letters.front() << '\n';// o
        cout << letters.back() << '\n'; // f
    }
}
```

2.8 array

更安全方便的陣列

1. 標頭檔：<array>
2. 建構式：array <T,N> a
3. a.size()：回傳元素個數，複雜度 $O(1)$
4. a.fill(T val)：將每個元素皆設為 val，複雜度 $O(size)$

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

int main(){
    array<int,5>a;

    cout<<a.size()<<'\n';

    for(int i=0;i<a.size();i++){
        a[i]=i*i;
    }

    for(int i=0;i<a.size();i++){
        cout<<a[i]<<' ';
    }
    cout<<'\n';
}
/*
5
0 1 4 9 16
*/
```

2.9 vector

動態陣列，可改變長度

1. 標頭檔：<vector>
2. 建構式：vector <T> v
3. v[i]：回傳 v 的第 i 個元素，複雜度 $O(1)$
4. v.push_back(T a)：插入尾端元素，複雜度 $O(1)$
5. v.size()：回傳元素個數，複雜度 $O(1)$
6. v.resize()：重設長度，複雜度 $O(1)$
7. v.clear()：清除元素，複雜度 $O(size)$
8. 兩特化
    (a) vector<char>：string
    (b) vector<bool>：bitset

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

int main(){
    vector<int>v;
    v.push_back(1);
```

```
    v.push_back(2);
    v.push_back(3);
    for(int i=0;i<v.size();i++){
        cout<<v[i]<<' ';
    }
    cout<<'\n';
    v.clear();

    v.resize(2);
    cout<<v.size()<<'\n';

    for(int i=0;i<v.size();i++){
    v[i]=5-i;
    }
    for(int i=0;i<v.size();i++){
    cout<<v[i]<<' ';
    }
    cout<<'\n';
}
/*
1 2 3
2
5 4
*/
```

2.10 string

可變動長度的字元陣列

1. 標頭檔：<string>
2. 建構式：string s
3. s=t：讓 s 的內容和 t 一樣，複雜度通常是 $O(size(s)+size(t))$
4. s+=t：將 t 加到 s 後面，複雜度通常是 $O(size(s)+size(t))$
5. s.cstr()：回傳和 s 一樣的 C 式字串，複雜度 $O(1)$
6. cin»s：輸入字串至 s，直到讀到不可見字元
7. cout«s：輸出字串 s
8. getline(cin,s,char c)：輸入 s 直到遇到字元 c，未指定 c 時，c 為預設字元。

```
// from cppreference
#include <iostream>
#include <string>
#include <sstream>// for getline

int main()
{
    // greet the user
    std::string name;
    std::cout << "What is your name? ";
    std::getline(std::cin, name);
    std::cout << "Hello " << name << ", nice to meet you.\n";

    // read file line by line
    std::istringstream input;
    input.str("1\n2\n3\n4\n5\n6\n7\n");
    int sum = 0;
    for (std::string line; std::getline(input, line); ) {
```

```
        sum += std::stoi(line);
    }
    std::cout << "\nThe sum is: " << sum << "\n";
}
/*
What is your name? John Q. Public
Hello John Q. Public, nice to meet you.

The sum is 28
*/
```

2.11 bitset

節省的 bool 陣列，可當二進位位元運算

1. 標頭檔：<bitset>
2. 建構式：bitset <N> b(a)
3. b[a]: 存取第 a 位, 複雜度 $O(1)$。
4. b.set(): 將所有位元設成 1。複雜度 $O(N)$。
5. b.reset(): 將所有位元設成 0。複雜度 $O(N)$。
6. b(位元運算): 不管是一元、二元運算皆可。二元位元運算長度需一致。複雜度 $O(N)$。
7. b.count(): 回傳 b 有幾個位元是 1。複雜度 $O(N)$。
8. b.flip(): 將所有位元的 0、1 互換。複雜度 $O(N)$。
9. b.to_string(): 回傳一個字串和 b 的內容一樣。複雜度 $O(N)$。
10. b.to_ulong(): 回傳一個 unsigned long 和 b 的內容一樣 (在沒有溢位的範圍內)。複雜度，$O(N)$。
11. b.to_ullong(): 回傳一個 unsigned long long 和 b 的內容一樣 (在沒有溢位的範圍內)，複雜度 $O(N)$。

2.12 priorty_queue 優先佇列

維護最大/小值，可插入、刪除、及詢問最大/小值，一種實作為 binary heap

```
int heap[N],top=0;
void push(int v){
    heap[++top]=v;
    for(int i=top;i>1;){
        if(heap[i]<=heap[i/2])break;
        swap(heap[i],heap[i/2]);
        i<<=1;
    }
}
void pop(){
    heap[1]=heap[top--];
    for(int i=1;(i<<1)<=top;){
        if(heap[i]<heap[i<<1]){
            swap(heap[i],heap[i<<1]);
            i<<=1;
        }else if((i<<1)<top&&heap[i]<heap[(i<<1)+1]){
            swap(heap[i],heap[(i<<1)+1]);
            i=(i<<1)+1;
        }else{
            break;
        }
    }
}
```

1. 標頭檔：<queue>
2. 建構式：priorty_queue <T> pq
3. 建構式：priorty_queue <T,Con,Cmp> pq
4. 建構式：priorty_queue <T,Con,Cmp> pq(iterator first, iterator seecond) 插入 $[first, second)$ 內的東西
5. pq.push(T a)：插入元素 a，複雜度 $O(\log size)$
6. pq.pop()：刪除頂端元素，複雜度 $O(\log size)$
7. pq.top()：回傳頂端元素，複雜度 $O(1)$
8. pq.size()：回傳元素個數，複雜度 $O(1)$
9. pq.empty()：回傳是否為空，複雜度 $O(1)$

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

int main(){
    priority_queue<int>Q;
    Q.push(2);
    cout<<Q.top()<<'\n';// 2
    Q.push(5);
    cout<<Q.top()<<'\n';// 5
    Q.pop();
    cout<<Q.top()<<'\n';// 2
    Q.push(3);
    cout<<Q.top()<<'\n';// 3
}
```

2.13 pair

兩個資料型態組織合盟

1. 標頭檔：‘<utility>‘
2. 建構式：‘pair <T1,T2> p‘
3. p.first,p.second：p 的第一、二個值
4. make_pair(T1 a1,T2 a2)：回傳一個 (a1,a2) 的 pair

2.14 tuple

generalize 的 pair

1. 標頭檔：‘<tuple>‘
2. 建構式：tuple<T1,T2...> t
3. ‘get<i>(t)‘：回傳 t 的第 i 個值。
4. make_tuple(T1 a1,T2 a2,...)：回傳一個 (a1,a2,...) 的 tuple。

2.15 set/map 自查找平衡二元樹

set 和 map 支援插入、刪除及查詢一個值，不同的是，set 會回傳鍵值，map 則是回傳對應值，也可以說 set 的鍵值和對應值一樣

2.16 set

1. 標頭檔：‘<set>‘
2. 建構式：‘set <T1> s‘
3. s.size()：回傳元素個數，複雜度 $O(1)$
4. s.empty()：回傳是否為空，複雜度 $O(1)$
5. s.clear()：清除元素，複雜度 $O(size)$
6. s.insert(T1 a)：加入元素 a，複雜度 $O(\log size)$。
7. s.erase(iterator first,iterator last)：刪除 $[first, last)$，若沒有指定 last 則只刪除 first，複雜度 $O(\log size)$ 與加上元素個數有關係。
8. s.erase(T1 a)：刪除鍵值 a，複雜度 $O(\log size)$。
9. s.find(T1 a)：回傳指向鍵值 a 的迭代器，若不存在則回傳 s.end()，複雜度 $O(\log size)$。
10. s.lower_bound(T1 a)：回傳指向第一個鍵值大於等於 a 的迭代器。複雜度 $O(\log size)$。
11. s.upper_bound(T1 a)：回傳指向第一個鍵值大於 a 的迭代器。複雜度 $O(\log size)$。

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

int main(){
  set<int>sb;
  sb.insert(1);
  sb.insert(2);
  sb.insert(3);

  cout<<"1 : "<<(sb.find(1)!=sb.end()?"find\n":"not find\n");
  cout<<"1 : "<<(sb.count(1)?"find\n":"not find\n");

  sb.erase(1);
  cout<<"1 : "<<(sb.find(1)!=sb.end()?"find\n":"not find\n");
  cout<<"1 : "<<(sb.count(1)?"find\n":"not find\n");
}
/*
1 : find
1 : find
1 : not find
1 : not find
*/
```

2.17 map

1. 標頭檔：‘<map>‘
2. 建構式：‘map <T1, T2> m‘
3. m.size(),m.empty(),m.clear(),m.erase(iterator first,iterator last),m.erase(T1 a),m.find(T1 a),m.lower_bound(T1 a),m.upper_bound(T1 a)：同 set。
4. ‘m[a]‘：存取鍵值 a 對應的值，若 a 沒有對應的值，會插入一個元素，使 a 對應到預設值並回傳之。複雜度 $O(\log size)$。
5. m.insert(pair<T1,T2> a)：若沒有鍵值為 a.first 的值，插入一個鍵值為 a.first 的值對應到 a.second，並回傳一個 pair,first 是指向剛插入的元素的迭代器、second 是 true；若已經有了，回傳一個 pair,first 是指向鍵值為 k.first 的元素的迭代器,second 是 false。複雜度 $O(\log size)$。

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

int main(){
    map<string,int>tb;
    tb["123"]=1;
    tb["owowowo"]=2;
    tb["omomo"]=3;
    cout<<"tb[\"123\"]: "<<tb["123"]<<'\n';
    cout<<"tb[\"owowowo\"]: "<<tb["owowowo"]<<'\n';
    cout<<"tb[\"omomo\"]: "<<tb["omomo"]<<'\n';

    cout<<"123 : "<<(tb.find("123")!=tb.end()?"find\n":"not find\n");
    cout<<"123 : "<<(tb.count("123")?"find\n":"not find\n");

    tb.clear();
    cout<<"123 : "<<(tb.find("123")!=tb.end()?"find\n":"not find\n");
    cout<<"123 : "<<(tb.count("123")?"find\n":"not find\n");

    cout<<"owo : "<<(tb.find("owo")!=tb.end()?"find\n":"not find\n");
    tb.insert(make_pair("owo",659));
    cout<<"owo : "<<(tb.find("owo")!=tb.end()?"find\n":"not find\n");
}
/*
tb["123"]: 1
tb["owowowo"]: 2
tb["omomo"]: 3
123 : find
123 : find
123 : not find
123 : not find
owo : not find
owo : find
*/
```

2.17.1 multi-系列

可插入重複元素代價為 map 無法用下標運算子

1. equal_range(T1 a)：回傳 iterator 的 pair<lower_bound(a),upper_bound(a)>，為 a 所在範圍
2. erase(T1 a)：刪除所有元素 a，如果只要刪除一個，用 s.erase(s.find(a))

2.17.2 unorder_ 系列

降低常數，期望複雜度少一個 log，代價為不會排序，沒有 lower_bound/upper_bound，也不會依鍵值大小遍歷。迭代器為單向。

2.18 題目

2.18.1 stack,queue,deque

先來看一題題目

(zjd555) 在平面上如果有兩個點 (x,y) 與 (a,b), 我們說 (x,y) 支配 (Dominate) 了 (a,b) 這就是指 $x \geq a$ 而且 $y \geq b$；用圖來看就是 (a,b) 座落在以 (x,y) 為右上角的一點無的區域中。
對於平面上的任意一個有限點集合而言，一定存在有若干個點，它們不會被集合中的內一點所支配，這些個數就構成一個所謂的極大集合。請寫一個程式，讀入一個新的集合，找出這個集合中的極大值。
簡單的說若找不到一點在 (x,y) 的右上方, 則 (x,y) 就要輸出

我們能肯定最右上角的點一定是極大點，再來的極大點一定都在他的右下方，那我們就以座標排序，先以 y 座標由大到小排序，再以 x 座標由大到小排序，每找到一個極大點，就找下一個 y 值比它小，x 比它到的點，那也是極大點，就這樣以此類推。
我們觀察到極大集合中，如果先以 Y 值排序由大到小排序，X 值會由小到大，如果我們發現，答案的候選人，有如果... 就大於/小於... 的關係，我們

就稱答案有” 單調性”。單調性這個東西筆者覺得很玄，我沒辦法明確定義什麼事單調性，需要多多寫題目才能自己感覺出來。
下列提供 stack,queue,deque 的題目，前半部分是基礎應用，後半部分是關於單調性的。

1. UVa514(Stack 應用)
2. UVa673(Stack 應用)
3. ZeroJudge d016(Stack 後續運算法)
4. UVa10935(Queue 應用)
5. UVa12100(Queue 應用)
6. Uva246(Deque 應用)
7. UVa11871(stack 單調)
8. TIOJ1618(Deque 單調)

2.18.2 list

1. ZeroJudge d718
2. TIOJ 1225
3. TIOJ 1930

2.18.3 string

1. ZeroJudge a011(getline 應用)
2. Zerojudge d098(StringStream 應用，請自行查詢)

2.18.4 PriortyQueue

1. Uva 10954
2. Uva 1203

2.18.5 set,map

1. ZeroJudge d512(Set 應用)
2. UVa 10815(Set 應用)
3. Zerojudge d518(Map 應用)
4. UVa 484(Map 應用)

3 演算法

3.1 何謂演算法

簡言之就是解決問題的方法，用程式語言把他明確地列出。

3.2 枚舉

枚舉是最直觀的演算法，將有可能的答案都搜過一遍，當然沒有頭緒的搜尋可能會得到龐大的複雜度，要根據題目的性質來降低複雜度。

3.2.1 回朔

枚舉有時能用遞迴實作，在遇到不可能的情形馬上回傳，這種方法就叫做回朔ㄊ。

3.2.2 特殊枚舉方式

二進位 利用二進位來表示集合內有哪些元素要用，進而枚舉所有元素子集，但受限於時間複雜度 $O(2^n)$，集合的元素個數通常只有 30 個 (甚至 15 個)。

字典序 利用 next_permutation 或 prev_permutation 達到枚舉元素的先後順序。時間複雜度為 $O(N!)$

3.2.3 折半枚舉

有時遇到複雜度 $O(2^n)$ 的算法，在無法用其他方法降低複雜度，可以試著將元素切成兩半，降低 n，再用其他算法組合起來。

3.2.4 題目

1. UVa 11059(區間列舉)
2. UVa 1481(區間列舉)
3. UVa 10976(減少列舉範圍)
4. UVa 750(回朔)
5. UVa 524(回朔)
6. UVa 11464
7. UVa 1326(折半枚舉)

3.3 貪心

對於一個問題，始終使用同一種方法，採取在目前狀態下最好或最佳（即最有利）的選擇。
有的貪心很直觀，有的就需要通靈才解得出來，往往做題目一開始想到的辦法是錯的，直到做到一半才發現。所以我們需要證明方法是不是對的，這往往需要時間練習，才不會到比賽遇到時，花了很多時間去解題。

3.3.1 證明的辦法

1. 試圖構造出反例，發現他不存在。
2. 如果存在更佳解的答案比你做出來的還好，那這組解一定可以再做得更好，進而達到反證出更佳解不存在。
3. 使用遞迴證法:(1) 證明基底是對的。(2) 假設小問題是好的。(3) 你一定可以用最好的方法來將問題簡化成剛才假設是好的小問題。

3.3.2 題目

1. UVa 11729
2. UVa 11292
3. UVa 11389
4. UVa 1445
5. UVa 993
6. TIOJ 1441

3.4 二分搜

對於一個函數 $F(n)$，如果存在一個 x，對於所有 $\geq x$ 的 a，$F(a) =$ true，反之 $F(a) =$ false，基於這樣的單調性，就可以用二分搜。

```
T binary_search(){
    while(L<R){
        int M=(L+R)>>1;
        if(F(M))R=M;
        else L=M+1;
    }
    return L;
}
```

有些題目為”最多/最少為何會成立”，那麼如果你可以在良好的時間檢查出”如果代價是 x，那可不可以達成目標”，並且 x 具有單調性，那麼你可以轉換成”如果代價是 x，那可不可以達成目標”傳換成 $F(x)$，對答案 (x) 進行二分搜。
二分搜要注意兩件事，一個是無限迴圈，要避免它可以在腦中先模擬一下。一個是在實數中二分搜，因為實數的稠密性，題目會有誤差容忍 (例如 10^{-6})，只要在誤差內都是容許的。

3.4.1 三分搜

對於 U 型函數 (例如 $y = F(x) = x^2$)，我們想要找尋其極值，意謂其左右兩側皆各自遞增/遞減，我們可以利用三分搜來解決 (二分搜只能解決全體單調性，不能解決有兩邊的)。
考慮三分後從左到右四個採樣點的關係

1. $S(a) < S(b) < S(c) < S(d)$，此時最小值一定不在最右邊
2. $S(a) > S(b) < S(c) < S(d)$，此時最小值一定不在最右邊
3. $S(a) > S(b) > S(c) < S(d)$，此時最小值一定不在最左邊
4. $S(a) > S(b) > S(c) > S(d)$，此時最小值一定不在最左邊

這段描敘還可以再簡化

1. $S(b) < S(c)$，此時最小值一定不在最右邊
2. $S(b) > S(c)$，此時最小值一定不在最左邊

每次都至少可以讓區間縮小 $\frac{1}{3}$

```
double trinary_search(double l,double r){
static const double EPS=1e-7;
while(r-l>EPS){
        double ml=(l+l+r)/3,mr=(l+r+r)/3;
        if(f(mr)>f(ml)) r=mr;
        else l=ml;
    }
    return l;
}
```

3.4.2 題目

1. Uva 714
2. Uva 1421
3. Uva 11627
4. zerojudge d732
5. neoj 72(三分搜)

3.5 分治

分治法會把問題分解成子問題 (分)，解決完再合併回原本的問題 (治)。
分治分成以下步驟

1. 切割：把一個問題切成子問題然後遞迴
2. 碰底：碰到不能再切割或是明顯有答案 (也許無解)，就算出答案再回傳
3. 合併：利用傳回來的子問題算出答案然後回傳

3.5.1 合併排序法

一個利用分治實作的排序法，逆序數對也會利用他的概念來實作。

1. 切割：把序列分成兩半然後遞迴
2. 碰底：直到序列長度為 1，這時候已為一個排好的序列，直接回傳
3. 合併：利用傳回來的兩串序列進行排序

```
using namespace std;
const int N = 100;
int arr[N], buf[N];
void sol(int L, int R) { // [L,R)
  if (R - L <= 1)return;
  int M = (R + L) / 2;
  sol(L, M);
  sol(M, R);
  int i = L, j = M, k = L;
  while (i < M || j < R) {
    if (i >= M)
      buf[k] = arr[j++];
    else if (j >= R)
      buf[k] = arr[i++];
    else {
      if (arr[i] <= arr[j])
        buf[k] = arr[i++];
      else {
        buf[k] = arr[j++];
      }
    }
    k++;
  }
  for (int k = L; k < R; k++) arr[k] = buf[k];
  return;
}
```

3.5.2 更多的經典題目

1. 快速排序法。
2. 逆序數對。(經典問題, 搭配 Merge Sort)

3.5.3 題目

1. uva 1608
2. uva 10810(逆序數對)
3. uva 11129
4. uva 10245

4 動態規劃

動態規劃 (Dynamic Programming, DP) 和遞迴一樣，會把問題分解成子問題，不同的是 DP 同樣的子問題會重複，這時會以”時間換取空間”的方式，把答案存起來，之後用到時不用再次進行計算。

4.1 特性

1. 重複子問題：相同的一個子問題，需要多次查詢。
2. 最佳子結構：當問題被拆成若干個規模較小的問題時，可以透過這些子問題得到這個問題的最佳解。
3. 無後效性：子問題不會互相呼叫，一定存在一個能完整求值的計算順序。

4.2 步驟

DP 通常會被定義成一個算出答案的函數，函數的參數個數可變，對於一種題目可能有不同種的定義辦法，要看自己如何決定。

1. 訂出狀態
2. 導出轉移：從一個子問題的答案推到另一個
3. 設好基底：有些問題明顯有答案 (也許無解)，就直接算出答案

而時間複雜度，通常就會是「使用到的狀態個數」乘上「轉移時間」了。

4.3 費式數列

費式數列是個看到膩的數列，我們就把他的 DP 式列出來

1. 狀態：$f(x)$ 代表費式數列第 n 項
2. 轉移：$f(n) = f(n-1) + f(n-2)$
3. 基底：$f(n) = n, where\ \ n \leq 1$

我們先用遞迴的方式寫寫看

```
int f(int n){
    if(n<=1)return n;
    return f(n-1)+f(n-2);
}
```

實際測試，會發現數字到 40 左右就跑不太動，原因是因為他有很多重複的子問題，那麼我們就以”空間換時間”的辦法來加速。

```
int dp[N]
int f(int n){
    if(n<=1)return n;
    if(dp[n])return dp[n];
    return dp[n]=f(n-1)+f(n-2);
}
```

這種用遞迴實作的叫做 Top-down，好處是容易寫出來，缺點是遞迴的常數非常大。
而費式數列還有一種用迴圈的寫法。

```
int dp[N];
int f(){
    dp[0]=dp[1]=1;
    for(int i=2;i<N;i++){
        dp[i]=dp[i-1]+dp[i-2];
    }
}
```

這種的迴圈實作的叫做 Buttom-up，雖然比較難寫，但是時間常數比遞迴小，有時可以利用”滾動陣列”來壓縮空間。
第一個單純遞迴的版本，其複雜度等同費式數列第 N 項。第二、三版本，每項費式數列只會被算一次，所以複雜度為 $O(N)$。

4.4 滾動陣列

如果一個 DP 式只會用到臨近的項數，就可以捨棄其他用不到的，藉此壓縮空間。例如：費式數列只會用到前兩項，我們就用取餘數來實作滾動陣列，注意被餘數要保持正數。

```
int dp[3];
int f(){
    dp[0]=dp[1]=1;
    for(int i=2;i<N;i++){
        dp[i%3]=dp[(i-1+3)%3]+dp[(i-2+3)%3];
    }
}
```

4.5 題目

1. UVa10003(區間 DP)
2. UVa10285
3. zerojudge d212
4. UVa11310

接下來我們談論一些經典問體

4.6 背包問題

背包問題中，會給你背包的容量 C，以及 N 樣物品，它們有不同的重量 w 和價值 v，背包裡物品價值總和最大為何?

4.6.1 0/1 背包

01 背包限制每樣物品最多只能放一個。那就來寫 DP 式。

1. 狀態：$dp[n][i]$ = 使用 $1\mathbb{R}n$ 個物品湊出重量 i 時，所可得到的最大價值。
2. 轉移：$dp[n][i] = max(dp[n-1][i-w[n]] + v[n], dp[n-1][i])$
3. 基底：$dp[0][0] = 0$, $dp[0][i] = -INF$ when $i > 0$

以防在 DP 的過程不小心用到，湊不出來的狀態要設成負無限大，或是使用 if 判斷取代預設。

```
int dp[N][C];
memset(dp,-INF,sizeof(dp));
dp[0][0]=0;
for(int i=0;i<N;i++){
    for(int j=w[i];j<=C;++j){
        dp[i][j]=max(dp[i-1][j-w[i]]+v[i],dp[i-1][j]);
    }
}
```

現在時間空間複雜度皆為 $O(NC)$，而空間可以在持續優化。
記得剛剛說過的滾動陣列嗎? 先回去看看 DP 式，$dp[n]$ 轉移只會用到 $dp[n-1]$，所以我們只要保留前一行（行列的區分我不管）就好了。

```
int dp[2][C];
memset(dp,-INF,sizeof(dp));
dp[0][0]=0;
for(int i=0;i<N;i++){
    for(int j=w[i];j<=C;++j){
        dp[i&1][j]=max(dp[i&1^1][j-w[i]]+v[i],dp[i&1^1][j]);
    }
}
```

這樣的空間複雜度為 $2C$，再來，我們發現第二個維度，他只會用到小於它的地方，那麼我們可以只用一行。但是要由後往前轉移，這樣才不會讓還沒轉移到的狀態，取到已轉移的，保證該項物品只會被取一次。

```
int dp[C];
memset(dp,-INF,sizeof(dp));
dp[0]=0;
for(int i=0;i<N;i++){
    for(int j=C;j>=w[i];j--){
        dp[j]=max(dp[j-w[i]]+v[i],dp[j]);
    }
}
```

4.6.2 無限背包

無限背包每樣物品能放無限多個。同樣要先列出 DP 式。

1. 狀態：$dp[n][i]$ = 使用 $1 \sim n$ 個物品湊出重量 i 時，所可得到的最大價值。
2. 轉移：$dp[n][i] = max(dp[n-1][i-w[n]] + v[n], dp[n][i-w[n]] + v[n], dp[n-1][i])$
3. 基底：$dp[0][0] = 0, dp[0][i] = -INF$ when $i > 0$

如果第 i 個物品重量為 w_i，那麼最多可以放 $\frac{C}{w_i}$ 次，要多一層迴圈來模擬。

```
int dp[C];
memset(dp,-INF,sizeof(dp));
dp[0]=0;
for(int i=0;i<N;i++){
    for(int j=w[i];j<=C;j++){
        for(int k=1;j-k*w[i]>=0;k++){
            dp[j]=max(dp[j-k*w[i]]+k*v[i],dp[j]);
        }
    }
}
```

時間複雜度為 $O(NC\Sigma\frac{C}{w_i})$
我們現在回到之前 01 背包最後一個範例，當初要由後往前，是為了保證每個物品都只放了一個，而現在我們要放越多越好，那我們就由前面開始轉移就行了。

```
int dp[C];
memset(dp,-INF,sizeof(dp));
dp[0]=0;
for(int i=0;i<N;i++){
    for(int j=w[i];j<=C;j++){
        dp[j]=max(dp[j-w[i]]+v[i],dp[j]);
    }
}
```

時間複雜度為 $O(NC)$

4.6.3 有限背包

有限背包每樣物品能放限制最多可以放 a_i 個。這裡請讀者自己列出 DP 式。我們一樣由 01 背包延伸，一樣物品最多可以放 a_i 個，那麼迴圈最多要跑 a_i 次，這樣整體時間複雜度為 $O(NC\Sigma a_i)$。
有限背包又沒有像無限背包那麼好的性質，但如果我們帶入二進位的觀念，第 i 位代表 2^{i-1}，我們把 a_i 個物品拆成 $1, 2, 4..., 2^k$ 次方個枚舉，那麼我們可以將時間複雜度降為 $O(NC\Sigma(\log a_i))$ 講道更進階一點的 DP，會學到”單調隊列優化”，能將時間複雜度降為 $O(NC)$

4.6.4 題目

1. zerojudge b184(01 背包)
2. UVa10664(01 背包)
3. UVa10898(無限背包)
4. Uva10086(有限背包)
5. TIOJ 1387(有限背包)

4.7 最長共同子序列 (Longest Common Subsequence)

子序列是指一個序列去除某些元素後所形成的新序列 (當然也可以不刪除任何東西)，而所謂的最長共同子序列，則是要求兩個字串去除元素後變成相同的序列，最長可以為何。這題觀念是簡單的，在這裡就列出 dp 式供讀者自行解讀。

1. 狀態：$dp[i][j]$ = 使用 $a[1]$ $a[i]$ 和 $b[1]$ $b[j]$ 的 LCS 長度。
2. 轉移：$dp[i][j] = \left\{ \begin{array}{cc} dp[i-1][j-1]+1 & if\ a[i]=b[j] \\ max(dp[i-1][j], dp[i][j-1]) & else \end{array} \right\}$
3. 基底：$dp[i][0] = dp[0][i] = 0$ when $i \geq 0$

4.7.1 題目

1. Uva10405

4.8 最長遞增子序列 (Longest Increasing Subsequence)

給你一個序列長度為 N，求最長遞增子序列的長度。
你可能會在其他地方看見一維 DP 式，在這裡不提，有興趣可以去演算法筆記看，這裡要介紹另一種 DP 式

1. 狀態：$dp[n][i]$ = 使用 $1 \sim n$ 個數字湊出長度 i 的 LIS, 末端數字最小為何。
2. 轉移：$dp[n][i] = \left\{ \begin{array}{cc} min(dp[n-1][i], a[n]) & if\ a[n] > dp[n-1][i-1] \\ dp[n-1][i] & else \end{array} \right\}$
3. 基底：$dp[0][0] = -INF, dp[0][i] = INF, dp[i][0] = don't\ care$ when $i \geq 1$

時間複雜度為 $O(n^2)$。
再來，我們令 $g[i] = dp[x][i]$, where $1 \leq x \leq N$，我們會發現 $g[i]$ 為一個嚴格遞減序列，而且改的數字剛好的 $a[i]$，更妙的是，被改的數字為 $lower_bound(a[i])$，我們就可以用二分搜來得到 $O(n \log n)$ 的演算法了。

```cpp
int main(){
    int n;
    while(cin>>n){
        vector<int>v;
        for(int i=0,x;i<n;i++){
            cin>>x;
            if(!v.size()||x>v.back())v.push_back(x);
            else *lower_bound(v.begin(), v.end(),x)=x;
            dp[i]=v.size();
        }
    }
}
```

4.8.1 題目

1. UVa11456
2. UVa11790

5 Graph

圖是由邊集合和點集合所形成的圖形，這種圖形通常用來描述某些事物之間的某種特定關係。頂點用於代表事物，連接兩頂點的邊則用於表示兩個事物間具有這種關係。
數學式為 $G = (V, E)$。G 代表圖 (Graph)，V 代表點 (vertex)，E 代表邊 (edge)。

5.1 術語

在深入圖論中，我們先介紹一些術語，這些術語在後面的內容，不時會扮演著關鍵角色。

1. 無向邊、有向邊：邊具有方向性，無向邊代表邊沒有指定方向，(u,v) 和 (v,u) 等價；有向邊則有指定方向，(u,v) 和 (v,u) 是不同的。
2. 無向圖、有向圖、混合圖：無向圖是只有無向邊的圖，類似地，有向圖是只有有向邊的圖，混和圖則是包含無向邊和有向邊。
3. 點、邊個數：一般來說會用 $|V|$、$|E|$ 來表示，在表示複雜度時，為了方便會用 V、E 來表示
4. 權重 (weight)：在點或邊上附帶一個數字稱做” 權重”，邊上權重較常見，權重通常代表代價，例如所需花費時間或金錢。
5. 相鄰 (adjacent)：無向圖中，兩個點 u，v 相鄰代表存在一個邊 $e_i=(u,v)$。
6. 指向 (consecutive)：有向圖中，u 指向 v 代表存在一個邊 $e_i=(u,v)$。
7. 度 (degree)：無向圖中，一個點連到的邊數稱為” 度”，在有向圖稱為出度 (out-degree，簡稱 d_{out}) 及入度 (in-degree，簡稱 d_{in})，分別代表該點指向別點及被指向的邊數。
8. walk：一條由 x 到 y 的路徑 $x=v_1,v_2,v_3...,v_k=y$。
9. trail：一條不重複邊的 walk。
10. 迴路 (circut)：起點和終點一樣的 trail。
11. path：一條不重複點 (起點和終點例外) 的 walk。
12. 環 (cycle)：起點和終點一樣的 path。
13. 自環 (loop)：一條邊 $e_i=(u,v)$ 滿足 $u=v$,e_i 即稱為自環。
14. 重邊 (multiple edge)：在一張圖中，存在 e_i，e_j 滿足 i != j and $e_i=e_j$，則稱為重邊。
15. 連通 (connected)：無向圖中，若 u 和 v 存在路徑，則 u 和 v 連通。若一群點兩兩連通，則這些點都連通。

5.2 儲存

至於圖要怎麼存起來呢，以下介紹兩種辦法。設 V 為點數，E 為邊數。

5.2.1 相鄰矩陣 (adjacency matrix)

：開一個 $V\times V$ 的資料結構 M(通常會用二維陣列)，$M[a][b]$ 代表的是點 a 至 b 的邊數或權重。空間複雜度 $O(V^2)$。加、刪邊時間複雜度 $O(1)$。

5.2.2 相鄰串列 (adjacency list)

：開 V 個可變長度的資料結構 (通常在 C++ 用 vector、在 C 用 linked list)，第 i 個裡面放所有第 i 個點指向的點的編號 (和邊權或其他邊的資訊)。空間複雜度 $O(V+E)$，加邊時間複雜度 $O(1)$、刪邊時間複雜度 $O(V)$。至於使用時機，如果邊數較密，且頻繁地需要找尋兩點之間的權重，那麼相鄰矩陣比較適合，其餘情況則是用相鄰串列。

5.3 遍歷

存好圖後，為了獲得某些資訊，需要一些讀取的方法，這些方法我們叫做” 遍歷”。以下介紹兩種方法：DFS 和 BFS。

5.3.1 DFS

找到一條新的路就繼續找下去，直到沒有新的路時，原地返回。通常用遞迴實作或用 stack 維護。

```
vector<int>G[N];
bitset<N> vis;
void dfs(int s){
    vis[s]=1;
    for(int t:G[s]){
        if(!vis[i])dfs(i);
    }
}
```

5.3.2 BFS

```cpp
vector<int>G[N];
bitset<N> vis;
void bfs(int s){
    queue<int>q;
    q.push(i);
    vis[i]=1;
    while(q.empty()){
        int v=q.front(); q.pop();
        for(int t:G[v]){
            if(!vis[t]){
                q.push(t);
                vis[t]=1;
            }
        }
    }
}
```

把所有看到的路都加入清單中，並且以加入的順序來遍歷。通常以 queue 來維護。
BFS 和 DFS 的時間複雜度皆為 $O(V+E)$，空間複雜度皆為 $O(V)$。
注意，圖不一定完全連通，我們通常會另外開一個陣列 (bitset) 紀錄是否拜訪過。

5.3.3 題目

1. zerojudge a290(給你一張有向圖，問你可不可以由 A 走到 B)
2. zerojudge a982(二維迷宮問題)
3. zerojudge a634(馬步問題)
4. UVa 572

5.4 樹

樹是一種特殊的圖，有許多算法都是由樹發展出來。

5.4.1 特性

以下這些無向圖的敘述都是在表示樹，這些敘述在競賽中有時能引導出答案。

1. 為連通圖且 $|V|=|E|+1$
2. 任意兩個點之間存在唯一路徑
3. 為連通圖，但拔掉一條邊即為不連通 (分成兩張連通圖)。
4. 沒有環，但加上一條邊會形成環。
5. 若節點編號存在順序，除了第一個節點，每個節點都會伸出一條邊連到順序比自己前面的節點。

5.4.2 術語

樹同樣也有一些術語要知道的。

1. 根 (root)：樹的一個代表性的點，通常會被當遍歷的起點，有給定根點的樹叫” 有根樹”。至於無根樹依照題目需求，有時要隨機找一個點當根。
2. 葉解點 (leaf)：度數為 1 的節點，有根樹的根結點則會是題目需求來決定是否為葉節點。
3. 父節點、子節點：有根樹中，兩個相連的節點，比較接近樹根的為父節點，反之為子節點。
4. 祖先 (ancestor)、子孫 (descandent)：有根樹中，節點到根結點中，所有的節點皆為祖先。反過來說，該節點是他的祖先的子孫。依題目所需，有時自己也是自己的祖先 (尤其是根最常這樣定義)。
5. 距離 (distance)：為兩個點所形成路徑之邊數，或是路徑上權重之和。
6. 深度 (depth)：有根樹中，節點到根結點之距離。

7. 高度 (height)：有根樹中，節點到與它距離最大的葉節點的距離稱為高度。根的高度稱為這整顆樹的高度。
8. 子樹 (subtree)：設有兩棵樹 T,T_1，如果 $V_1 \in V$，$E_1 \in E$，那麼我們說 T_1 為 T 的子樹
9. N 元樹：每個節點最多有 N 個節點，稱為 N 元樹。

5.5 二元樹

二元樹在程式競賽中常常被討論，有許多資料結構都是二元樹，例如 STL 提到的 heap。

5.5.1 遍歷

基於根結點遍歷順序，二元樹的遍歷有”前序、中序、後序”三種辦法，通常 DFS 會使用前序來實作，如果想要得到任何一種順序的結果，只要改變輸出的順序。

```
void dfs(int v){
    // cout<<v<<'\n'; preorder
    if(v.lc)dfs(v.lc);
    // cout<<v<<'\n'; inorder
    if(v.rc)dfs(v.rc);
    // cout<<v<<'\n'; postorder
}
```

5.5.2 二元搜尋樹 (Binary Search Tree, BST)

二元搜尋樹是二元樹的應用，利用遞迴方式來定義，如下：

1. 根結點的值大於左子節點的值，小於右子節點的值。
2. 其左右子樹亦為二元搜尋樹。

用上敘定義，就可以建造出 BST，不過如果我們將一個以排序的串列建成 BST，會發現 BST 會”退化”成一條傾斜的串列。BST 本身不實用，重要在於它的推廣結構，例如 AVL 樹、紅黑樹、treap，不過這些資料結構比較進階，在這裡先不提。

5.6 並查集

並查集是一種樹狀結構，他支援兩件事

1. 查詢所隸屬集合
2. 合併兩個集合

我們把集合轉化成樹，一顆樹代表一個集合，樹根代表集合的老大，查詢隸屬集合就回傳樹根是誰 (一個樹舖可能有兩顆樹根吧)，合併的時侯，就把一顆樹的樹根只到另一顆，以下為詳細的描述。

5.6.1 初始

一開始的時候，每個點自成一個集合，所以把樹根都設為自己。

5.6.2 查詢

查詢的時候，要查到樹根為自己的點，為止否則的話就要繼續查。

5.6.3 狀態壓縮

在合併之後原本被指向的樹根就沒用了，我們可以一邊做查詢時，一邊做更新。

5.6.4 啟發式合併

建立一個 $h[i]$ 代表樹的高度，亦是元素最大遞迴次數，$h[i]$ 一開始為 1。再來，我們每次都讓高度小的高度大的合併，如果遇到高度一樣的，就讓合併別人的樹高度加 1。如果要把高度變為 x，則至少需要 2^x 個點，由此推出 N 個點所形成最高之高度為 $\log(N)$。

```
int p[N],rank[N];
void init(){
    for(int i=0;i<N;i++){
        p[i]=i;
        rnak[i]=1;
    }
}
int Find(int x){
    if(x==p[x])return x;
    return p[x]=find(p[x]);
}
void Union(int a,int b){
    a=Find(a); b=Find(b);
    if(a==b)return;
    if(rank[a]<rank[b])p[a]=b;
    else if(rank[a]>rank[b])p[b]=a;
    else {p[a]=b; rank[a]++;}
}
```

複雜度為 $O(\alpha(N))$。並查集最常用的地方是最小生成樹的 Kruskal's algorithm。

5.6.5 題目

1. zerojudge d808
2. UVa 1160
3. UVa 10158(陣列開兩倍)
4. UVa 1329(帶權並查集)

5.7 最小生成樹 (Minimun Spanning Tree, MST)

在一張圖中，如果有子圖剛好為也為一顆樹，我們就稱該子圖為生成樹。現在我們在圖上加上權重，而在所有的生成樹中，權重最小的，我們稱為" 最小生成樹"，最小生成樹並不唯一，以下介紹幾種最小生成樹的演算法。

5.7.1 Kruskal's algorithm

Kruskal's algorithm 的概念是，合併兩顆 MST 的時候，加入連接兩顆樹中，最小權重的邊。所以我們就利用 greedy，將邊依權重由小到大排序，如果邊的兩邊是在不同的 MST，我們就把合併 (並查集應用於此)，反之就跳過。排序需花 $O(E\log E)$ 的時間，選邊需要花 $O(E\alpha(V))$ 的時間，總共時間複雜度 $O(E(\log E+\alpha(V)))$

```
struct Edge{
    int s,t,w;
    bool operaotr<(const Edge&rhs)const{
        return w<rhs.w;
    }
};
void Kruskal(){
    int cost=0;
    vector<Edge>E;
    init();
    for(auto it:E){
        it.s=Find(it.s);
        it.t=Find(it.t);
        if(it.s==it.t)continue;
        cost+=it.w;
        Union(it.s,it.t);
```

```
17      }
18  }
```

5.7.2 Prim's algorithm

Prim's algorithm 的思維則是，將一棵 MST 連出的邊中，加入權重最小的邊 (距離最近的點)，重複執行後得出最小的生成樹。在實作上，首先取一個點當 MST，更新所有與它相鄰的點，更新後把距離最小的點加入 MST(不用並查集)，持續執行更新及加入點的動作，直到所有點都已加入 MST。維護最小距離用 priority_queue 維護，每個點只會被合併一次，每條邊都只會遍歷一次，複雜度 $O((V+E)logE)$。另外有一個資料結構用費波那契堆 (fibonacci heap) 可以達到 $O(E+V\log V)$。但是因為它常數比較大，實作複雜，我們不會使用它。總體而言，Kruskal 比 Prim 好用。

5.7.3 Borůvka's algorithm

Borůvka's algorithm 和 Prim 一樣都在加入 MST 和最鄰近的點，不一樣的是，它讓所有的 MST 一起做這件事。每次找出每棵 MST 外權重最小的邊，並加入 MST(如果權重一樣，就找索引值最小的)，檢查是否只剩一棵 MST，如果不是就重複掃描的動作，這裡一樣用並查集維護聯通性。
最差的情況為每次都剛好兩兩成對合併，這樣最多只會執行 $\log V$ 次，整題複雜度為 $O((V+E)\log V)$)。期望複雜度可以達到 $O((V+E))$ (因為每次並查集都會被合併 + 查詢，所以 α 可以完全省略)。

5.7.4 題目

1. zerojudge a129

5.8 最短路徑

5.8.1 術語

1. 負邊：權重為負的邊
2. 負環：權重和為負的環
3. 點源：成為起點的點，分成單源頭及多源頭。
4. 鬆弛：單源頭最短路徑中，對於任意兩個點 u,v，起點 s 到它們的距離 d_u,d_v，如果 $d_u > d_v + w_{u,v}$，$w_{u,v}$ 為邊 (u,v) 的權重，我們可以讓 d_u 更新為 $d_v + w_{u,v}$，讓 s 到 u 的距離縮短，這個動作稱為" 鬆弛"。

5.8.2 Floyd-Warshall Algorithm

為多源頭最短路徑，求出所有點對的最短路徑。
Floyd-Warshall 是動態規劃，以下是他的 dp 式。

1. 狀態：$dp[k][i][j]$ 代表，若只以點 $1\text{⁄}2k$ 當中繼點的話，則 $dp[k][i][j]$ 為 i 到 j 的最短路徑長。
2. 轉移：$dp[k][i][j] = min(dp[k\text{-}1][i][k] + dp[k\text{-}1][k][j], dp[k\text{-}1][i][j])$
3. 基底：$dp[0][i][j] = \left\{ \begin{array}{cc} w[i][j] & if\ w[i][j]\ exists \\ INF & else \end{array} \right\}$

時/空間複雜度皆為 $O(V^3)$，然而此 DP 是可以滾動，所以空間複雜度可降為 $O(V^2)$

```
for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
          w[i][j] = w[j][i] = min(w[i][j], max(w[i][k], w[k][j]));
```

執行的時候如果 $dp[i][j] < 0$，代表存在負環，Floyd-Warshall 是可以判斷負環。

5.8.3 單點源最短路徑

求出一個點到所有點的最短路徑，其實就是以起點為根，最短路徑是由父節點鬆弛而來的最短路徑樹。我們找最短路徑，就是一直把鬆弛，直到所有點都不能鬆弛，所有點都獲得最短路徑了。要蓋出最短路徑樹，就只要把點指向最後一次被誰鬆弛就好了。

5.8.4 Bellman-Ford Algorithm

為單點源最短路徑，設起點的最短路徑為 0，其他點為無限大，每次對所有邊枚舉，因為最短路徑不會經過同樣的邊第二次，所以只要執行 $V-1$ 輪，複雜度為 $O(VE)$。如果執行第 V 次時還有邊可以鬆弛，代表有負環，Bellman-Ford 也可以當成負環的判斷方法。

```
void bellman_ford(int s){
    d[s]=0;
    p[s]=s;
    for(int i=0;i<V-1;i++){
        for(int ss=0;ss<V;ss++){
            for(auto tt:v[ss]){
                if(d[ss]+w[ss][tt]<d[tt]){
                    d[tt]=d[ss]+w[ss][tt];
                    p[tt]=ss;
                }
            }
        }
    }
}
void has_negative_cycle(){
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(d[i]+w[i][j]<d[j])return true;
        }
    }
    return false;
}
```

此演算法還有一個優化版本叫做 Shortest Path Faster Algorithm(SPFA)，他的做法是枚舉起點是鬆弛過的邊，以鬆弛過的點除非被重新鬆弛，否則不會更動。預期複雜度為 $O(V+E)$，不過最差狀況仍為 $O(VE)$。

5.8.5 Dijkstra’s Algorithm

同樣為單點源最短路徑，他的想法和 Prim’s Algorithm 類似，每次把離樹根最近的點加入最短路徑樹裡，並把所有與該點相連的邊鬆弛，已經加入的點不會在被鬆弛。使用 priority_queue 的複雜度為 $O((V+E)\log E)$，使用費波那契堆，複雜度為 s$O(E+V\log V)$

```
struct edge{
    int s,t;
    LL d;
    edge(){};
    edge(int s,int t,LL d):s(s),t(t),d(d){}
};

struct heap{
    LL d;
    int p; //point
    heap(){};
    heap(LL d,int p):d(d),p(p){}
    bool operator <(const heap& b)const{
        return d>b.d;
    }
};
int d[N],p[N];
vector<edge>edges;
vector<int>G[N];
bitset<N>vis;
void dijkstra(int ss){
    priority_queue<heap>Q;
    for(int i=0;i<V;i++)d[i]=INF;
    d[ss]=0;
    p[ss]=-1;
    vis.reset():
    Q.push(heap(0,ss));
```

```
28      heap x;
29      while(!Q.empty()){
30          x=Q.top(); Q.pop();
31          int p=x.p;
32          if(vis[p])continue;
33          vis[p]=1;
34          for(int i=0;i<G[p].size();i++){
35              edge& e=edges[G[p][i]];
36              if(d[e.t]>d[p]+e.d){
37                  d[e.t]=d[p]+e.d;
38                  p[e.t]=G[p][i];
39                  Q.push(heap(d[e.t],e.t));
40              }
41          }
42      }
43  }
```

而 Dijkstra's Algorithm 不能處理負邊，原因是一旦點加入最短路徑樹，就不會再被更新，以維持良好複雜度，負邊會破壞此規則。

5.8.6 題目

1. UVa 534
2. UVa 10048
3. UVa 929(方格上)
4. UVa 11090