2020 台南一中第二次資訊段考段考複習

編輯序

大家好，我是編輯此份講義的作者 Hsuan，我想，在這份講義開始前，有些話想先跟大家說，大家都已經學習完一學期的資訊課了，相信大家對於 C++、部分演算法已經有一定了解了，不論你喜不喜歡資訊這一科，希望大家都抱著學習的熱忱努力學習完這些東西，若有不懂的，歡迎像資訊社各個社員請教，務必將這些不會的基礎學習好，有些東西是到你要用到的時候，你才發現到它的重要性，大家不是每一個人都要選資奧國手，你們大可不必學習那些艱深的演算法，但學校老師教的都是基礎，這些基礎的東西，其實對未來的你們很有用，再何況，我們學校已是南部學校中資訊算是強的學校了，這些東西學會了，你們相較於其他人就更有競爭力，所以，請大家，不論你抱持著甚麼態度讀這份講義，如果有問題一定要問，不要害怕問，不問一定不會進步。

本份講義拿掉了部分第一次段考段考複習講義的內容，如果有需要請再向我索取，這裡留下的都是我認為更重要的，比起第一次資訊段考段考複習講義，此份講義多了許多關於演算法的部分，內含 Chyen 大電神的教學講義，十分艱澀，請酌量服用。

這份講義由 Hsuan、Wei、Chyen 共同合力撰寫完成。

謹此 Hsuan 2020.06.30

基礎注意事項

1. 變數型態

下表整理了常見的變數型態

類別	變數型態	範圍
整數	long long	±9223372036854775807
正整數	unsigned long long	18446744073709551615
小數	float	有效位數 7 位
小數	double	有效位數 15 位

unsigned 表示無號，若使用他，必須確保數字非負數。

必須注意: main 一定要用 int 或 signed

建議會用#define 的人，`#define int long long` 以避免不必要的麻煩，

對於需要用到純 int 的東西就用 signed 吧!

例如: signed main(){}

2. 輸入優化

如下所示:

```
int main(){
  ios::sync_with_stdio(0);cin.tie(0);

  return 0;
}
```

建議直接背起來，考試時每一份 code 都加進去，需注意的是，使用了這個程式碼之後，printf/scanf 和 cout/cin 不能混用喔!

經過測試，加入輸入優化的程式碼在極大測資的狀態下，可以達到時間減少 3-4 成，真的值得一背。

3. 小數輸出控制

敢不會寫就慘了，小數輸出位數控制有兩種方法

假設有一個小數 a，控制輸出至小數點後第 2 位。

- `cout << fixed << setprecision(2) << a << endl;`
- `print("%.2f\n",a)`

4. 因數判斷

因數判斷: `a%b == 0` (b 為 a 的因數)

5. 迴圈

for 迴圈必考，謹記語法。

while 迴圈也是相同概念。

for(初始值;條件;改變量)

while(條件)

6. 交換兩個變數

有兩個變數 a,b，今天要交換 ab 使 a 為 b，b 為 a

簡單版:

```
swap(a,b)
```

三行版

```
int _ = a;a = b;b = _;
```

題外話:

阿那我們今天如果要把一個字串轉過來怎麼做?

有下列兩種方法供參考

- `str.reverse()`
- 用 for 迴圈，搭配 `swap(str[i],str[n-i])`

7. 字串轉換為數字

假設今天有個 `str = "123456"`

我們可以透過 str[0] - '0' 直接將第一個 1 轉為數字一

8. 詢問範圍

詢問範圍 A-B 的題目，A 有可能比 B 大，要 swap

9. 多筆測資

題目如果寫到:有多筆測資，請記得使用 while 迴圈+cin 去寫，避免拿不到分

迴圈使用範例

星星樹的解法

星星樹乖乖用雙層迴圈寫太累人了，我不教簡單一點的解法真的對不起自己。

用下面這個函式，能產生一個重複 N 次的字串，不用問為甚麼，拿的到分比較重要，他就是這樣用的。

string(次數, 字元)

星星樹就是要多練，多多練習，寫起來就會比較快，像是我們在程式設計贏的時候每天在寫星星樹，因此，到最後都變得很很很很熟練了。

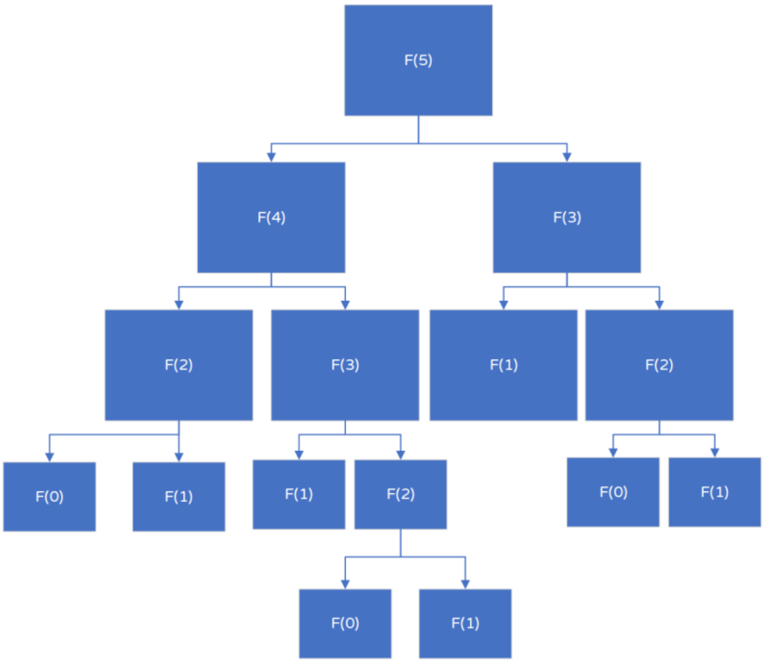
例如這一題感覺超難的題目，用我上面的解法會快很多。

https://toj.tfcis.org/oj/pro/110/ 自己去看題目喔

```
#include<iostream>
using namespace std;
int printOut(int height) {
    /* 第一行 */
    cout << string(height-1,' ') << "*" << "\n";
    /* 中間 N 行 (N=h-4) */
    int nowStar=1;
    for(int i=0;i<height-4;i++){
        cout << string(height-i-2,' ') << string(nowStar+2,'*') << "\n";
        nowStar=nowStar+2;
    }
    /* 中間的上下兩行 (h-1)*2+1 */
    cout << string((height-1)*2+1,'*') << "\n";
    /* 中間 (h-1)*2-1 */
    cout << ' ' << string((height-1)*2-1,'*') << "\n";
    /* 中間的上下兩行 (h-1)*2+1 */
    cout << string((height-1)*2+1,'*') << "\n";
    /* 中間 N 行 (N=h-4) */
```

```
    int nowStar2=1+2*(height-4)+2;
    for(int i=0;i<height-4;i++){
        cout << string(i+3,' ') << string(nowStar2-2,'*') << "\n";
        nowStar2=nowStar2-2;
    }
    /* 最後一行 */
    cout << string(height-1,' ') << "*" << "\n";
    return 0;
}
int main() {
    //110
    int input;
    cin >> input;
    for(int i=0;i<input;i++){
        int a;
        cin >> a;
        printOut(a);
    }
    return 0;
}
```

附加:費氏數列的解法



◆ 陣列建表法

```
int Fi[100000+5] = {0};
```

```
int main() {
    Fi[0] = 0;Fi[1] = 1;
    for(int i=2;i<100000;i++){
      Fi[i] = Fi[i-1] + Fi[i-2];
    }
    int ask;cin >> ask;
    //第 N 項
    cout << Fi[ask] << " ";
    //前 N 項
    for(int i=0;i<ask;i++)cout << Fi[i] << " ";
    return 0;
}
```

- 遞迴法

```
int dp[100000+5] = {0};
int Fi(int n) {
  if(n == 1) {
      return 1;
  } else if(n <= 0) {
      return 0;
  } else {
      if(!dp[n]){
          int x = Fi(n-1) + Fi(n-2) ;
          dp[n] = x;
          return x;
      }else{
          return dp[n];
      }
  }
}
int main() {
  for(int i=0; i<10; i++)cout << Fi(i) << " ";
  return 0;
}
```

陣列 (by Wei)

陣列是一個鏈狀的資料結構。但究竟和一般的變數差別在那裡呢？

1. 陣列可以用相同的變數名稱建立一個大型的空間存放資料。

2. 陣列跟數學的集合似乎差異不大。

建立陣列：

資料結構型態 變數名稱[資料大小];

EX: `int Array[100];` 命名一個叫 Array 的陣列且共有 100 個儲存空間（索引值從 0 到 99）

使用陣列：

1. 於初始化時可順便賦值

EX：`int Array[8] = {1,2,5,4,8,6,2,74};`

2. 在設定變數大小後隨意更改項目

EX：`int Array[8];`

```
Array[0] = 1;Array[1] = 2;…
```

Array[0] 就好比是一個變數可以隨意更改

實際使用：

```
int Array[100];
int N;cin >> N;
for(int i = 0;i < N;++i)cin >> Array[i];
```

N 為一個初始值，在這個程式中 N 不可大於 100。

如此一來，利用陣列我們就可以很輕鬆地把資料儲存起來，不再需要大量的變數了。

DP (by Hsuan、參考 SA、Algorithm Note)

DP 的核心精神就是將大問題切分為小問題，透過小問題去解決大問題。

以階層為例，假設今天要算兩個階層，5! 和 8!，那麼你會發現，5!=5*4*3*2*1，8!=8*7*6*5*4*3*2*1，你可能會想，那就用一個 for 迴圈慢慢乘下去就好了，但這樣子會導致執行的時間太久，有時候會 TLE，我們可以考慮開一個陣列 dp，dp[n]就存 n 階層，那麼就只需要用 for 迴圈跑過一次，每一次將 dp[n-1] * n，再存到 dp[n]裡面，這樣就可以用比較節省時間的方式計算(算過的不重算)。

DP 還有很多經典的題目，但由於這是段復講義，而且這次段考應該考不多吧!可以下次再仔細研究，這個章節只是為了給你一種「想法」。

Ref : https://bit.ly/TNFSHinfo

Sort (by Hsuan)

這裡的 sort 只教程式碼，不教理論的部分，為了速成(?

需要引入#include<algorithm>這個標頭檔

假設今天有一個名字叫做 Array、長度為 5 的陣列[3,9,4,2,1]

若我今天要把它變成[1,2,3,4,9]，你可以使用

```
sort(Array,Array+n);
```

若我今天要把它變成[9,4,3,2,1]，你可以使用

```
sort(Array,Array+n,greater<int>);
```

就是這短短的一行程式碼，可以用來排列陣列，而甚麼時候會用到，那就仔細看題目

的詢問吧!

大數加法(by 玹)

甚麼是大數?當一個數字超過程式語言內建型態能夠裝下的範圍，那我們就稱他為大數，像是在 C++中若一個數字超過 long long 的範圍，那麼他就是大數，這種數值該怎麼處裡呢?讓我們繼續看下去。

我們回想一下，國小老師好像有教過下面這個東西吧!

$$\begin{array}{r} {}^{1} \\ 27 \\ +\ 59 \\ \hline 86 \end{array}$$

沒錯，就是直式加法，大數運算的想法就跟直式差不多，用陣列表示一個數字，我們讀入一個字串，將第一位做為第一位，每次加東西上去時，從第一位開始，超過十就要在下一位進位，由此一來就可以逐步將整個結果算出來。

我也不知道會不會考せ

```
int Result[100+5],A[100+5],B[100+5];
```

或許我們可以先宣告陣列，將數字一一存進去

```
for(int i=0,j=a.length()-1; j>=0; i++,j--)A[i] = a[j] - '0';
for(int i=0,j=b.length()-1; j>=0; i++,j--)B[i] = b[j] - '0';
```

兩個 for 迴圈可以解決存入問題

接下來是加法的部分，每超過十就要記得進位喔

```
int x = 0;
for(int i=0; i<=102; i++) {
        Result[i] = A[i] + B[i]+x;
        x = Result[i] /10;
        Result[i] %= 10;
}
```

乘法的話，大家可以自己去 Google 一下，應該也不會太困難。

當然，並不是每一次都要這麼麻煩的寫，偉大的 Python 沒有整數範圍的限制，只要使用宣

告便數就可以直接存入很大很大的數字，請看下面範例

```
a,b = map(int,input().split())
print(a+b)
```

Python 的程式碼簡潔，運算上也很方便，上面的程式碼等價於給你 a b 求 a+b，偷偷告訴你，聽說 Sky 可以用 Python 喔

但 Python 有些東西就是大家不熟悉的啦

例如說如果有多筆測資要 while cin 的時候，Python 的寫法會像下面那樣

```
while 1 :
    try:
        …
    except EOFError:
        break
        pass
```

如此一來，在…裡面寫程式就可以解決多筆測資的問題了

阿注意一點，Python 沒有{}的概念，因此，在使用時必須小心縮排，每個不同的區塊要用 tab 分隔，像是如果在 while 迴圈裏面，每一行程式前面都要加一個 tab

阿你會不會在考試上實際用到 Python 就看個人造化程度了，連假說不定可以學一下(?

最短路徑演算法(by chyen)

獨家授權，謝謝電神

- 一些名詞

圖的邊有時候會有一些比較特殊的情形。

負邊 ： 邊權為負的邊

負環 ： 圖中有一個環，且這個環上的邊權和為負。

擁有此性質的邊沒有最短路徑，原因是:

當某點對的路徑上經過負環，只要一直繞著環走，路程(花費)就可以變得無限小。

而我們可以透過一些方法來判斷此圖是否有負環。

我也不知道會不會考せ

- 初始化

dis[x]為詢問的點 i 到 x 的最短路徑，

而一開始我們設 dis[x]=infinite。唯獨詢問的點 i，設 dis[i]=0，畢竟應該不會有點到自己本身距離不是 0 的吧?

小小編補充: 通常我們用 memset 來初始化，Inf 無限大設為 0x3f3f3f3f 請見下方範例

```
memset(dis,0x3f3f3f3f,sizeof(dis))
```

- 鬆弛

可以看成更新最短路徑的動作。大概就是對一個相鄰點對(u,v)做

```
dis[v] = min(dis[v],dis[u]+w(u,v))
```

這裡的 w(u,v)代表 u 跟 v 的邊權。
意思就是 ：dis[v] = min(原本紀錄的最短路，經由 u 點再到 v 點的最短路)。

1. floyd-warshall

是一種可以一次處理所有點對的最短路徑演算法。
小小編: 是一種很慢的最短路徑演算法，小小編在高一排名賽時，原本有一題已經想出來怎麼寫了，一看就是最短路徑裸提，結果我只學會這種演算法，然後他是一個 O(n^3)演算法，害我一直 TLE Q_Q

- 想法

枚舉 3 個點 i,j,k，dis(i,j) = min(dis(i,j), dis(i,k)+dis(k,j))。
存邊方法比較特殊，用 dis[i][j]表示 n 到 m 的最短路徑。

- code

```
const int INF = 1e9+10;

void init(int V=105){
    for(int i=1;i<=V;i++){
        for(int j=1;j<=V;j++){
            if(i==j)dis[i][j] = 0;
            else dis[i][j] = INF;
        }
    }
}

void floyd(int n){
    for(int k=1;k<=n;k++){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                dis[i][j] = min(dis[i][j],dis[i][k]+dis[k][j]);
            }
        }
    }
```

```
}
```

時間複雜度為 O(V^3)，空間複雜度為 O(V^2)，只適合用在節點數較少的圖上。

聽到沒!!!

2. dijkstra algorithm

超快。

缺點:無法解決有負邊的圖。

- 想法

建立兩個集合 S(已找出最短路的節點)、Q(還沒找出)。

先從起點開始當作 i，將走過跟 i 直接相通(不經過其他節點)的點做更新。

做完後從 Q 中選擇離起點最近的點，做為下一個 i。而這個 i 即為起點到 i 點的最短路徑，重複上面步驟，直到 Q 為空集合。

因為 i 跟起點的距離為 Q 當中最小的，所以沒辦法在 Q 當中找到經由 x 到達 i 的其他更短路徑，故為最短路徑。

至於如何找到 Q 當中最短的路徑呢，用 O(V)搜尋每個點太慢了，所以把每個經過更新路徑變得更短的推進 priority_queue(或 set)，就可以在 O(log V)找到最近的點了。

- code

```
const int INF = 1e9+10;
long long dis[10010];
vector< pair<int,int> >v[10010];
bool vis[INF];
void dijkstra(int x){
    priority_queue< pair<int,int>, vector< pair<int,int> >, greater<
pair<int,int> > >pq;
    pq.push({0,x});
    while(!pq.empty())
    {
        int p = pq.top().second; pq.pop();
        if(vis[p])continue;
        vis[p] = true;
        for(auto i:v[p])
        {
            if(dis[i.first]>dis[p]+i.second){
                dis[i.first] = dis[p]+i.second;
                pq.push({dis[i.first],i.first});
            }
```

```
        }
    }
}
```

- 在複雜度的部分:

每個點照理來說只會被當作 i 一次，所以總共會推出來 V 次，乘上 priority_queue 操作的複雜度為 VlogV。在推出來 i 後，還要遍歷跟 i 直接相鄰的點，時間複雜度 O(E)。
回到實做，每個點有可能不只被推進去一次，最多會推進 E 次，一樣配合 priority_queue 操作時間複雜度，總共為 ElogV。
所以整體時間複雜度為 O((E+V)logV)。

- 至於為什麼無法處理帶負邊的圖:

dijkstra 演算法假設的是:在集合 S 的點已經為最短路徑了，而如果存在負邊，則此性質則有可能出錯。

3. spfa(shortest path faster algorithm)

bellamn_ford 的優化版。

- 想法

在對 u 點做鬆弛時，如果 u 點的最短路徑沒被更新，那麼在下次鬆弛時，跟 u 相鄰的 v 在鬆弛時，因為 u 的路徑沒被更新，所以 v 的最短路也不會經由 u 被更新。
根據這個性質，我們可以過濾掉那些沒更新的點，只對有更新的點在下去做鬆弛就好了。
而可以用 queue 來裝那些要鬆弛的點。
這樣看起來根本就是 BFS 了吧

- code

```
void spfa(int x){
    queue<int>q;
    q.push(x); inque[x] = 1;
    while(!q.empty()){
        int p = q.front(); q.pop(); inque[p] = 0;
        for(auto i:v[p]){
          if(dis[i.first]>dis[p]+i.second){
            dis[i.first] = dis[p]+i.second;
            if(!inque[i.first]){
              q.push(i.first);
              inque[i.first] = 1;
```

```
            }
        }
      }
    }
}
```

就像 bellman-ford 一樣可以偵測負環，spfa 也一樣可以。

在沒有負環的圖裡面，每個點的最短路徑最多只需要 V-1 條路，所以如果有個點的最短路徑需要的>=V，代表他會繞著負環走。

我們只需要紀錄每個點最短路徑需要幾條路，就可以判斷是否有負環了。

- code

```
bool spfa(int x){
  queue<int>q;
  q.push(x); inque[x] = 1;
  while(!q.empty()){
    int p = q.front(); q.pop(); inque[p] = 0;
    for(auto i:v[p]){
      if(dis[i.first]>dis[p]+i.second){
        dis[i.first] = dis[p]+i.second;
        len[i.first] = len[p]+1;
        if(len[i.first]>=n)return false;//有負環
        if(!inque[i.first]){
          q.push(i.first);
          inque[i.first] = 1;
        }
      }
    }
  }
  return true;
}
```

複雜度為 O(kE)，這裡的 k 是指每個點進出 queue 的次數，所以最好的情況為 O(E)，但最壞情況會是 O(VE)，還是跟 bellamn_ford 一樣。

同餘(by Hsuan)

(a*b)%c=(a%c)*(b%c)%c

同餘的整個概念大概就上面那樣吧，有時候題目會給你大數 a，然後一個很大的 b 叫你求 a 的 b 次方 mod 1e9+7，這種時候該怎麼辦呢?讓我們來看看吧

我也不知道會不會考せ

```
for (int i = 0; i < num.length(); i++)
     a = (a * 10 + (num[i] - '0')) % mod;
for (int i = 0; i < num2.length(); i++)
     b = (b * 10 + (num2[i] - '0')) % (mod - 1);
while (b) {
     if (b & 1)
          res = res * a % mod;
          b = b / 2;
          a = a * a % mod;
     }
     cout << res << endl;
}
```

讓我們看看程式碼吧!

首先因為他是大數嘛，想當然爾，當然是用字串去儲存他。

我們將每一個數字跑過，將其轉換成數字，接下來，對他取模，再加上下一位，一直持續下去，最後，我們可以用類似快速冪的想法將取模過後的數字求出。

分治法(by Chyen)

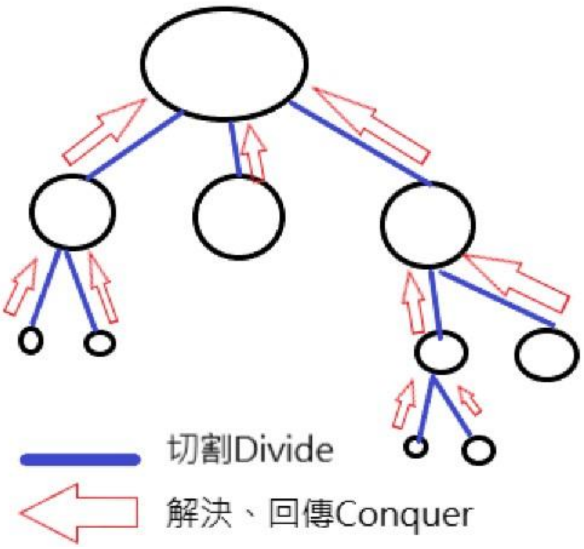
分治法是一種演算法設計方式，並非演算法，所以基本上它沒有固定的 code 可以參考，它只是一種概念，協助你在解題時朝這方向想，然後想出解法。

分治就跟字面上的意思一樣，就是「分而治之」。

利用遞迴，將較大規模的問題，化為數個小規模的子問題，並且將這些小規模問題的答案合併求出答案。

- 分治法的步驟大約如下:
  - 把大問題分割成數個小的子問題(Divide)
  - 如果小問題可以直接解決，解決並回傳;反之則再次分割，直到可以直接解決

■ 將所有解決的答案回傳，組成原問題的答案(Conquer)



● 可以用分治法的問題的性質:

小到一定程度的問題可以解決
每個子問題各自獨立，不影響其他問題
可以利用子問題的答案構成原問題的答案
比一般解法更有效率(不然幹嘛沒事寫遞迴)、或者只能用分治解決

● 一些例子

■ Merge_sort 合併排序
一個神奇的排序法，
概念是將一個數列分割成兩個，割出來的子數列再一直割成兩個子數列，直到每個子數列只有一個數，再將其合併。

將每兩個子數列合併時，依大小排序合併，直到全部合併成原本大小的數列，就是排序完數列了。

◆ 合併過程:

將兩個數列{2,5}、{3,4,8}進行合併時

比較 2、3(兩個數列的頭)，2<3，先把 2 放進合併的數列

2	5	
3	4	8

合併的數列	2

比較 5(2 的下一個數)、3(原本的數)，5>3，把 3 放進合併的數列

2	5	
3	4	8

合併的數列	2	3

比較 5(原本的數)、4(3 的下個數)，5>4，把 4 放進合併的數列

2　5

3　4　8

合併的數列	2	3	4

比較 5(原本的數)、8(4 的下個數)，5<8，把 5 放進合併的數列

2　5

3　4　8

合併的數列	2	3	4	5

剩下 8 沒被放進合併的數列，把 8 放進去

2　5

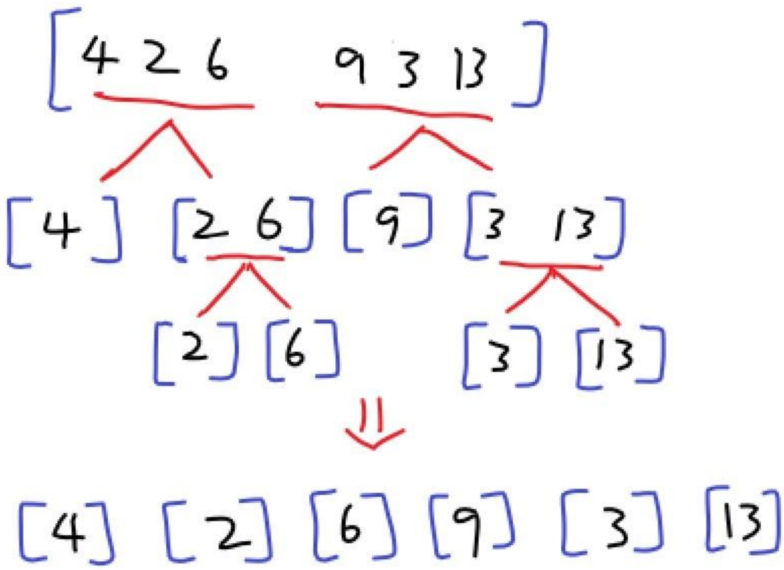
3　4　8

合併的數列	2	3	4	5	8

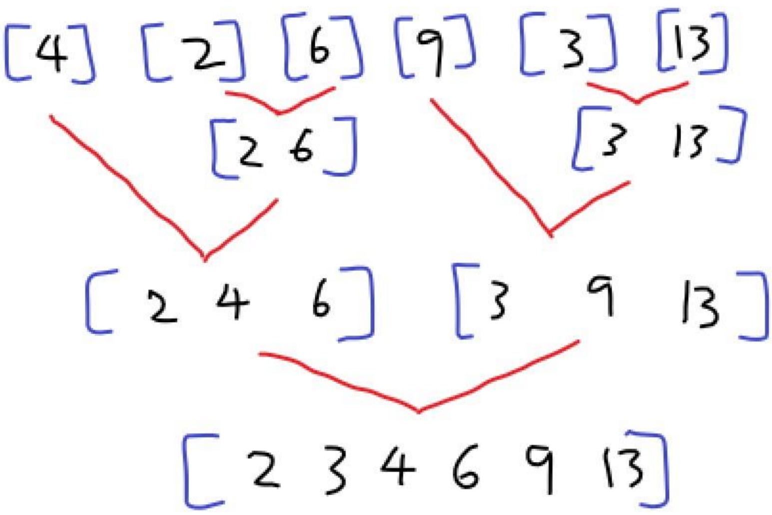
合併完的數列再跟另一個數列做合併時，因為兩個數列都排序過了，所以也只需要用上面的方法合併即可。

舉例:將{4,2,6,9,3,13}做合併排序。

- 第一步:分割



- 第二步:合併



這樣子排序有什麼優點呢? 快，很快，比一般的排序法快上非常多。

在分割的部分，大約會分割 N 次(總共分割成 N 個子數列)，
而每次分割都割成兩份，我們可以把分割的過程看做一棵樹，這棵樹的高度約為以 2 為底的 logN(+1)。

合併時，每層約會做 N 次比較並合併，而總共有 logN 層，所以會花 NlogN 次運算合併。

總共執行差不多(NlogN)+N 次運算，時間複雜度為 O(NlogN)。
比一般的 sort(像是 Bubble_sort、Selection_sort)的 O(N^2)快很多。

- 程式碼

```
int ary[200010];
int tmp[200010];

void merge(int l,int r){
    int mid = (l+r)/2;
    int pl = l,pr = mid+1;//兩個要比較的位置
    for(int i=l;i<=r;i++){
        if(pl>mid){//左邊的數列的數都已放進合併的數列
            tmp[i] = ary[pr]; pr++;
        }
        else if(pr>r){//右邊的數列的數都已放進合併的數列
            tmp[i] = ary[pl]; pl++;
        }
        else if(ary[pl]>ary[pr]){
```

```
                tmp[i] = ary[pr]; pr++;
            }
            else{
                tmp[i] = ary[pl]; pl++;
            }
        }
          //先用tmp[]裝合併後的數列
        for(int i=l;i<=r;i++)ary[i] = tmp[i];
          //再複製回原本的數列
    }

    void merge_sort(int l,int r){//l:陣列的頭位置，r:陣列的尾位置
        if(l==r)return;//數列只有一個數，不用再分割
        int m = (l+r)/2;
          //分割成兩個子數列
        merge_sort(l,m);
        merge_sort(m+1,r);
          //合併
        merge(l,r);
    }
```

- 河內塔

  相信這個很有名，大家也應該都聽過(數學課上到遞迴時)，不知道的可以看一下這裡。(題目:TIOJ1355)

  在移動 N 個碟子時，必須先把上面 N-1 個碟子移開，才能移動最下層的碟子。
  所以我們可以把上面 N-1 個碟子從 A 柱移至 B 柱，把最下面的碟子從 A 柱移至 C 柱，最後再將 B 柱的碟子移至 C 柱。

  而如何將 N-1 個碟子從 A 柱移至 B 柱?
  做法也跟上面的一樣:上面 N-2 個碟子從 A 柱移到 C 柱，最下面的碟子從 A 柱移到 B 柱，N-2 個碟子從 C 柱移回 B 柱。
  所以問題就變成如何移動 N-1、N-2、N-3…個碟子。
  直到只剩一個碟子要移動，就直接移動至目標柱子就好了。

  - 程式碼

```
#include<iostream>
void move(int l,int u,char A,char B,char C){
    //把第 l 個到第 u 個從 A 移到 C
```

```
    if(l==u){
        //只有一個要移
        cout << "move the dish from " << A << " to " << C << endl;
  return;
    }
    move(l+1,u,A,C,B);//把上面N-1個從A移到B
    move(l,l,A,B,C);//最下面的從A移到C
    move(l+1,u,B,A,C);//把移到B的碟子移到C
}
int main(){
    int N; cin >> N;
    move(1,N,'A','B','C');//有N個碟子
    return 0;
}
```

複雜度的部分，可以用數學的遞迴算出，有 N 個碟子，需要移動 2^n -1 次，複雜度為 O(2^n)，至於怎麼算的可以參考這裡。