APCS 2018 年 6 月實作題檢測
題目分析與解答 (草稿，更一版)

吳邦一 20181221

前言：

本題解以教學的角度來分析說明APCS實作題並提供範例解答程式，主要目的是讓程式的初學者能夠學習到面對題目時如何設計程式來解決問題。這裡所謂初學者定義為：對程式的基本指令已經有所了解的學習者。所以文件中並不會對指令做太多的解釋，另一方面，也因為對象是初學者，本文件提供的解答方法與範例程式，盡量都是只使用基本指令與常用的庫存函式，並不會使用複雜的資料結構與庫存函式(例如C++ STL)。APCS允許使用的語言為C、C++、Java與Python，由於前三者語法相近，文件中的範例程式碼將提供C/C++與Python(3)兩種版本。

2018年6月APCS實作題有四個題目，以下依照題目順序編排。每一個題目先列出原始題目敘述，然後有題目分析、程式實作與Python程式實作三個部分，在題目分析中我們會分析題意以及解題的重點，進而說明解題的方法。在程式實作單元，我們會說明如何使用C/C++的程式語言來實作出前一單元的解題方法，必要時會分段講解，講解完畢後提出範例的解答程式。在Python程式實作單元裡，則說明使用Python應該注意的事項以及提供Python的範例解答。

參考本文件時有下列需要留意的事項：

1. 因為每個人的思考方式不盡想同，任何一個題目都可能有多種的解法，本說明文件對於每一題提出可能的解法，並不代表沒有其他的方法也不代表本文件的解答是最好的。
2. 撰寫程式時除了程式的正確性之外，也應該要注意好的程式風格，例如：有意義的變數名稱與適當的註解以提高程式可讀性、結構化的程式、以及避免濫用全域變數。不過，在考試的時間壓力下，通常我們都會忘掉這些事，畢竟，考試的程式不是大型軟體，甚至大部分都很簡短，快速的答對才是最重要的事。本文件既然以教學為目的，當然應該提供好的程式風格的解答程式，讀者應了解：在考試時，使用簡短變數名稱，沒有註解，使用全域變數都是人之常情、無可厚非、理所當然的事情。
3. 正規來講，撰寫程式前應該先對所設想的方法分析複雜度，但APCS的實作題大部分都沒有太講究程式效率的問題，因此，除了必要的題目，本文件將略去時間空間的分析，以避免初學者反而被一些名詞困擾。
4. APCS的考題通常有若干子題，這些子題的差異通常在於輸入資料的範

圍與特性，相同題目不同的輸入會有不同的難度，但是原則上一定有一個完全解的解法可以解所有子題，也就是說，會的人不必使用兩個方法，但是子題提供想不出完全解的人得到部分分數。本文件在分析題目時，會先分析各子題的差異以及所對應的解決方法，但是為了閱讀流暢，在子題差異分析完畢後會以完全解為最主要的說明目標，重要的子題範例程式則在最後在予以補充納入，但是某些非常簡單的子題就不會單獨說明。

第 1 題 特殊編碼

問題描述

任何文字與數字在電腦中儲存時都是使用二元編碼，而所謂二元編碼也就是一段由 0 與 1 構成的序列。在本題中，A~F 這六個字元由一種特殊方式來編碼，在這種編碼方式中，這六個字元的編碼都是一個長度為 4 的二元序列，對照表如下：

字元	A	B	C	D	E	F
編碼	0 1 0 1	0 1 1 1	0 0 1 0	1 1 0 1	1 0 0 0	1 1 0 0

請你寫一個程式從編碼辨識這六個字元。

輸入格式

第一行是一個正整數 N，$1 \le N \le 4$，以下有 N 行，每行有 4 個 0 或 1 的數字，**數字間彼此以空白隔開**，每一行必定是上述六個字元其中之一的編碼。

輸出格式

輸出編碼所代表的 N 個字元，字元之間不需要空白或換行間格。

範例一：輸入
```
1
0 1 0 1
```

範例一：正確輸出
```
A
```

範例二：輸入
```
1
0 0 1 0
```

範例二：正確輸出
```
C
```

範例三：輸入
```
2
1 0 0 0
1 1 0 0
```

範例三：正確輸出
```
EF
```

範例四：輸入
```
4
1 1 0 1
1 0 0 0
0 1 1 1
1 1 0 1
```

範例四：正確輸出
```
DEBD
```

評分說明

輸入包含若干筆測試資料，每一筆測試資料的執行時間限制均為 1 秒，依正確通過測資筆數給分。其中：

第 1 子題組 50 分：$N = 1$。

第 2 子題組 50 分：$N \le 4$。

題目分析：

在本題中定義了ABCDEF六個英文字母的某種特殊編碼，每個字元由四個0/1數字來代表，輸入代碼，你的程式要分辨出是六個中的哪一個字元。此外第一行輸入的數字N是說明下面有多少行要處理，每一行是一個要辨識的代碼。

子題分析：

本題有兩個子題，其差異在輸入數字N，第一子題只有一行輸入辨識一個代碼，第二子題可能有1~4個代碼要辨識，很明顯的，第一子題只要寫判斷的部分，完全解則要加上一個外迴圈。

方法與流程：

這是一個相當簡單的題目，基本上只是分支指令(如if)以及判斷式的運用，第2子題再加上迴圈的基本運用。基本的演算流程式這樣的：

```
輸入數字N;
控制一個迴圈執行N次
    讀入四個0或1的數字;
    判斷讀入的代碼是哪一個字元必輸出;
迴圈結尾
結束
```

這一題的重點是在判斷代碼的部分，這個部分有很多方式都可以來做，以下說明兩種，假設讀入的代碼存放在x1, x2, x3, x4：

1. 直接列舉每一個字元的四個代碼，用邏輯的and來連接條件。例如A的條件就是
   if (x1==0 && x2==1 && x3==0 && x4==1)
2. 使用多層的if指令，每次用一個if來將要分辨的對象區分成兩類，直到剩下一種情形。例如，第一層的if可以用來判斷：x1如果是0則是{A,B,C}其中之一，否則是{D,E,F}其中之一。
3. 將代碼轉成10進位的數字後判斷。

細節在下面程式實作單元詳述。

程式實作：

首先，讀入N並控制一個迴圈執行N次，我們先把外層架構寫好：

```
scanf("%d",&n);
for (i=0; i<n; i++) { // 也可以用while (n--) {
    // 輸入一行4個數字
    // 判斷代碼代表的字元並輸出
}
```

接著說明如何實作迴圈內部的判斷，先說明第一種：列舉每一個字元的4位代碼。這種方式非常直接，只要把每個字元的每一位都用邏輯的「且」連接起來，就像下面的範例程式。

```
1  #include<stdio.h>
2  int main() {
3      int i,n,x1,x2,x3,x4;
4      scanf("%d",&n);
5      for (i=0;i<n;i++) {
6          scanf("%d%d%d%d",&x1,&x2,&x3,&x4);
7          if (x1==0 && x2==1 && x3==0 && x4==1)
8              printf("A");
9          else if (x1==0 && x2==1 && x3==1 && x4==1)
10             printf("B");
11         else if (x1==0 && x2==0 && x3==1 && x4==0)
12             printf("C");
13         else if (x1==1 && x2==1 && x3==0 && x4==1)
14             printf("D");
15         else if (x1==1 && x2==0 && x3==0 && x4==0)
16             printf("E");
17         else if (x1==1 && x2==1 && x3==0 && x4==0)
18             printf("F");
19     }
20     printf("\n");
21     return 0;
22 }
23
```

第二種方法是使用多層的if指令，不斷地將要分辨的對象區分出來。這個方法其實是在建一棵決策樹，因為資料很小，不需要考慮效率的問題，我們以最直接從前面開始看的方式來實作，請看下面的範例程式。首先以x1來區分，x1==0的有{A,B,C}；x1==1的則是{D,E,F}，下面的程式中第9~13行是{A,B,C}的情形，要區分{A,B,C}我們以x2來判斷，x2==0的是{C}，否則是{A,B}，對於後者我們在用x3把A與B區分出來。對於第16~20行{D,E,F}的情形也是類似，請自行研究，當然分辨的方式有非常多種。

```
        if (x1==0) {
            if (x2==0) ch='C';
            else {
                if (x3==0) ch='A';
                else ch='B';
            }
        }
        else {
            if (x2==0) ch='E';
            else {
                if (x4==0) ch='F';
                else ch='D';
            }
        }
        printf("%c",ch);
```

以下說明第三種方法：把代碼看成二進位的數字，轉換成10十進制的數字後再判斷。這個作法需要一點二進位數字的概念，由於代碼只有四位，我們也不需要迴圈，可以直接計算，請看下面的範例，這裡我們運用switch指令來做分支判斷，當然，如果用if指令也是可以。

```
#include<stdio.h>
int main() {
    int i,n,x1,x2,x3,x4;
    scanf("%d",&n);
    for (i=0;i<n;i++) {
        scanf("%d%d%d%d",&x1,&x2,&x3,&x4);
        switch (x1*8+x2*4+x3*2+x4) {
            case 5: printf("A"); break;
            case 7: printf("B"); break;
            case 2: printf("C"); break;
            case 13: printf("D"); break;
            case 8: printf("E"); break;
            case 12: printf("F"); break;
        }
    }
    printf("\n");
    return 0;
}

```

Python的程式設計：

Python語法雖然跟C有很大的不同，不過在寫這些小程式的時候，架構沒有太大的差別，以這一題來說，if與迴圈指令和C的寫法很像。初學者使用Python要先注意輸入的問題，Python的輸入是以一整行輸入為主，讀進來的

東西是個字串，必須要把其中的數字分拆出來，有些固定的使用方式，可以當片語記起來用。以下是這個題目的範例程式，是使用上面所說的第二種判斷方式：

```
n=int(input())
out=''  # output string
for i in range(n): # 執行n次
#輸入一行，空白分隔的數字拆開放在一個list (陣列)中
    c=[int(x) for x in input().split()]
    if c[0]==0:
        if c[1]==0: ch='C'
        else:
            if c[2]==0: ch='A'
            else: ch='B'
    else:
        if c[1]==0: ch='E'
        else:
            if c[3]==0: ch='F'
            else: ch='D'
    out+=ch  #字串後面串接一個字元
print(out)
```

第 2 題 完全奇數

問題描述

如果一個正整數的每一位數都是奇數時，例如：7、19、1759977 等，我們稱這種數字為完全奇數。對於輸入的一正整數 N，如果 K 是最靠近 N 的完全奇數，請寫一程式找出 K 與 N 之間差距的絕對值，也就是說，請計算並輸出 $|K-N|$。

以 $N = 13256$ 為例，比 13256 大的最小完全奇數是 13311，比它小的最大完全奇數是 13199，因為 $|13311-13256| = 55 < |13256-13199| = 57$，因此輸出 55。

輸入格式

一個正整數 N，$N < 10^{18}$。

輸出格式

輸出 N 與其最近的完全奇數的差距。

範例一：輸入
```
135
```

範例一：正確輸出
```
0
```

範例二：輸入
```
13256
```

範例二：正確輸出
```
55
```

範例三：輸入
```
35001
```

範例三：正確輸出
```
110
```

範例四：輸入
```
1001
```

範例四：正確輸出
```
2
```

評分說明

輸入包含若干筆測試資料，每一筆測試資料的執行時間限制均為 1 秒，依正確通過測資筆數給分。其中：

第 1 子題組 20 分：$N < 100$。
第 2 子題組 30 分：$N < 10^6$。
第 3 子題組 50 分：$N < 10^{18}$。

題目分析：

在本題中定義了所謂完全奇數就是一個正整數，滿足它的每一位數字都是奇數，對於輸入的正整數n，要找出跟它與最接近的完全奇數的差值。題目的舉例也暗示了先找出

(1)比N大的最小完全奇數(n_next)

(2)比N小的最大完全奇數(n_pre)

再輸出 min(n_next - n, n-n_pre)。因此，問題的核心在如何求出n_next與n_pre。

子題分析：

本題有三個子題，其差異在輸入數字N的大小，但都還在C/C++ long long的範圍之內，這表示最後求差值可以直接用數字運算而不需要大數運算。對於子題一，只有一位或兩位數，可以觀察規律後直接判斷，例如一位數如何，二位數偶數開頭如何，奇數開頭又如何，以下就不多加敘述了。

對於子題二，數字的範圍在一百萬以內，我們可以使用枚舉測試法，在找下一個的時候，我們可以從n開始往下找，將每一個數字都拿來檢查，找到第一個完全奇數就是所要找的下一個，相同的方法也可以拿來找前一個。

對於第三子題就不能使用枚舉了，因為找到下一個或前一個可能需要枚舉10^{17}，現在電腦大約一秒只能處理數千萬個指令，此法顯然無法通過。能夠解第三子題的方法顯然可以解全部的子題，以下的分析將以說明完全解為主，在本題的最後再說明枚舉測試法的範例程式。

方法與流程：

這問題第一個要考的是將一個整數的每一個位數(digit)進行操作，所以要先將一個整數拆解成一個一個digit，我們可以用整數輸入再拆解，更可以直接把它看成字串讀進來。我們先看n_next要怎麼做。

找下一個完全奇數n_next其實非常簡單，以13256為例，我們先從前(左)往後找到第一個偶數「2」，這是一定要改變的位置，因為要比它大，所以我們將這個數字加一，然後後面的位數都盡量放最小的「1」，這樣就得到比它大的最小完全奇數13311。上面的程序會不會有麻煩呢？其實不會，因為找到的偶數最大是「8」，一定可以加一。因為要求差值，所以我們最後要把它轉換回去成數字。

```
計算正整數n的下一個完全奇數 n_next：
    從前往後找到第一個偶數位數 x；
    將 x 改成 x+1 ；
    把它後面的位數都改成 1 ；
    轉換成整數；
```

再來看看如何求前一個n_pre，這比較麻煩。主要的想法和找n_next一樣：「先找到第一個偶數，將它減一，後面的都改成最大的9」，以13256為例，會成功地找到13199。有問題嗎？有的，因為找到的第一個偶數如果是0就無法減一了！這時候的狀況有如做減法時需要借位一樣，我們要把他的前面借位，

把它變成「9」，它的前面如果有數字，一定是奇數(因為剛才找的是第一個偶數)，所以我們要把前面那個數字減二，不能減一，因為要變奇數。例如n=13056，會變成11999。可是如果前面這個奇數是「1」呢？相同的狀況，又要跟前面借位，所以，有可能會需要一直往前借位，直到可以借(碰到一個數字>1)或者已經走到最前頭無位可借。看例子就比較清楚該如何做，如果n=5111056，這個0要減一，要向前借位一直到3才能借，所以n_pre=3999999。如果n=1111056，這時0的前面都是不能借的「1」，到最前面時就直接把第一個1改成0，因為leading 0是不算的，所以找到的n_pre=999999。

```
計算正整數n的前一個完全奇數 n_pre：
    從前往後找到第一個偶數位數 x；
    將 x 改成 x-1 ；
    把它後面的位數都改成 9 ；
    處理借位；
    轉換成整數；
```

註：如果只要解本題，當借位兩位以上時其實可以不用計算，因為這時的最小差值一定發生在n_next-n。讀者可自行驗證，本說明還是完整的說明如何找出前一個的方法。

程式實作：

首先我們構思整個程式的架構，準備把下一個與前一個寫成副程式，所以先寫出主程式的部分，我們把找出第一個偶數的部分寫在主程式中。主程式開始先以字串方式讀入n並求出其長度len，在第39行的迴圈計算出第一個偶數的位置，請注意，雖然是字元的形式，仍然可以用除以2的餘數來判斷其偶，因為字元0的ASCII是0x30=48。

```
31  int main() {
32      int i,len;
33      char n[20];
34      long long n_next,n_pre,n_num,dif;
35      scanf("%s",n);
36      n_num=atoll(n); // convert to long long int
37      len=strlen(n);
38      // the first even digit
39      for (i=0;i<len;i++) {
40          if (n[i]%2==0) break;
41      }
42      if (i==len) { // no even digit
43          printf("0\n");
44          return 0;
45      }
46      n_next=find_next(n,i,len);  // find next all_odd
47      n_pre=find_pre(n,i,len);// find previous all_odd
48      dif=n_next-n_num;
49      if (n_num-n_pre<dif)
50          dif=n_num-n_pre;
51      printf("%lld\n",dif);
52      return 0;
53  }
```

在第42行的判斷式，我們處理輸入的就是完全奇數的情形。除此之外，我們呼叫副程式找出下一個與前一個，然後輸出較小差值。寫程式要注意資料型態，本題的整數範圍會超過C/C++的int，所以必須使用long long的整數資料型態。

接著說明find_next的副程式，傳入的參數為字串n、第一個偶數位置even以及字串長度length。我們先將字串複製一份，因為處理過程會將字串破壞，而我們還需要它。接著就依照上面的分析，將偶數加一，在把後面改成字元1就可以了，如果讀者對第9~10行的迴圈不喜歡，也可以改成比較直接的

for (i=even+1 ; i<len ; i++) temp[even]='1';

```
5   long long find_next(char nstr[],int even,int len) {
6       char temp[20];
7       strcpy(temp,nstr);
8       temp[even]++;
9       while (++even<len)
10          temp[even]='1';
11      return atoll(temp);
12  }
```

最後我們利用庫存函數atoll(asci to long long)將字串轉換成整數後回傳。如果讀者不熟悉字串函數或是考試時不記得怎麼辦？其實只要懂得原

理，這個函數自己寫也很簡單，以下是自製的函數寫法。

```
long long my_atoll(char s[]) {
    long long result=0;
    int i,len=stelen(s);
    for (i=0;i<len;i++)
        result=result*10+s[i]-'0';
    return result;
}
```

接下來說明如何實作副程式find_pre，傳入的參數跟find_next一樣。第18~20行我們直接先將該偶數減一，並把後面設成字元9，接著第21行的while迴圈處理可能的借位問題。當我們發現temp[even]是比0字元還小的時候，就需要借位：把它設成字元9，移到前一位，減二。迴圈在沒有借位或無位可借時停止，第26行處理無位可借的情形。

```
14  long long find_pre(char nstr[],int even,int len) {
15      char temp[20];
16      int i,j;
17      strcpy(temp,nstr);
18      temp[even]--;
19      for (i=even+1;i<len;i++)
20          temp[i]='9';
21      while (even>0 && temp[even]<'0') {
22          temp[even]='9';
23          even--;
24          temp[even]-=2;
25      }
26      if (temp[even]<'0') temp[even]='0';
27      printf("%s\n",temp);
28      return atoll(temp);
29  }
```

把主程式與副程式合起來就是完整的程式了。

Python的程式設計：

基本的程式架構與上面C的程式雷同，只變更成Python語法。我們一樣把n_next與n_pre寫成副程式，主程式的部分如下：

```
n_str=input()
n=int(n_str)
n_dig=[int(x) for x in n_str]
length=len(n_dig)
for i in range(length):
    if n_dig[i]%2==0: break
if n_dig[i]%2==1:
    print(0)
else:
    n_next=next(n_dig,i,length)
    n_pre=pre(n_dig,i,length)
    print(min(n_next-n,n-n_pre))
```

輸入的部分略有不同，n_str是輸入數字的字串形式，n是將它轉成整數，n_dig則是每一個digit的list(可看成陣列)，它的內容是數字不是字元。底下一樣是找出第一個偶數，如果沒有，就直接輸出0；否則就計算下一個與前一個。接著說明副程式，我們這裡自己寫一個將digit list轉換成整數的副程式，因為它不是字串，不能直接轉。

```
def ltoi(n_list): # list_of_digit to int
    result=0
    for dig in n_list:
        result=result*10+dig
    return result
#find next all-odd number
# 314652 =>315111
def next(n_dig,even,length): #even= first even-digit
    n_list=n_dig.copy()
    n_list[even]+=1
    for i in range(even+1,length):
        n_list[i]=1
    return ltoi(n_list)
```

副程式next與前面C的程式碼幾乎相同，只是list中的是數字不是字元罷了。下面則是找前一個的副程式pre，它的寫法也與前述的C程式差不多。

```
# return next all_odd number, even: first even digit
def pre(n_dig,even,length):
    n_list=n_dig.copy()
    n_list[even]-=1
    for i in range(even+1,length):
        n_list[i]=9
    j=even
```

```
    while j>0 and n_list[j]<0:
        n_list[j]=9
        j-=1
        n_list[j]-=2
    #case like 11105 =>09999
    if n_list[j]<0: n_list[j]=0
    return ltoi(n_list)
```

子題一與二的枚舉測試法：

對於子題一與二，我們可以使用枚舉測試法：在找下一個的時候，我們可以從n開始往下找，將每一個數字都拿來檢查，找到第一個完全奇數就是所要找的下一個，相同的方法也可以拿來找前一個。我們要做的是寫一個檢查某數是不是完全奇數的副程式test，請看以下的範例程式：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   // test if n is all_odd
4   int test(long long n) {
5       while (n) {
6           if ((n%10)%2==0) return 0;
7           n/=10;
8       }
9       return 1;
10  }
11  int main() {
12      long long n,n_next,n_pre,dif;
13      scanf("%lld",&n);
14      for (n_next=n; test(n_next)==0 ;n_next++);
15      for (n_pre=n; test(n_pre)==0 ;n_pre--) ;
16      n_next-=n;
17      n_pre=n-n_pre;
18      dif=(n_next<n_pre)? n_next: n_pre;
19      printf("%lld\n",dif);
20      return 0;
21  }
```

第5行的while迴圈每次在第6行檢查個位數是否為偶數，如果是就回傳0表示測試失敗；否則將n除以10，也就是將n向右移動一位。主程式相當簡單，第14行的迴圈找下一個，第15行的迴圈找前一個，然後計算比較小的差值輸出。這個程式的執行時間大約跟N的大小成正比，所以當N達到八或九位數時就可能需要太長的執行時間，讀者可以自行試試。

第 3 題 工作排程

問題描述

有 M 個工作要在 N 台機器上加工，每個工作 i 包含若干個工序 o_{ij}，這些工序必須依序加工，也就是前一道工序 $o_{i(j-1)}$ 完成後才可以開始下一道工序 o_{ij}。每一道工序 o_{ij} 可以用一個有序對 (k_{ij}, t_{ij}) 來表示它需要在機器 k_{ij} 上面花費 t_{ij} 小時來完成。每台機器一次只能處理一道工序。

所謂一道工序 o_{ij} 的「最早完成時間 c_{ij}^*」是指考慮目前排程中機器 k_{ij} 之可用性以及前一道工序 $o_{i(j-1)}$（如果該工序存在）之完成時間後可得的最早完成時間。工廠經理安排所有工序的排程規則如下：

> 針對每一個工作的第一個尚未排程的工序，計算出此工序的「最早完成時間」，然後挑選出最早完成時間最小的工序納入排程，如果有多個最早完成時間都是最小，則挑選其中工作編號最小之工序。一個工序一旦納入排程就不會再更改，重複以上步驟直到所有工序皆納入排程。

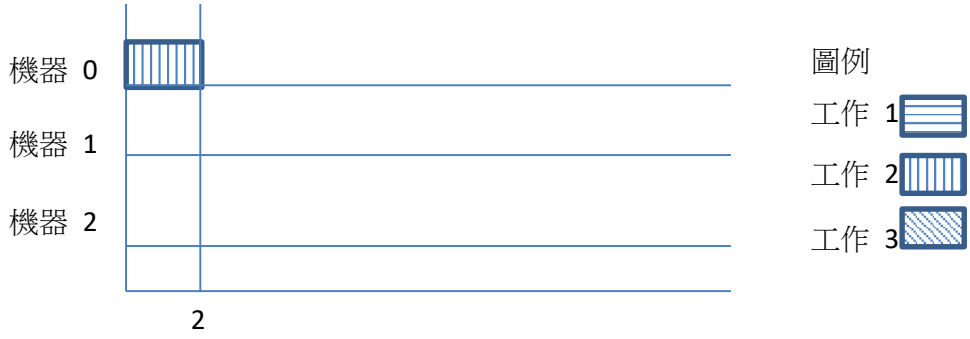
我們總是從時間 0 開始排程，每個工作的完成時間為其最後一個工序的完成時間，本題的目標是計算出每個工作的完成時間並輸出其總和。

以下以一個例子來說明，在此例中，有三個工作要在三台機器上排程，各工作的資料如下。

	工序	說明
工作 1	$o_{11} = (2, 4)$ $o_{12} = (1, 1)$	此工作有兩道工序，第一道需要在機器 2 執行 4 小時，第二道需要在機器 1 執行 1 小時。
工作 2	$o_{21} = (0, 2)$ $o_{22} = (2, 2)$ $o_{23} = (0, 1)$	有三道工序，第一道需要在機器 0 執行 2 小時，餘類推。
工作 3	$o_{31} = (0, 7)$	有一道工序需要在機器 0 執行 7 小時。

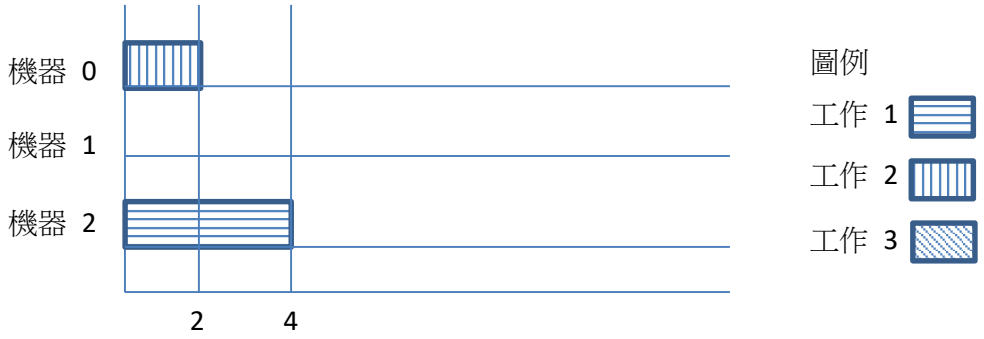
排程的過程說明如下：

1. 在開始時，每個工作都是考慮第一道工序，三個工作第 1 道工序需要的時間分別是 $t_{11} = 4$、$t_{21} = 2$、$t_{31} = 7$，這也是它們的最早完成時間，也就是 $c_{11}^*=4$、$c_{21}^*=2$、$c_{31}^*=7$，因此會先排 o_{21}。

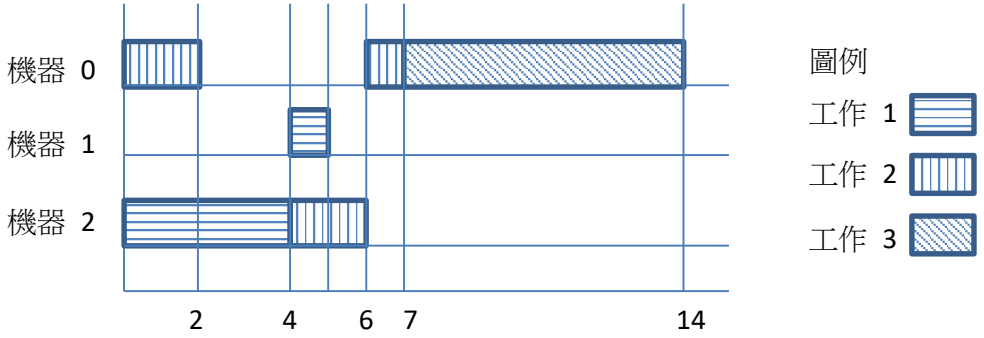


2. 接下來，三個工作要考慮的工序分別是第 1、2、1 個工序，即 o_{11}、o_{22} 和 o_{31}。
   (1) o_{11} 需要機器 2 執行 4 小時，而機器 2 可以開始加工的時間點是 0；o_{11} 沒有前一道工序。因此，這工序可以開始的時間是 $\max(0, 0) = 0$。是故，其最早完成時間 $c_{11}^* = \max(0, 0) + 4 = 4$。
   (2) o_{22} 需要機器 2 執行 2 小時，而機器 2 可以開始加工的時間點是 0；o_{22} 前一道工序 o_{21} 的完成時間是 2。因此，這工序可以開始的時間是 $\max(0, 2) = 2$。是故，其最早完成時間 $c_{22}^* = \max(0, 2) + 2 = 4$。
   (3) o_{31} 需要機器 0 執行 7 小時，而機器 0 可以開始加工的時間點是 2；o_{31} 沒有前一道工序。因此，這工序可以開始的時間是 $\max(2, 0) = 2$。是故，其最早完成時間 $c_{31}^* = \max(2, 0) + 7 = 9$。

   因此，由於 c_{11}^* 和 c_{22}^* 都是最小，根據規則，工作編號小的先排，所以會排 o_{11}。



3. 三個工作目前要考慮的工序分別第 2、2、1 個工序。依照類似的推論，我們可以得到 $c_{12}^* = 5$，$c_{22}^* = 6$，$c_{31}^* = 9$，因此排 o_{12}。工作 1 的工序均已排完，所以它的完成時間是 5。
4. 剩下工作 2 與 3。$c_{22}^* = 6$，$c_{31}^* = 9$，因此先排 o_{22}。
5. $c_{23}^* = 7$ 而 $c_{31}^* = 9$，因此排 o_{23}，工作 2 的工序已排完，所以它的完成時間是 7。
6. 剩下工作 3，因為機器 0 的下一個可以開始時間是 7，o_{31} 的完成時間是 7+7=14。



三個工作的完成時間分別是 5、7、14，所以最後輸出答案 5+7+14=26。

輸入格式

第一行有兩個整數 N 與 M，代表 N 台機器與 M 個工作，接下來有 M 個工作的資訊，輸入的順序即是工作編號順序。每個工作資訊包含兩行，第一行是整數 P，代表到工序數量；第二行是 $2{\cdot}P$ 個整數，每兩個一組依序代表一道工序的機器編號與需求時間。機器的編號由 0 開始。參數 N、M、P 以及每個工序的需求時間都是不超過 100 的正整數。

輸出格式

輸出每個工作的完成時間的總和。

範例一：輸入

```
3 3
2
2 4 1 1
3
0 2 2 2 0 1
1
0 7
```

範例一：正確輸出

```
26
```

範例二：輸入

```
2 3
1
0 4
1
1 5
1
1 3
```

範例二：正確輸出

```
15
```

評分說明

輸入包含若干筆測試資料，每一筆測試資料的執行時間限制均為 1 秒，依正確通過測資筆數給分。其中：

第 1 子題組 20 分：只有一台機器，各工作只有一道工序。

第 2 子題組 30 分：各工作只有一道工序。

第 3 子題組 50 分：無其他限制。

題目分析：

這個題目看起來題目敘述很長，題目又涉及似乎很高深的工作排程，或許讓一些人望之卻步。其實題目敘述本身不長，是例子的說明比較長，而APCS的考試不會要求合基本數學以外的背景知識，所以這個題目完全不需要擔心懂不懂排程。這種題目是屬於模擬題，題目中會敘述一個流程，希望你模擬這個流程來計算某些數值。模擬題的重點是:「設定適當的變數記錄某些狀態，

根據流程來逐步更新這些狀態。」

在本題中，有M個工作與N台機器，每個工作有若干工序(task)必須依序進行，每個task需要某台特定機器以及需要若干時間，最重要的流程就是安排task的規則：針對每一個工作的第一個尚未排程的工序，計算出此工序的「最早完成時間」，然後挑選出最早完成時間最小的工序納入排程，如果有多個最早完成時間都是最小，則挑選其中工作編號最小之工序。對於每一個工作，它的下一個task的最早完成時間是什麼呢？應該是「這個task可以開始的時間+它所需要的時間」，那麼，它可以開始的時間又是什麼呢？依據定義，它必須在它的前一個task完成後才能開始，而且它所需要的機器必須是空下來的。所以，在模擬的過程，除了輸入的資料之外，我們必須去記錄：

(1)對每一個工作i，下一個要安排的task：cur[i]。

(2)對每一個工作i，前一個task的完成時間：pre_fin[i]。

(3)對於每一個機器j，下一個可以開始的時間：avl[j]。

子題分析：

本題有三個子題：

第1子題組20分：只有一台機器，各工作只有一道工序。

第2子題組30分：各工作只有一道工序。

第3子題組50分：無其他限制。

其差異在機器是否只有一台，每個工作是否只有一個task，這三個子題有向上包含的關係，解第二子題的解法可以解第一子題，解第三子題的解法可以解全部子題。

對於子題一，只有一台機器且每個工作只有一個task，所以相當於最短的工作先做，我們只需要每次挑尚未排入的工作中最短的就可以了，讀者或許會發現，這相當於將工作先從小到大排序後一一排入，順便一提，這其實是有名的「shortest job first」排程法。

對於子題二，每一個工作只有一個task，所以，不會有工作在某機器上執行了以後再跑到別的機器上要插隊，因此它還是「shortest job first」，只是要把每個工作丟到它要的機器後，每個機器再把它的工作從小排到大。

以下的分析將以解所有子題的完全解為主，在本題的最後再說明子題一與二的範例程式。

方法與流程：

程式的架構其實很簡單，用一個迴圈每次安排一個task直到所有task都安排完畢。迴圈中，計算每一個工作下一個task的可能最早完成時間，並且

找出在最小值。找到之後，將此task排入機器，更新機器與工作的狀態。程式架構如下：

```
輸入資料；
設定機器與工作初始狀態；
while 尚未完成所有工作 {
    計算每一個工作下一個task的最早完成時間；
    找出各個工作的最小值；
    將該工作的該task排入機器；
    更新機器與工作的狀態；
    如果該工作已經完成，記錄完成時間；
}
```

註：迴圈內每次尋找task最早完成時間的最小值的時候，我們都重新計算，事實上，利用優先佇列(priority_queue)或類似資料結構，我們可以更有效率地完成此事，學過資料結構又有興趣的讀者可自行嘗試。

程式實作：

這個問題的實作部分可以有兩個方式：使用資料結構中的佇列(queue)或者只用陣列。為了讓初學者也能明瞭，我們還是以基本指令與資料儲存方式為主，後面在補充說明使用queue的方法。誠如上述，模擬題最重要的是記錄狀態，所以變數的宣告設計是一個重點，下面是整個程式的上半部，列出了需要記錄的資料變數宣告以及輸入的部分。

```
1   #include<stdio.h>
2   #define oo 10000000
3   #define NJ 110
4
5   int main() {
6       int cur[NJ]; // current task for each job
7       int num_task[NJ]; // number of tasks for job i
8       int mac[NJ][NJ]; // machine for each task of each job
9       int task_len[NJ][NJ]; // time for each task
10      int avl[NJ]; // available time for machine i
11      int pre_fin[NJ]; // finish time for previous task for jobs
12      int i,j,num_mac,num_job;// num of machine and job
13      // read input
14      scanf("%d%d",&num_mac,&num_job);
15      for (i=0;i<num_job;i++) {
16          scanf("%d",num_task+i);
17          for (j=0;j<num_task[i];j++)
18              scanf("%d%d",&mac[i][j],&task_len[i][j]);
19      }
```

首先定義了oo無窮大的數值，這是用來找最小值使用的，只要比可能的時間來得大就夠了，第6~11行是我們需要的變數，除了輸入的資料之外，也

包含上面分析所提到需要記錄的狀態，其定義可以參考程式內的備註。第13~19行是依據題目敘述的輸入格式讀入個資料，附帶一提，使用scanf讀入陣列資料時，&num_task[i]也可以寫成num_task+i，如第16與18行。

接著開始模擬流程，在21~22行設定狀態初值：機器的可開始時間為0，工作的下一個task為0，以及前一個task完成時間也為0。從第23行開始進入排程的迴圈，earliest是用來找最小值的變數，job與machine用來存目前找到最小時是哪一個工作與使用哪一台機器。第27行的for-loop計算每一個工作目前task的最早完成時間，第28行的判斷式表示：如果工作i的目前task編號超過它所有的task數量，則該工作已經全部排完，不需要管它了。

```
20      int earliest,total=0;
21      for (i=0;i<=num_mac;i++) avl[i]=0;
22      for (i=0;i<num_job;i++) cur[i]=pre_fin[i]=0;
23      while (1) { // loop until all job finished
24          earliest=oo;
25          int mytime,job=-1,machine=-1;
26          //find the earliest finished task
27          for (i=0;i<num_job;i++) { // for job i
28              if (cur[i]>=num_task[i]) // no task left, job finished
29                  continue;
```

最後一段程式在下方。第31~33行計算該工作目前task的最早完成時間，計算方式在上面的分析與程式註解中都有，接著是比較是否為目前最小，如果是，就記錄之(第34~38行)，注意第34行的比較沒有等號，因為題目說相等時編號小的先排。在離開第39行的迴圈之後，我們先檢查是否earliest==oo，如果是，代表每一個工作都已經完成，我們就可以結束迴圈了。否則，我們要將task排入機器，要做的其實是更新狀態：第41行更新機器與工作的下一個可以開始時間，第42行更新該工作的下一個task編號。第43~44行判斷該工作是否已經全部完成，如果是，依照題意，將完成時間加總到變數中。最後，程式結束前輸出答案。

```
30  // task_finish_time=task_length+max(machine_available,previous_task_finish)
31              mytime=avl[mac[i][cur[i]]];
32              if (pre_fin[i]>mytime) mytime=pre_fin[i];
33              mytime+=task_len[i][cur[i]];
34              if (mytime<earliest) {
35                  earliest=mytime;
36                  job=i;
37                  machine=mac[i][cur[i]];
38              }
39          }
40          if (earliest==oo) break; // all job finished
41          avl[machine]=pre_fin[job]=earliest;
42          cur[job]++;
43          if (cur[job]>=num_task[job]) // job finished
44              total+=earliest;
45      }
46      printf("%d\n",total);
47      return 0;
48  }
```

程式實作(使用queue)：

接下來說明在使用queue的資料結構情形下的範例程式，由於其流程與上述程式完全相同，只有儲存資料的方式不同，以下我們只說明差異的部分。首先，queue可以說是一個一端入口，另一端出口的一維陣列，因為每一個job的task都是依序執行的，這是適合使用queue來存放task的原因。在C++中有內建的queue資料結構，所以使用C++的人適合使用。雖然自己以C的語法來寫queue也不難，但在考試的時候，由於考試時間有限，就使用前述的陣列方法就可以了。以下使用C++的語法來實作。首先，要使用queue必須先include適當的標頭檔，這裡介紹一個考試時候的偷懶方法：只要include第一行的那個標頭檔，就可以擁有全部需要的標頭檔了：你給她一行，她給你全世界。

為了讓程式更清楚，我們定義了一個名為TASK的結構(struct)，對於每一個task我們在結構中儲存該task需要的機器(mach)與需要的時間(len)。在第9行我們幫每一個工作都宣告一個內容為TASK的queue，第10~11行與前面的程式一樣是記錄機器與工作的狀態。這裡可以發現我們的變數宣告簡單很多也清爽很多，原因之一是我們不需要記錄各工作目前排到哪一個task，因為在queue中的出口端就是目前的task。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  #define oo 10000000
4  #define NJ 110
5  struct TASK {
6      int mach,len; // needing machine and time for task
7  };
8  int main() {
9      queue<TASK> task[NJ]; // task queue for each job
10     int avl[NJ]; // available time for machine i
11     int pre_fin[NJ]; // finish time for previous task for jobs
12     int i,j,num_mac,num_job;// num of machine and job
```

下面是第二段程式，是輸入與初值設定的部分，與前述程式的差異是第20行放入queue的時候要使用queue的push()函數。

```
13     // read input
14     scanf("%d%d",&num_mac,&num_job);
15     for (i=0;i<num_job;i++) {
16         int t,m,l;
17         scanf("%d",&t);
18         for (j=0;j<t;j++) {
19             scanf("%d%d",&m,&l);
20             task[i].push({m,l});
21         }
22     }
23     int earliest,total=0;
24     for (i=0;i<=num_mac;i++) avl[i]=0;
25     for (i=0;i<num_job;i++) pre_fin[i]=0;
```

最後是模擬流程的迴圈，因為當一個task被排入之後，在第44行我們會

將它從queue中移除，所以第31行判斷該工作是否已經全部完成時，可以用queue的empty()函數。第34行的t是目前要計算的task，它是以front()的方式設定，第35行計算兩個數的最大值可以直接用C++的max()。

```
26      while (1) { // loop until all job finished
27          earliest=oo;
28          int mytime,job=-1,machine=-1;
29          //find the earliest finished task
30          for (i=0;i<num_job;i++) { // for job i
31              if (task[i].empty()) // no task left, job finished
32                  continue;
33  // my_task_finish_time=task_length+max(machine_available,my_previous_finish)
34              TASK t=task[i].front();
35              mytime=t.len+max(pre_fin[i],avl[t.mach]);
36              if (mytime<earliest) {
37                  earliest=mytime;
38                  job=i;
39                  machine=t.mach;
40              }
41          }
42          if (earliest==oo) break; // all job finished
43          avl[machine]=pre_fin[job]=earliest;
44          task[job].pop();
45          if (task[job].empty()) // job finished
46              total+=earliest;
47      }
48      printf("%d\n",total);
```

使用C++或者C是因人而異，C++中有許多好用的函數，當然比較方便，但是要熟悉這些函數對初學者也是一個負擔。對於APCS的實作題來說，不會有太大的差別，因為題目都不太難，而且考試設定的是使用基本語法與基本資料儲存方式(陣列)就可以完成的題目，所以，如果你只學會C也不必太擔心。當然，如果已經學到一個程度，還是建議可以使用C++，畢竟對於較難的題目，C++方便得多。

Python的程式設計：

基本的程式架構與上面C的程式雷同，只變更成Python語法：

```
n_mac,n_job=map(int,input().split())
job=[[] for i in range(n_job)] # list of list of pair
total_task=0   # total number of tasks
for i in range(n_job):
    k=int(input())
    total_task+=k
    l=[int(x) for x in input().split()]
    for j in range(0,2*k,2):
        job[i].append((l[j],l[j+1])) # pair(machine,length)
avl=[0]*n_mac # available time for machine
pre_fin=[0]*n_job # fininish time of previous task for job
cur=[0]*n_job # current task for job
```

```
for iter in range(total_task): # each task per iteration
    earliest=100000000
    who=machine=-1  #which job and which machine
    for i in range(n_job):
        if cur[i]>=len(job[i]): continue
        # compute the earliest task-finish-time for job i
        my_machine=job[i][cur[i]][0]
        my_len=job[i][cur[i]][1]
        mytime=max(avl[my_machine],pre_fin[i])+my_len
        if mytime<earliest: # current minimum
            earliest=mytime
            who=i
            machine=my_machine
    #update status after scheduling the task
    avl[machine]=earliest
    cur[who]+=1
    pre_fin[who]=earliest
print(sum(pre_fin)) # fin contains the finish time for each job
```

陣列與輸入的部分略有不同，請注意本題的輸入格式，每一個工作的所有task所需的機器與時間是在同一行輸入，所以我們把讀入一行後，是每兩個一組放在job中，而job是一個list of list，類似二維陣列，留意Python初始二維陣列與一維陣列的方式。其他部分與C/C++的程式大同小異，這裡我們改寫了原來的while (1)迴圈，而是控制一個迴圈執行total_task次，另外，當迴圈結束後，每個工作完成時間就是它最後一個task的完成時間，所以我們直接輸出sum(pre_fin)。

子題一與二的程式實作：

最後我們補充說明子題一與二的程式實作。這兩個子題雖然是第三子題的退化(簡化)形式，它們本身是非常有趣的問題，是很好的教學範例，所以我們會舉出一些不同的解法。首先看子題一：只有一台機器且每個工作只有一個工序。依照題意，就把所有的工作從短到長依序安排即可。先提出最直接的寫法。第5~10行是宣告與輸入的部分，有些輸入資料在第一子題是沒用的，但還是要把她輸入。工作的長度放在job_len[]中。第13行我們宣告了一些變數，其中avl是機器可以開始的時間。第14行的迴圈我們每次要挑選尚未排程中的最短工作，找最小值的迴圈在第16~21行。這裡有一個要處理的問題就是要避免挑到以經挑選過的。在陣列中將它移除是不明智的做法，通常的解決方法有兩個：使用變數來標示已經挑過，或者是將挑過的改成一個不可能再被挑選的數字。我們採用後者，所以再在第24行把挑過的改成無窮大(定義為1000，因為本題最大只到100)。

```c
// only for subtask 1, one machine, one task per job
#include<stdio.h>
#define oo 10000
int main() {
    int job_len[110]; // time for each job
    int i,j,num_mac,num_job;// num of machine and job
    scanf("%d%d",&num_mac,&num_job); //num_mac==1, useless
    for (i=0;i<num_job;i++) {
        scanf("%d",&j); // num of task==1, useless
        scanf("%d%d",&j,&job_len[i]); // j is machine, useless
    }
    // start scheduling, avl=available time of machine
    int avl=0,total=0,job,mini;
    for (i=0;i<num_job;i++) { // loop until all job finished
        mini=oo; // to find shortest job
        for (j=0;j<num_job;j++) {
            if (job_len[j]<mini) {
                mini=job_len[j];
                job=j;
            }
        }
        avl+=mini; // complete time for this job
        total+=avl; // add to solution
        job_len[job]=oo; // never chosen again
    }
    printf("%d\n",total);
    return 0;
```

從教學的觀點，我們再提出一種更簡單的寫法，因為每次都挑最小，不如一次把全部排好序再一個一個排就可以了，這次我們以C++的與法而且要用到sort()，讓大家可以學習使用sort()的方式。和上面的程式唯一的差別是第15行的sort()，所以就不多作說明了。

```c
// only for subtask 1, one machine, one task per job
// using sort
#include <bits/stdc++.h>
using namespace std;
#define oo 10000
int main() {
    int job_len[110]; // time for each job
    int i,j,num_mac,num_job;// num of machine and job
    scanf("%d%d",&num_mac,&num_job); //num_mac==1, useless
    for (i=0;i<num_job;i++) {
        scanf("%d",&j); // num of task==1, useless
        scanf("%d%d",&j,&job_len[i]); // j is machine, useless
    }
    // shortest job first, sort the job_len
    sort(job_len,job_len+num_job);
    int avl=0,total=0; // complete time of previous
    for (i=0;i<num_job;i++) {
        avl+=job_len[i]; // complete time of this job
        total+=avl; // add to solution
    }
    printf("%d\n",total);
    return 0;
}
```

接著提供子題一的Python範例程式，它更為精簡，和上面C++的程式的差異只有語法，但要留意輸入的方式。

```python
n_mac,n_job=map(int,input().split())
job=[0 for i in range(n_job)] # list job length
for i in range(n_job):
    k=int(input()) # num of task, ==1 useless
    mac,len=input().split() # mac==1 is useless
    job[i]=int(len)
job.sort()
avl=0; total=0 # avl= machine available time
for len in job: # each task per iteration
    avl+=len
    total+=avl
print(total)
```

接著我們來看看子題二，有多台機器，但每個工作只有一個task。先說明不使用sort()的寫法。因為要儲存每個工作的機器與時間，在第6行我們先宣告兩個陣列，接著是讀入資料，第i個工作的機器與時間分別存在mac[i]與job_len[i]。

```
1  // only for subtask 2, several machine, one task per job
2  #include<stdio.h>
3  #define oo 1000000
4  int main() {
5      // time and machine for each job
6      int job_len[110],mac[110];
7      int i,j,num_mac,num_job;// num of machine and job
8      scanf("%d%d",&num_mac,&num_job);
9      for (i=0;i<num_job;i++) {
10         scanf("%d",&j); // num of task==1, useless
11         scanf("%d%d",&mac[i],&job_len[i]);
12     }
```

接著開始安排，誠如前面分析的，這個子題每個工作只有一道工序，不回有人插隊，所以其實可以針對每台機器各自排序，但這裡我們先依照直接的題意要求計算每一個工作的最早開始時間，因為每一台機器要記錄它的可開始時間，所以我們宣告了avl[]的陣列，此外在運用迴圈挑選最小值時，我們宣告了一個done[]的陣列以done[j]表示第j個工作已完成，就不再列入比較。在第19行比較最小值的時候檢查該工作是否已完成，而該工作的最小完成時間是它的長度加上它所要的機器的可以開始時間。其他部分跟子題一相同就不多說明。

```
13      // start scheduling, avl=available time for each machine
14      int avl[110]={0},total=0,job,mini;
15      int done[110]={0};
16      for (i=0;i<num_job;i++) { // loop until all job finished
17          mini=oo; // to find earliest finished job
18          for (j=0;j<num_job;j++) {
19              if (done[j]==0 && job_len[j]+avl[mac[j]]<mini) {
20                  mini=job_len[j]+avl[mac[j]];
21                  job=j;
22              }
23          }
24          avl[mac[job]]=mini; // complete time for this job
25          total+=mini; // add to solution
26          done[job]=1; // never chosen again
27      }
28      printf("%d\n",total);
29      return 0;
30  }
```

以下再來看使用排序的方法。基本上，我們幫每一台機器準備一個陣列儲存需要它的那些工作長度各是多少，然後對每個機器各自從小到大排程就可以了。這個有很多方式可以來實作：

1. 像上面可以那支程式先存好每個工作的時間與需求機器，然後跑一個迴圈每次處理一個機器，迴圈內掃描所有工作抓出需要它的機器放入一個陣列，然後針對此陣列排序後計算。這個方法其實浪費很多時間(每次重新找需要某台機器的工作)，但是就本題的需求是足夠的。

2. 用二維陣列來完成，每一列存一台機器的所有工作。讀入資料時直接將工作長度存入所屬陣列，再輸入與處理過程我們需要記錄每個機器的陣列長度，所以還要多一個陣列來記錄。

3. 與二維陣列類似，但使用C++的vector來存每一個機器的工作。

這份文件雖然以基本指令為主，但也希望能講解一些常用好用的資料結構，一方面讓沒學過的人有機會學，另一方面讓本來會使用這些東西的人有適合的範例程式參考。所以以下我們講解第3種使用vector的方法，其他兩種留給有興趣的人自己嘗試。

C++的vector其實可以看成可變長度的陣列，使用起來跟陣列差不多。在第7行我們宣告了一個陣列，每一個成員是一個vector<int>，這代表一個裝整數的vector。在第11行我們將工作存入其所需機器的vector中，增加元素用push_back(存入值)。

```
1  // only for subtask 2, using vector and sort
2  #include <bits/stdc++.h>
3  using namespace std;
4  int main() {
5      int i,j,mach,num_mac,num_job;// num of machine and job
6      scanf("%d%d",&num_mac,&num_job);
7      vector<int> mach_task[num_mac]; // all tasks on machine i
8      for (i=0;i<num_job;i++) {
9          scanf("%d",&j); // num of task==1, useless
10         scanf("%d%d",&mach,&j); // machine and length
11         mach_task[mach].push_back(j); // assign to mach
12     }
13     // avl=available time of machine
14     int avl,total=0;
15     for (i=0;i<num_mac;i++) {
16         // shortest job first, sort the job_len
17         sort(mach_task[i].begin(),mach_task[i].end());
18         avl=0;
19         for (int len: mach_task[i]) {
20             avl+=len; // complete time of this job
21             total+=avl; // add to solution
22         }
23     }
24     printf("%d\n",total);
25     return 0;
```

第15行的迴圈每次處理一台機器，將它分配到的工作長度排序，然後在第19行的迴圈將這些工作一個一個計算完成時間，這個迴圈的寫法是range-base for-loop)，也可以改寫成：

```
for (j=0; j<mach_task[i].size(); j++) {
    avl+=mach_task[i][j];
    total+=avl;
}
```

最後提出子題二的Python範例程式，你可以發現它的方法跟上面使用vector的非常類似，事實上Python的for-loop就是range-based for-loop。

```
n_mac,n_job=map(int,input().split())
mac_task=[[] for i in range(n_mac)] # list of list of pair
for i in range(n_job):
    k=int(input()) # num of task, ==1 useless
    mac,len=map(int,input().split()) # mac==1 is useless
    mac_task[mac].append(len)
total=0
for i in range(n_mac):
    mac_task[i].sort()
    avl=0
    for len in mac_task[i]: # each task per iteration
        avl+=len
        total+=avl
print(total) # fin contains the finish time for each job
```

第 4 題 反序數量

問題敘述

考慮一個數列 $A = (a[1], a[2], a[3], ..., a[n])$。如果 A 中兩個數字 $a[i]$和 $a[j]$滿足 $i < j$ 且 $a[i] > a[j]$，則我們說$(a[i], a[j])$是 A 中的一個反序(inversion)。定義 $W(A)$ 為數列 A 中反序的數量。例如，在數列 $A = (3, 1, 9, 8, 9, 2)$中，一共有 $(3, 1)$、$(3, 2)$、$(9, 8)$、$(9, 2)$、$(8, 2)$、$(9, 2)$一共 6 個反序，所以 $W(A) = 6$。

給定一個數列 A，計算 $W(A)$ 最簡單的方法是對所有 $1 \le i < j \le n$ 檢查數對$(a[i], a[j])$，但是在序列太長時，計算時間就會超過給定的時限。以下是運用分而治之 (divide and conquer) 的策略所設計的一個更有效率的計算方法。

1. 將 A 等分為前後兩個數列 X 與 Y，其中 X 的長度是 $n/2$。
2. 遞迴計算 $W(X)$ 和 $W(Y)$。
3. 計算 $W(A) = W(X) + W(Y) + S(X, Y)$，其中 $S(X, Y)$是由 X 中的數字與 Y 中的數字所構成的反序數量。

以 $A = (3, 1, 9, 8, 9, 2)$ 為例，$W(A)$ 計算如下。

1. 將 A 分為兩個數列 $X = (3, 1, 9)$ 與 $Y = (8, 9, 2)$。
2. 遞迴計算得到 $W(X) = 1$ 和 $W(Y) = 2$。
3. 計算 $S(X, Y) = 3$。因為有三個反序 $(3, 2)$、$(9, 8)$、$(9, 2)$是由 X 中的數字與 Y 中的數字所構成。所以得到 $W(A) = W(X) + W(Y) + S(X, Y) = 1+2+3 = 6$。

請撰寫一個程式，計算一個數列 A 的反序數量 $W(A)$。

輸入格式

測試資料有兩列，第一列為一個正整數 n，代表 A 的長度。第二列有 n 個不大於 10^6 的非負整數，代表 $a[1], a[2], a[3], ..., a[n]$，數字間以空白隔開。

輸出格式

輸出 A 的反序數量 $W(A)$。請注意 $W(A)$可能會超過一個 32-bit 整數所能表示的範圍。

範例一：輸入

```
6
3 1 9 8 9 2
```

範例一：正確輸出

```
6
```

範例二：輸入

```
5
5 5 4 3 1
```

範例二：正確輸出

```
9
```

評分說明

輸入包含若干筆測試資料，每一筆測試資料的執行時間限制均為 1 秒，依正確通過測資筆數給分。其中：

第 1 子題組 10 分：$1 \le n \le 10^3$。

第 2 子題組 30 分：$1 \le n \le 10^5$，n 為偶數，輸入數列保證 $a[1] \le a[2] \le a[3] \le ... \le a[m]$ 且 $a[m+1] \le a[m+2] \le a[m+3] \le ... \le a[n]$，其中 $m = n/2$。也就是數列前半與後半是各自排好序的。

第 3 子題組 60 分：$1 \le n \le 10^5$，無其他限制。

題目分析：

這個題目的敘述不長，看懂反序的定義也不難，但是這題目是涉及計算效率的演算法問題，所以並不簡單，是本次考試中為五級分設計的題目。對於學習過演算法的人，這題目不算難，因為事實上這個題目的程式設計與所謂的合併排序法(merge sort)幾乎是一模一樣，而後者是講解分治(divide-and-conquer)策略時最常用的經典範例之一。不過，因為題目給了提示，沒有學習過演算法的人也有機會答對的。底下的說明會涉及程式的複雜度(效率)，所以會有一點數學，我們盡量簡單的說明。

分治是一個演算法的重要策略，我們可以用本題的提示中所描述的方法來講說分治。我們要計算A數列的反序量W(A)，我們可以先將A一刀切成左右兩個等長數列X與Y(長度差不超過1)，然後 $W(A) = W(X) + W(Y) + S(X, Y)$，其中S表示$X$中的數字與$Y$中的數字所構成的反序數量。分治的好處在於等號右端的$W(X)$與 $W(Y)$可以遞迴求解，所以我們只要會設計如何有效的計算$S(X, Y)$就可以輕易地得到一個有效率的程式。打趣的說，所謂分治的精神就是：「一刀均分左右，兩邊各自遞迴，返回合併之日，答案獲得之時。」以這題來說，我們就是只要設計一個計算$S(X, Y)$的方法就行了。

子題分析：

本題有三個子題：

第1子題組10分：$1 \le n \le 10^3$。

第2子題組30分：$1 \le n \le 10^5$，數列前半與後半是各自排好序的。

第3子題組60分：$1 \le n \le 10^5$，無其他限制。

對於子題一，$n \le 10^3$，使用題目所講的兩兩比較就可以算出答案，實作時只需要一個雙迴圈加上一個if判斷就可以了，因為太簡單，以下我們就不說明了。對於子題二，數列長度到達十萬，兩兩比較需要大約50億次比較，顯然耗時過長。因為這個序列的左右兩半都是各自排好序的，因為排好序的序列是沒有反序對的，所以這個子題的意思就是要計算分治遞迴中一次合併

的$S(X, Y)$，所以這個子題也就是第三子題完全解的一個部分，因為長度大，所以還是需要設計一個有效率計算$S(X, Y)$的方法。以下的分析與實作將只針對完全解，子題二的寫法已經包含在內。

方法與流程：

重點在有效率的計算$S(X, Y)$，看似簡單也許不太簡單，如果我們將左邊的每一個跟右邊的每一個做比較，那至少要做(n/2)×(n/2)次比較，這樣的方式複雜度又會來到$O(n^2)$，沒有得到分治的好處，因此我們需要一個比$O(n^2)$更好的方法計算S(X,Y)。假設我們先將X與Y都各自從小到大排好序，可以怎麼做呢？有人會想到對排好序的數列，二分搜應該有幫助，確實沒錯，只要有一邊是排好序的，例如說右邊，我們可以將左邊的每一個在右邊進行二分搜，這樣可以在O(nlog(n))的時間完成S的計算，整個程式的時間複雜度則是$O(n\log^2(n))$，以本題要求的資料量大小十萬來說，這個方法就足夠好了。這裡說明一下，遞迴程式的複雜度計算不是一件容易的事，涉及一些離散數學的內容，我們這裡無法做完整的教學。不過我們可以簡單的說明，對於常見均分為二的分治，如果分割與合併所花的時間是O(n)，則全部的複雜度是O(nlog(n))；如果分割與合併所花的時間是O(nlog(n))，則全部的複雜度是$O(n\log^2(n))$；如果分割與合併所花的時間是$O(n^2)$或更多，則全部的複雜度與單一次分割與合併一樣。上面的說明，我們可以得到以下遞迴的分治演算法，我們習慣用左閉右開的區間。

```
//計算陣列arr在 [left,right) 區間內的反序量
sol(left, right)
    if right-left<2 return 0;
    mid=(left+right)/2;
    inv=sol(left,mid)+sol(mid,right); // recursive
    將陣列在[mid,right)區間的資料排序;
    for each i in [left,mid) do
        以二分搜找出[mid,right)中有x個小於arr[i];
        inv+=x;
    輸出inv;
```

以解此題來說，以上的說明或許足夠了，但其實我們可以更簡單的做到更好，以下說明如何以O(n)的時間完成S(X,Y)的計算，這可以讓整體時間複雜度降到O(nlog(n))，其內容也就是merge sort。上面的做法是將一半排序，另外一半對它進行二分搜，所以有(n/2)個log(n/2)，加上排序也是O(nlog(n))，所以一層的複雜度是O(nlog(n))。如果兩邊都排好序呢？在兩邊都排好序的情形下，其實是不需要二分搜的。假設我們有兩個排好序的陣

列X與Y，我們要對每一個X[i]算出Y中有幾個小於它，也就是說，對與X[i]，我們要找到最小的j，使得X[i]>=Y[j]，這個j就是我們要的數量。如果Y中有j個<X[i]，那麼<X[i+1]的數量一定>=j，因為X[i+1]>=X[i]。換句話說，X[i+1]所要找的Y的index不需要管Y[j]的左邊，只要往Y[j]的右邊找就行了。以此觀念，我們得到下列方法。

```
//計算陣列X與Y構成的反序量，X與Y皆已排序，假設長度len
    inv=0;
    j=0; // Y的index
    for (i=0; i<len(X); i++)
        while (j<len(Y) and Y[j]<X[i]) j++;
        inv+=j;
    輸出inv;
```

這個方法的時間複雜度是多少？雖然是雙層迴圈，但是你可以看到i與j都是只會往前，永不回頭，每一次比較(不管在內層或外層)，i或j的值都會增加，所以它的執行時間是與兩陣列總長度的線性關係O(n)。回到我們的整個分治程序，那排序怎麼辦？如果要排序，又必須花O(nlog(n))，那麼這裡省下來的時間還是沒用！事實上，剛才這個運用兩個指標一路往前爬的方法，就是合併兩個排好序的陣列的過程，稱為merge，我們可以一面計算一面合併，計算完成時，兩個陣列也合併成一個排好序的陣列了，請看下面的演算法。

```
//計算陣列X與Y構成的反序量並合併到T，X與Y皆已排序
    inv=0;
    j=0; // Y的index
    k=0; // T的index
    for (i=0; i<len(X); i++)
        while (j<len(Y) and Y[j]<X[i])
            T[k]=Y[j];
            j++; k++;
        inv+=j;
        T[k]=X[i];
        k++;
    將Y中剩下的複製到T中。
    輸出inv;
```

為了清楚易懂，上面的說明是以兩個陣列為例，在本題中，是一個陣列的兩個區段，其實沒有不一樣，只是索引值的處理要留意。

程式實作：

遞迴程式通常觀念需要清楚，實作相對簡單，程式碼也清楚明瞭，我們先看第一個排序後二分搜的方法。C++ STL中有內建的二分搜lower_bound()，當然也可以自己寫，以下是用C++內建二分搜的寫法，上半段是遞迴副程式。

```
1   #include <bits/stdc++.h>
2   using namespace std;
3   typedef long long LL;
4   int ar[100010];
5   // interval= [le,ri), return #inversion
6   LL sol(int le,int ri) {
7       int i,j,mid=(ri+le)/2; // middle point
8       if (le+1>=ri) return 0; // terminal case
9       // recursively solve left and right parts
10      LL w1=sol(le,mid), w2=sol(mid,ri),cross=0;
11      sort(ar+mid,ar+ri); // sort the right part
12      // for each in left, binary search to find #inversion
13      for (i=le;i<mid;i++)
14          cross+=lower_bound(ar+mid,ar+ri,ar[i])-(ar+mid);
15      return w1+w2+cross;
16  }
```

不熟悉C++的人可以把第1~2行記起來，這是一個偷懶可以include所有需要的標頭檔的方式。寫區間的問題最好都要清楚定義自己要處理的區間是否含端點才不會寫錯，大部分的習慣都是處理左閉又開的區間[le,ri)。遞迴程式通常一開始都是判斷結束條件，本題的結束條件在第8行，當區間內不超過一個元素時就不必再做，此時反序數量是0。第10行遞迴呼叫計算左右兩部分，cross是用來計算跨左右的反序數，小心要使用long long否則可能溢位。第11行我們呼叫sort()將右半部排序，如果使用C就要呼叫qsort()，C++的sort()在使用上是比較方便的，切記考試或比賽時，處理大量資料的排序必須要使用庫存函數，自己寫的排序法會跑太慢。第13~14行將每一個左邊的元素對右邊使用lower_bound()進行二分搜，此函數回傳的是第一個大於等於搜尋目標的位置(指標)，將回傳值減去搜尋範圍的起始位置就是有多少個小於它。你可以注意到不管是sort()或是lower_bound()，它們的範圍描述方式都是左閉右開區間。下面是主程式的部分，非常簡單，只有將資料讀入，然後呼叫遞迴後印出就結束了，整個程式含註解與宣告也不過23行。

```
17  int main() {
18      int i,n;
19      scanf("%d",&n);
20      for (i=0;i<n;i++) scanf("%d",ar+i);
21      printf("%lld\n",sol(0,n));
22      return 0;
23  }
```

當然，有些人可能沒有記得這麼多庫存函數，一般來說，sort/qsort是

非要會呼叫不可，二分搜如果沒用過或不記得，倒是可以自己寫，因為程式並不長，但是小心，二分搜非常容易寫錯，甚至很多有經驗的人也會寫錯，因為它的道理雖然簡單，但是在不同場合，條件會有些許不同，提醒大家兩件事：(1)記得你要搜的區間範圍含不含左右端點，(2)檢查自己的程式在剩下兩個以內的資料時是否可以正確結束並傳回正確值。以下是本題以指標方式寫的二分搜，上面那支程式的lower_bound可以換成呼叫此副程式。

```
5   // binary search [le,ri) to find the first >=x
6   int *bisearch(int *left,int *right,int x) {
7       int *m;
8       while (right-left>=2) {
9           m=left+((right-left)>>1);
10          if (*m>=x) right=m;
11          else left=m;
12      }
13      return (*left>=x) ? left: right;
14  }
```

接著要說明最佳解的實作部分，也就是不需要呼叫sort()也不用二分搜的演算法，我們以C的語法來實作，因為主程式與前述寫法相同，以下只說明遞迴副程式的部分。在第4~7行是判斷結束條件與遞迴呼叫左右兩部分，與前述方法一致。第8~9行我們宣告一個暫存空間以及三個當作index的變數，從第11~18行的迴圈，我們以左邊的ar[i]為主，每次檢查是否有右邊的元素小於ar[i]，如果有，則將其複製到temp中，注意此處右邊的index每次並沒有重頭找，而是從上次停下來的地方開始。第16行計算ar[i]與多少個右邊的元素形成反序，將其加入變數cross中，最後，在回傳之前，我們將複製到temp中的寫回到ar陣列中，注意，右邊的尾巴或許有些並未移出，所以我們只要回寫temp中的個數(k)就可以了。

```
1   #include <stdio.h>
2   // interval= [le,ri), return #inversion and sorted subarray
3   long long sol(int ar[],int le,int ri) {
4       if (le+1>=ri) return 0;
5       int mid=(ri+le)/2;
6       // recursively call to left and right parts
7       long long w1=sol(ar,le,mid), w2=sol(ar,mid,ri), cross=0;
8       int temp[ri-le+1]; // working space
9       int i, j=mid, k=0; // index of left, right and temp
```

```
10      // merge two sorted lists into temp
11      for (i=le; i<mid ; i++, k++) {
12          while (j<ri && ar[i]>ar[j]) {
13              temp[k]=ar[j];
14              j++; k++;
15          }
16          cross+=j-mid; // # (right part)<ar[i]
17          temp[k]=ar[i];
18      }
19      // copy temp[0:k-1] back to ar
20      for (i=le,j=0;j<k;i++,j++)
21          ar[i]=temp[j];
22      return w1+w2+cross;
23  }
```

Python的程式設計：

基本的程式架構與上面C的程式雷同，只變更成Python語法，以下是使用排序與二分搜的寫法，程式的詳細說明請參見前面：

```
import bisect
# recursive solve [le,ri) return sorted list
def sol(ar,le,ri):
    if ri-le<2: return 0
    mid=(le+ri)//2
    inv=sol(ar,le,mid) # left part
    inv+=sol(ar,mid,ri) # right part
    tem=ar[mid:ri]
    tem.sort() #sort the right part
    for x in ar[le:mid]: #binary search for each left
        inv+=bisect.bisect_left(tem,x)
    return inv
#main program
n=int(input())
ar=[int(x) for x in input().split()]
print(sol(ar,0,n))
```

接著是不使用排與與二分搜的最佳演算法：

```python
# recursive solve [le,ri) return sorted list
def sol(ar,le,ri):
    if ri-le<2: return 0
    mid=(le+ri)//2
    inv=sol(ar,le,mid) # left part will be sorted
    inv+=sol(ar,mid,ri) #also right part
    tem=[0 for i in range(le,ri)]
    j=mid
    k=0
    for i in range(le,mid):
        while j<ri and ar[i]>ar[j] :
            tem[k]=ar[j]
            j+=1
            k+=1
        inv+=j-mid
        tem[k]=ar[i]
        k+=1
    ar[le:le+k]=tem[0:k] # sort before return
    return inv
#main program
n=int(input())
ar=[int(x) for x in input().split()]
print(sol(ar,0,n))
```

這裡要說明，在一般的個人電腦上，Python的這兩支程式對於10萬筆資料的執行時間有可能稍稍超過一秒，前面的C/C++版本則遠低於一秒。不過以前的APCS實作題測驗，對於繳交的Python程式的執行時限都是另外放寬處理。