◯Accessories

この問題では、ある商品を買うために付属品を買わなければならないという情報が与えられる。そのため、M個の情報のうち、x_i番目の商品を買うためには、a_{x_i}円+y_i円が必要となる。ただし、１つの商品を買うために複数の付属品を買わなければならない場合もあるため、元々の商品の値段$a_1,a_2,a_3,...,a_n$を配列に格納しておき(ここではaという配列に格納したとする）、i番目の情報に対して、 a[x_i-1]+=y_i という計算を行えば良い。

　この問題で注意すべき点は、商品の値段a_iおよび付属品の値段y_iの最大値が10^9で、与えられる情報量Mの最大値が10^3というところ。全ての情報で、付属品の値段が10^9であったとすると考えられる値段の最大値は、$10^9 \times 10^3=10^{12}$となるためint型ではオーバーフローしてしまう。この対策として、long long int型を使うなどより大きい値を扱える型を用意する必要がある。

◯Conbinates

まず、全ての商品の値段が異なる場合を考える。すなわち、２つの商品を買う選び方は何通りあるかを考える。N個の商品がある時、選び方は N×(N-1)/2 通りあると考えられる。これは、1番目の商品を必ず買う時の選び方は、N-1通り、2番目の商品を必ず買う時の選び方は、さっき選んだものを除外するとN-2通り、3番目の商品を必ず買う時の選び方は、N-3通り、…となるので、i番目($1 \leq i \leq n-1$)の商品を必ず買う時の選び方は、N-i通りとなる。よって、商品の選び方は、

$$(N-1)+(N-2)+(N-3)+\cdots+1$$

となる。この計算式を見ると、差が1の等差数列であるため、等差数列の和の式から

$$(N-1+1)\times(N-1)/2=N\times(N-1)/2$$

となる。

次に、商品の値段に重複がある場合を考える。この値段の種類がk種類だったとすれば、まず考えられるべき値段の選び方は、先ほどのことから k×(k-1)/2 通りある。では、重複箇所はどうすればいいだろうか。重複したところの選び方は、重複した値段の種類がk’種類あるとすれば、 k’×1=k’ 通りあると考えられる。例えば、{100,200,100}という値段の場合、重複している値段は、100円のみである。重複した箇所の選び方は、(100,100)以外考えられないため、1通りである。{100,100,100,100,100,200}のような場合であっても、重複した箇所での選び方は、(100,100)以外考えられない。よって、同じ値段が2つ以上存在する値段の種類をカウントしておき、重複箇所を除いたものだけでの選び方を求め、それに重複していた値段の種類を加算することで、この問題を解くことができる。

◯Increase

この問題では、M円で買える最大の商品の個数を求めるので、次のような貪欲的な解法で求められる。

・値段の最小値を求め、最小値が今持っている金額よりも小さければその商品を買い、値段を更新してまた同じような操作をする。そうでなければ、その時点で購入するのをやめ、それまでに購入した商品の個数が答え。
ただし、この問題の制約を見ると、商品の個数が10^5まであるので毎回、貪欲に最小値を求めていては、当然間に合わない。そこで、最小値を高速に取り出すためのデータ構造アルゴリズムを考える。RangeMinimumQuery(RMQ)を用いるといった解法も考えられるが、これは、ある区間内の最小値を求めることができるという特徴があるので、この問題ではその特徴をあまり活用できない。よって、プライオリティーキュー
(優先度付きキュー）を用いることが適切だろう。プライオリティーキューとは、根をその木の最小値とする二分木を構成することで、最小値を高速に取り出せるデータ構造のアルゴリズムである。今回は、このアルゴリズムの

$Push(x)$:プライオリティーキューにxを挿入する。

$Pop()$:プライオリティーキューから最小値を取り出す。

の機能を用いれば、この問題を解くことができる。下に、この問題をc++で実装したコードを示す。(このコードでは、c++で使えるpriority_queueを用いている)

increase.cpp　　Thu Mar 05 11:28:48 2020　　1

```cpp
#include<iostream>
#include<vector>
#include<queue>
#define ll long long int
#define rep(n) for(int i=0;i<n;i++)

int main()
{
  //(1)
  std::priority_queue< std::pair<ll,ll> > pq;
  int n,m;
  std::cin>>n>>m;
  std::vector<ll> a(n);
  rep(n)
  {
    std::cin>>a[i];
  }
  std::vector<ll> b(n);
  rep(n)
  {
    std::cin>>b[i];
  }
  rep(n)
  {
    //(2)
    a[i]*=-1;
    b[i]*=-1;
    std::pair<ll,ll> p=std::make_pair(a[i],b[i]);
    pq.push(p);
  }
  //(3)
  ll ans=0;
  while(true)
  {
    //(4)
    std::pair<ll,ll> p;
    p=pq.top();
    if(m>p.first*-1)
    {
      pq.pop();
      m+=p.first;
      p.first+=p.second;
      pq.push(p);
      ans++;
    }
    else
    {
      break;
    }
  }
  std::cout<<ans<<std::endl;
}
```

※日本語でのコメントがうまく書き出せなかったのでここに記述
(1)PrioriryQueueの生成(キューに入れるのはpair)
(2)c++で使えるPrioriryQueueは、最大値が取り出されるため、-1をかけることで絶対値にしたときの最小値を取り出せるようにしている
(3)何個購入できるか格納する変数
(4)最小値を取り出して、まだ購入できるかどうか

◯Relatives

この問題では、同じ親戚同士である店員の関係をつなげていくことで、いくつかのグループを作り、そのグループ数とグループに属している最大人数を問われている。ただ、制約を見ると、店員の最大人数が10^5、店員同士の親戚関係が2×10^5あるため、この関係を高速に処理する必要がある。そのため、店員同士の関係を木構造のようにみなし、各グループを連結させ、同じグループに属しているかどうかを高速に処理できるデータ構造アルゴリズムとして、UnionFindTree(UFT)を用いる。今回は、UFTの

Find(x):xの根を探索する。
Union(x,y):xとyの属するグループを連結する。
Size(x):xの属するグループの大きさを取得する。

の機能を用いる。下に、C++での実装例を示す。

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#define ll long long int
#define rep(n) for(int i=0;i<n;i++)

//(1)
class UnionFindTree
{
  public:
    std::vector<ll> par;
    std::vector<ll> rank;
    std::vector<ll> size;
    //(2)
    UnionFindTree(ll n)
    {
      par.resize(n);
      rep(n)
      {
        par[i]=i;
      }
      rank.resize(n);
      rep(n)
      {
        rank[i]=0;
      }
      size.resize(n);
      rep(n)
        {
        size[i]=1;
      }
      }
    //(3)
    ll Find(ll x)
    {
      if(par[x]==x)
      {
        return x;
      }
      par[x]=Find(par[x]);
      return par[x];
    }
    //(4)
    void Union(ll x,ll y)
    {
      x=Find(x);
      y=Find(y);
      if(x==y)
      {
        return;
      }
      //(5)
      if(rank[x]<rank[y])
      {
        par[x]=y;
        size[y]+=size[x];
      }
      else
      {
        par[y]=x;
        size[x]+=size[y];
        if(rank[x]==rank[y])
        {
          rank[x]++;
        }
      }
    }
    //(6)
    bool Same(ll x,ll y)
    {
```

```
   71:       return Find(x)==Find(y);
   72:     }
   73:
   74:     //(7)
   75:     ll Size(ll x)
   76:     {
   77:       return size[Find(x)];
   78:     }
   79: };
   80:
   81: int main()
   82: {
   83:   ll n,m;
   84:   std::cin>>n>>m;
   85:   UnionFindTree *uft=new UnionFindTree(n);
   86:   rep(m)
   87:   {
   88:     ll x,y;
   89:     std::cin>>x>>y;
   90:     x--;
   91:     y--;
   92:     uft->Union(x,y);
   93:   }
   94:   //(8)
   95:   std::vector<ll> group(n);
   96:   rep(n)
   97:   {
   98:     group[i]=uft->Find(i);
   99:   }
  100:   //(9)
  101:   std::sort(group.begin(),group.end());
  102:   group.erase(std::unique(group.begin(),group.end()),group.end());
  103:   ll group_num=group.size();
  104:   //(10)
  105:   ll group_max=0;
  106:   rep(group_num)
  107:   {
  108:     ll val=uft->Size(group[i]);
  109:     if(val>group_max)
  110:     {
  111:       group_max=val;
  112:     }
  113:   }
  114:   std::cout<<group_num<<" "<<group_max<<std::endl;
  115: }
```

※日本語でのコメントがうまく書き出せなかったのでここに記述
(1)UnionFindTree(UFT)の実装をクラスを使って行う。
(2)UFTの初期化。parは、要素の根ノードを指すので、最初は、自分の要素番号にする。rankは、根の高さを表し、これを使うことで併合を簡単にさせる。最初は、０で初期化。sizeは、要素番号の属するグループの大きさ（連結数）を表し、最初は、自分しかそのグループに属さないため1で初期化。
(3)根ノードの探索を再帰的に実装。
(4)x,yの属するグループの併合（連結）を行う。
(5)根の低い方を連結させる（根の低かった方の根ノードを更新する）。
(6)今回は使わないが、x,yが同じグループに属するかどうか判定する関数。
(7)xの属するグループの連結数を求める。
(8)各要素の根ノードをvectorに格納しておく。
(9)根ノードの重複箇所を削除している。
(10)各グループにおける人数を取得して、最大値を求めている。
UnionFindTreeの参考文献
・https://atc001.contest.atcoder.jp/tasks/unionfind_a

○RangeIncrease

この問題で行う操作を少し分割してみる。

(1)l_j番目からr_j番目までの商品を購入し、その合計金額を出力する。
(2)l_j番目からr_j番目までの商品の値段をx_j円増加させる。

これらの操作をM回、順に行う。まず、区間内の総和を求めるので、これはRangeSumQuery(RSQ)を構築することが考えられる。そのため、通常は、データ構造アルゴリズムのセグメント木や平方分割を使って求めることができるが、今回は、要素の更新が範囲的に行われるため一点更新セグメント木では、１つの値を加算するためにO(log n)ほどかかるため間に合和ない。そこで、値の更新を必要な時だけ行うようにした（値の更新を遅延させる）セグメント木、遅延伝搬セグメント木(または、LazySegmentTree、以後、遅延セグ木とする）を用いる。今回の遅延セグ木の機能は、

Add(s,t,x,i,l,r):区間[s,t)の値にxを加算する。
getSum(s,t,i,l,r):区間[s,t)の総和を取得する。

を用いる。下に、c++での実装例を示す。

```cpp
#include<iostream>
#include<vector>

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

//(1)
class LazySegTree
{
  public:
    //(2)
    std::vector<ll> node;
    std::vector<ll> lazy;
    ll n_;
    LazySegTree(ll n)
    {
      n_=1;
      while(n_<n)
      {
        n_*=2;
      }
      node.resize(2*n_-1);
      rep(2*n_-1)
      {
        node[i]=0;
      }
      lazy.resize(2*n_-1);
      rep(2*n_-1)
      {
        lazy[i]=0;
      }
    }
    //(3)
    void lazy_eval(ll i,ll l,ll r)
    {
      if(lazy[i]!=0)
      {
        node[i]+=lazy[i];
        if(r-l>1)
        {
          lazy[i*2+1]+=lazy[i]/2;
          lazy[i*2+2]+=lazy[i]/2;
        }
        lazy[i]=0;
      }
    }
    //(4)
    void Add(ll s,ll t,ll x,ll i,ll l,ll r)
    {
      lazy_eval(i,l,r);
      if(r<=s || t<=l)
      {
        return;
      }
      if(s<=l && r<=t)
      {
        lazy[i]=x*(r-l);
        lazy_eval(i,l,r);
      }
      else
      {
        ll mid=(l+r)/2;
        Add(s,t,x,i*2+1,l,mid);
        Add(s,t,x,i*2+2,mid,r);
        node[i]=node[i*2+1]+node[i*2+2];
      }
    }

    //(5)
    ll getSum(ll s,ll t,ll i,ll l,ll r)
```

```
 71:     {
 72:       if(r<=s || t<=l)
 73:       {
 74:         return 0;
 75:       }
 76:       lazy_eval(i,l,r);
 77:       if(s<=l && r<=t)
 78:       {
 79:         return node[i];
 80:       }
 81:
 82:       ll mid=(l+r)/2;
 83:       ll leaf_l=getSum(s,t,i*2+1,l,mid);
 84:       ll leaf_r=getSum(s,t,i*2+2,mid,r);
 85:       return leaf_l+leaf_r;
 86:     }
 87: };
 88: int main()
 89: {
 90:   ll n,m;
 91:   std::cin>>n>>m;
 92:   LazySegTree *lst=new LazySegTree(n);
 93:   //(6)
 94:   rep(n)
 95:   {
 96:     ll a;
 97:     std::cin>>a;
 98:     lst->Add(i,i+1,a,0,0,lst->n_);
 99:   }
100:   rep(m)
101:   {
102:     ll l,r,x;
103:     std::cin>>l>>r>>x;
104:     l--;
105:     r--;
106:     //(7)
107:     std::cout<<lst->getSum(l,r+1,0,0,lst->n_)<<std::endl;
108:     //(8)
109:     lst->Add(l,r+1,x,0,0,lst->n_);
110:   }
111:   return 0;
112: }
```

※日本語でのコメントがうまく書き出せなかったのでここに記述
(1)遅延セグ木をクラスで実装。
(2)nodeでRSQを構築し、lazyに伝搬加算させる値を格納する。(共に0で初期化)
(3)伝搬させるか必要があるか判定し、伝搬させる必要があるなら、値を加算して、子に伝搬する。

(4)区間[s,t)に値を加算する。
(5)区間[s,t)の総和を取得する。
(6)元々の商品の値段をRSQに代入している。
(7)購入した合計金額の出力。
(8)購入した範囲の金額をx円加算。
遅延セグ木の参考文献
・http://tsutaj.hatenablog.com/entry/2017/03/30/224339

◯Sort

この問題で問われているのは、「バブルソートで発生する交換回数を高速に求められるか」ということ。もちろん、素直にバブルソートを組んでしまってはO(N^2)となるため間に合わない。そこで、数列の転倒数を高速に処理できるデータ構造アルゴリズムBinaryIndexedTree(BIT)を用いる。ここで転倒数とは、数列aにおいて$i<j$のとき$a_i>a_j$となる数のことを表す。つまり、転倒数がバブルソートの交換回数と一致するため、この問題の場合、１つ目の数列の転倒数から２つ目の数列の転倒数を引いたものが操作した回数となる。では、どのようにして転倒数を求めればいいだろうか。そもそも、BITとは、セグメント木などで実装するRSQをより高速に処理できるデータ構造である。そのため、区間総和を求めることが可能だ。この性質を使うことで、次のような組み合わせを数えることができる。
・数列aにおいて、$i<j$のとき$a_i<a_j$となる組み合わせ
例えば、{1,3,4,2,5}の場合を考えてみる。まず、BITの1番目に1を加算する。次に、3番目までの総和を求める。このとき、総和は1となる。次に、3番目に1を加算する。次に、4番目までの総和を求めると、2となり、4番目に1を加算する。同様に、2番目までの総和が1、5番目までの総和が4となる。最後に、それぞれの総和を全て足すと1+2+1+4=8となり、これが先ほどの組み合わせの数になる。実際に確かめてみると、
(1,3),(1,4),(1,2),(1,5),(3,4),(3,5),(4,5),(2,5)となり、確かに8通り存在する。
この組み合わせから、転倒数を求める。先ほどの組み合わせと転倒数との違いは、

$i<j$のとき、$a_i<a_j$となる組み合わせか$a_i>a_j$となる組み合わせ

ということだけである。また、この違いは互いに逆であることもわかる。よって、転倒数を求めるには、

転倒数＝全体の総和-($i<j$のとき$a_i<a_j$となる組み合わせの数)

で求めることができる。
BITの機能は、

Add(i,x):i番目にxを加算する。
Sum(i):i番目までの総和を求める。

があり、これらを使って実装することができる。c++での実装例を下に示す。

```cpp
#include<iostream>
#include<vector>

#define ll long long int
#define rep(n) for(int i=0;i<n;i++)
//(1)
class BIT
{
  public:
    //(2)
    std::vector<ll> node;
    ll n_;
    BIT(ll n)
    {
      n_=n;
      node.resize(n+1,0);
    }

    //(3)
    void add(ll i,ll x)
    {
      while(i<=n_)
      {
        node[i]+=x;
        i+=i&-i;
      }
    }

    //(4)
    ll sum(ll i)
    {
      ll s=0;
      while(i>0)
      {
        s+=node[i];
        i-=i&-i;
      }
      return s;
    }
};

int main()
{
  ll n;
  std::cin>>n;
  std::vector<ll> v(n);
  std::vector<ll> p(n);
  BIT *bit_v=new BIT(n);
  BIT *bit_p=new BIT(n);
  ll cnt_v=0;
  ll cnt_p=0;
  //(5)
  rep(n)
  {
    int v;
    std::cin>>v;
    cnt_v+=i-bit_v->sum(v);
    bit_v->add(v,1);
  }
  //(6)
  rep(n)
  {
    int p;
    std::cin>>p;
    cnt_p+=i-bit_p->sum(p);
    bit_p->add(p,1);
  }
  std::cout<<cnt_v-cnt_p<<std::endl;
  return 0;
}
```

※日本語でのコメントがうまく書き出せなかったのでここに記述

(1)BITをクラスで実装。

(2)BITでは、n個の要素数のみでRSQを構築できる。

(3)i番目にxを加算して、RSQを更新する。

(4)0番目からi番目までの総和を取得する。

(5) 1 つ目の数列において、転倒数を計算。
(6)2つ目の数列において、転倒数を計算。
BITの参考文献
・https://www.slideshare.net/hcpc_hokudai/binary-indexed-tree