Editorial

SEED (5pt)

Il problema richiede di trovare una terna di **interi non negativi distinti** tra di loro che formino una certa somma S e nel caso in cui non sia possibile stampare -1.

Proviamo prima di tutto a identificare quale insieme di numeri possiamo utilizzare, si parla ovviamente dell'insieme dei numeri naturali formato da $N = \{0, 1, 2, 3, 4, 5, \dots\}$.

Una volta identificato proviamo a trovare i casi in cui non sia possibile realizzare una terna: realizziamo la terna più piccola possibile prendendo i primi 3 interi e notiamo che la loro somma risulta $0 + 1 + 2 = 3$. È facile notare che per i numeri naturali $0, 1, 2$ non esistono terne e per questi casi la risposta risulta -1.

Ora proviamo a calcolare le terne possibili. Avendo a disposizione la terna A, B, C che genera la somma S è banale calcolare la somma per $S + 1$: basta addizionare semplicemente 1 ad uno dei termini evitando che si creino ripetizioni. Per evitare ripetizioni si può stabilire la seguente relazione tra i membri della terna $A < B < C$ e incrementare C di 1. Ora che sappiamo generare una terna per $S + 1$ sappiamo generare qualsiasi terna per $S + K$: basterà incrementare la C per K. Da cui possiamo dedurre che una generica terna per $S > 2$ è generata da $0, 1, S - 1$.

S	A	B	C
0	-1		
1	-1		
2	-1		
3	0	1	2
4	0	1	3
5	0	1	4
6	0	1	5

S	A	B	C
7	0	1	6

Questo non è l'unico modo per generare le terne possibili, qualsiasi altro metodo è corretto fin quando rispecchi le condizioni indicate nel testo. Un altro esempio per generare è il seguente: $A = floor(S/2) - 1, B = floor(S/2) + 1, C = S \mod 2$

S	A	B	C
0	-1		
1	-1		
2	-1		
3	0	2	1
4	1	3	0
5	1	3	1
6	2	4	0
7	2	4	1

Esempio di soluzione in c++

Complessità per testcase: $O(1)$

```
#include <bits/stdc++.h>

using namespace std;

void Genera(int S, int &A, int &B, int &C){
  if(S <= 2){
      A = -1;
  }else{
      A = 0;
      B = 1;
      C = S - 1;
  }
  return ;
}
```

```
int main(){
  int T;
  assert(cin >> T);
  for(int t = 1; t <= T; t++){
    int S;
    assert(cin >> S);
    int A, B, C;
    Genera(S, A, B, C);
    if(A == -1){
      cout << "Case #" << t << ": " << A << '\n';
    }else{
      cout << "Case #" << t << ": " << A << " " << B << " " << C << '\n';
    }
  }
  return 0;
}
```

ASCENSORE (13pt)

Il problema richiede di trovare il giorno in cui Giacomo percorre il numero minimo di scale tra i Q giorni presenti per poter effettuare le N consegne nei piani $V_1, V_2, V_{...}, V_N$ partendo ogni volta dal piano $P_i (1 \leq i \leq Q)$.

La chiave di risoluzione di questo problema è di risolvere il sotto-problema in cui dato un piano P_i si calcoli il numero minimo di scale per effettuare le consegne. Dunque cerchiamo di trovare una strategia per farlo. Analizziamo due casi limite:

1. Giacomo inizia ad effettuare le consegne dal piano più alto del palazzo; $P_i = 999\,999\,999$.
2. Giacomo inizia ad effettuare le consegne dal piano più basso del palazzo; $P_i = 0$.

In entrambi i casi conviene percorrere le scale sempre in un verso ed effettuare la consegna non appena si incontra un piano in cui è richiesta. È facile notare che in questi due casi Giacomo non torna mai in un piano che ha già visitato. Per cui la risposta nel primo caso è uguale alla differenza tra P_i e quello più basso in cui è richiesta una consegna (d'ora in poi indicata con V_{min}); nel secondo la differenza tra il piano più alto in cui è richiesta la consegna(d'ora in poi indicata con V_{max}) e P_i. Si intuisce che questa strategia è ottimale fin quando $0 \leq P_i \leq V_{min}$ oppure $V_{max} \leq P_i \leq 999\,999\,999$.

Bisogna ora analizzare l'ultimo caso possibile: $V_{min} < P_i < V_{max}$. Ci possiamo ricondurre alla strategia precedente se riuscissimo ad arrivare a V_{min} o a V_{max}. Si

osservi che arrivare a uno dei due piani comporti prendere due volte le stesse scale. Dunque è conveniente recarsi al piano più vicino per poter attuare la strategia precedente. Una volta afferrato ciò ci sono diversi modi per calcolare tale numero, uno tra questi è il seguente: $V_{max} - V_{min} + min(|V_{max} - P_i|, |V_{min} - P_i|)$.

Per la risoluzione del problema basta eseguire il calcolo per ogni giorno e salvarsi il **primo giorno** con il **minor numero di scale** prese.

Esempio di soluzione in c++
Complessità per testcase: $O(N + Q)$

```
#include <bits/stdc++.h>

using namespace std;

int Stabilisci(int N, int Q, vector <int> &V, vector <int> &P){
  int MAXH = *max_element(V.begin(), V.end());
  int MINH = *min_element(V.begin(), V.end());
  pair <int,int> ans(INT_MAX, INT_MAX);
  for(int i = 0; i < Q; i++){
    int piani = min(abs(MINH - P[i]), abs(MAXH - P[i])) + (MAXH - MINH);
    ans = min(ans, {piani, i});
  }
  return ans.second + 1;
}

int main(){
  int T;
  assert(cin >> T);
  for(int t = 1; t <= T; t++){
    int N, Q;
    assert(cin >> N >> Q);
    vector <int> V(N);
    for(int i = 0; i < N; i++){
      assert(cin >> V[i]);
    }
    vector <int> P(Q);
    for(int i = 0; i < Q; i++){
      assert(cin >> P[i]);
    }
    int ans = Stabilisci(N, Q, V, P);
    cout << "Case #" << t << ": " << ans << '\n';
  }
  return 0;
}
```

CARTE (17pt)

Il problema richiede di trovare il punteggio massimo che Fabrizio può ottenere giocando in modo ottimale.

Analizziamo il caso più banale in cui Fabrizio ha una carta con valore Y e Simone con valore X. Allora:

- Se $Y \leq X$ Fabrizio guadagnerà 0
- Se $Y > X$ Fabrizio guadagnerà $Y - X$

Osserviamo che Fabrizio conosce le carte di Simone e che ogni turno gioca per secondo, questo implica che Fabrizio può decidere quale delle sue carte giocare per ogni carta di Simone prima che la partita inizi. Inoltre notiamo anche come l'ordine in cui gioca le carte Simone è totalmente indifferente, perché come detto prima, Fabrizio avrà già scelto quale carta giocare a seconda della carta giocata da Simone.
Detto ciò possiamo immaginare che simone disponga le sue carte in un vettore s e Fabrizio disponga le sue in un vettore f, dove s_i rappresenta la carte che Simone giocherà al turno i, e f_i rappresenta la carta che Fabrizio ha scelto di giocare al turno i conoscendo s_i.

Bisogna dunque trovare in che modo disporre le carte nei vettori in modo che il risultato sia ottimale per Fabrizio.

Supponiamo che Fabrizio abbia 2 carte A e B con $A > B$ e Simone abbia 2 carte C e D con $C > D$ e proviamo a vedere quali delle possibili disposizioni massimizza la risposta per Fabrizio.
Consideriamo tutti i possibili casi:

- $B \geq C$: la disposizione è indifferente in quanto $(A - C) + (B - D) = (A - D) + (B - C)$
  - Es: $A = 6, B = 3, C = 2, D = 1 : (6 - 2) + (3 - 1) = (6 - 1) + (3 - 2) = 6$
- $A \leq D$: la disposizione è indifferente in quanto otteniamo sempre 0 punti.
  - Es: $A = 4, B = 2, C = 7, D = 5 : \{A, C\}, \{B, D\} = \{A, D\}\{B, C\} = 0$
- $A \geq C, B \leq D$: la disposizione più conveniente è $\{A, D\}\{B, C\}$, in quanto B otterrà sempre 0 punti e dunque conviene accoppiare A con D, considerato che $D \leq C$.

    - Es: $A = 7, B = 1, C = 5, D = 2 : \{A, C\}, \{B, D\} = 2 < \{A, D\}\{B, C\} = 5$
- $C \geq A \geq D, B \leq D$: la disposizione più conveniente è $\{A, D\}\{B, C\}$.Anche in questo caso Fabrizio otterrà 0 punti da B, gli conviene quindi accoppiare A con C per ottenere dei punti.
    - Es: $A = 4, B = 2, C = 5, D = 2 : \{A, C\}, \{B, D\} = 0 < \{A, D\}\{B, C\} = 2$
- $A \geq C, C \geq B \geq D$: la disposizione più conveniente è $\{A, D\}\{B, C\}$, che totalizza $B - C$ punti in più di $\{A, C\}, \{B, D\}$.
    - Es: $A = 9, B = 4, C = 8, D = 2 : \{A, C\}, \{B, D\} = 3 < \{A, D\}\{B, C\} = 7$

Si nota facilmente come la disposizione $\{A, D\}\{B, C\}$ sia sempre la migliore. Supponiamo dunque di avere i 2 vettori in un certo ordine che pensiamo essere ottimale, se esistono 2 indici per cui quell'accoppiamento non è rispettato allora utilizzarlo aumenterà il punteggio finale e quindi l'ordine che avevamo non era ottimale.

Di conseguenza i vettori devono essere ordinati in modo che quell'accoppiamento sia rispettato per ogni possibile coppia di indici, questo si ottiene solo ordinandone uno dei due in modo crescente e l'altro in modo decrescente.

Esempio di soluzione in c++
Complessità per testcase: $O(NlogN)$

```
#include <bits/stdc++.h>

using namespace std;

int Gioca(int N, vector <int> &S, vector <int> &F){
  sort(S.begin(), S.end());
  sort(F.rbegin(), F.rend());
  int ans = 0;
  for(int i = 0; i < N; i++){
    ans += max(0, F[i] - S[i]);
  }
  return ans;
}

int main(){
  int T;
  assert(cin >> T);
  for(int t = 1; t <= T; t++){
    int N;
    assert(cin >> N);
```

```
    vector <int> S(N), F(N);
    for(int i = 0; i < N; i++){
      assert(cin >> S[i]);
    }
    for(int i = 0; i < N; i++){
      assert(cin >> F[i]);
    }
    int ans = Gioca(N, S, F);
    cout << "Case #" << t << ": " << ans << '\n';
  }
  return 0;
}
```

Dublino(29pt)

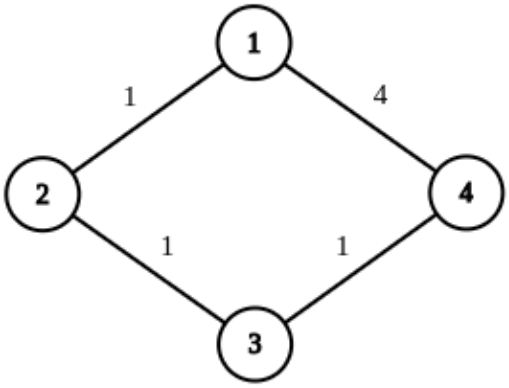
Il problema richiede di minimizzare la differenza tra i soldi spesi per raggiungere la N-esima attrazione e i soldi guadagnati vendendo i biglietti.

Proviamo a semplificare il problema considerando che non ci siano biglietti da poter vendere. Il problema richiesto si trasforma nel semplice calcolo del percorso dal costo minimo, risolvibile con dijkstra.

Aggiungiamo una piccola difficoltà, ora abbiamo a disposizione un biglietto. Con un biglietto abbiamo la possibilità di attraversare un arco a costo 0 oppure di poterlo vendere una volta arrivati all'attrazione N per minimizzare la differenza. Una prima osservazione è che non ci è dato sapere se il vendere il biglietto costituisce la scelta ottimale; questo vale anche per la scelta di azzerare il costo di un arco. Bisogna quindi valutare entrambe le possibilità e scegliere la migliore.

Valutare la differenza tra il percorso ottimale e vendere il biglietto è facile, basta sottrarre al percorso minimo il costo del biglietto. Ciò che risulta difficile è decidere quale arco dobbiamo azzerare il costo per minimizzare la differenza. Una prima strategia potrebbe risultare nell'usare il biglietto sull'arco più pesante del percorso con costo minimo; strategia non ottimale, basta considerare il seguente caso:

```
1
4 4 1
1 2 1
2 3 1
3 4 1
1 4 4
1
```



Il percorso minimo risulta uguale a 3 con arco massimo 1 totalizzando una differenza di 2. Mentre se usassimo il biglietto sul arco $1, 4$ totalizzeremmo una differenza uguale a 0. Abbiamo bisogno di una mano da una tecnica particolare che ci permette di valutare la scelta migliore, la tecnica di cui sto parlando è la programmazione dinamica. Dobbiamo in qualche modo poter identificare un sotto-problema in modo tale da non doverlo ogni volta ricalcolare. Ogni stato è identificabile dalla coppia $nodo$ e $BigliettoUsato$. In questo modo pian piano che ci espandiamo con dijkstra abbiamo due possibilità: attraversare l'arco non usando il biglietto oppure attraversarlo azzerandone il costo e usando il biglietto. Nel primo caso modifichiamo l'adiacenza con lo stesso stato di $BigliettoUsato$; nel secondo lo stato in cui non si può utilizzare il biglietto.

Ora la tabella delle distanze invece di avere una sola dimensione ne avrà due, una corrispondente al nodo e l'altra allo stato del biglietto. I vari risultati quindi saranno immagazzinati in corrispondenza dell'ultimo nodo.

Ora possiamo estendere il problema per K biglietti. Invece di tenere lo stato per un biglietto, lo modifichiamo per rappresentare lo stato di tutti i K biglietti. Ciò che importa non è sapere se abbiamo utilizzato un determinato biglietto ma quanti ne abbiamo utilizzati. Permettendo cosi di farci costruire una tabella NK dove salviamo il costo minimo per arrivare alla i-esima attrazione avendo utilizzato j biglietti ($0 \leq j \leq K$).

Computata tale tabella ora non dobbiamo far altro che salvare la differenza minima ipotizzando ogni volta di aver usato dei biglietti. Se ipotizziamo di aver usato X biglietti, dobbiamo far rifermento alla tabella che ci da il costo minimo per aver usato X biglietti e sottrarre tale valore alla somma dei $K - X$ biglietti più costosi. Tra tutti i possibili utilizzi basta salvarsi la differenza minima.

Esempio di soluzione in c++

Complessità per testcase: $O(MK + NKlog(NK))$

```
#include <bits/stdc++.h>

using namespace std;

int Visita(int N, int M, int K, vector <int> &A, vector <int> &B, vector <i
    vector < vector <pair <int, int> > > adj(N + 1);
    for(int i = 0; i < M; i++){
      int a = A[i], b = B[i], w = X[i];
      adj[a].push_back({b, w});
      adj[b].push_back({a, w});
    }
    vector <vector <int> > dista(N + 1, vector <int> (K + 1, INT_MAX));
    vector <vector <bool> > visti(N + 1, vector <bool> (K + 1, false));
    priority_queue <pair <int, pair <int, int> > > q;
    dista[1][0] = 0;
    q.push({0,{1, 0}});
    while(!q.empty()){
      int node = q.top().second.first;
      int k = q.top().second.second;
      q.pop();
      if(visti[node][k])continue;
      visti[node][k] = true;
      for(pair <int,int> i : adj[node]){
        int b = i.first;
        int w = dista[node][k] + i.second;
        // non uso il biglietto
        if(w < dista[b][k]){
          dista[b][k] = w;
          q.push({-w, {b, k}});
```

```
        }
        // uso il biglietto
        w -= i.second;
        if(k + 1 <= K && w < dista[b][k + 1]){
          dista[b][k + 1] = w;
          q.push({-w, {b, k + 1}});
        }
      }
    }
    sort(biglietti.rbegin(), biglietti.rend());
    int ans = dista[N][K];
    int biglietti_venduti = 0;
    for(int i = 0; i < K; i++){
      biglietti_venduti += biglietti[i];
      ans = min(ans, dista[N][K - (i + 1)] - biglietti_venduti);
    }
    return ans;
}

int main(){
  int T;
  assert(cin >> T);
  for(int t = 1; t <= T; t++){
    int N, M, K;
    assert(cin >> N >> M >> K);
    vector <int> A(M), B(M), X(M);
    for(int i = 0; i < M; i++){
      assert(cin >> A[i] >> B[i] >> X[i]);\
    }
    vector <int> biglietti(K);
    for(int i = 0; i < K; i++){
      assert(cin >> biglietti[i]);
    }
    int ans = Visita(N, M, K, A, B, X, biglietti);
    cout << "Case #" << t << ": " << ans << '\n';
  }
  return 0;
}
```

ALBERI2(pt31)

Il problema richiede le mosse da effettuare per poter abbattere tutti gli alberi nel minor tempo possibile. Ricordiamo le nostre opzioni:

- *A*: Abbatti, abbatte l'albero con la velocità corrente dell'ascia.
- *E*: Esplodi, abbatte almeno 3 alberi con tempo Y.
- I: Migliora, raddoppia la velocità dell'accetta e abbatte l'albero per un costo di K euro.

Già dalla premessa di calcolare il tempo minimo e la possibilità di effettuare diverse mosse ci dovrebbe indurre a risolvere il problema utilizzando la tecnica della **programmazione dinamica**.

Un primo approccio di memorizzare lo stato di un sotto-problema è il seguente: $dp[albero][velocita][soldi]$. Per transitare da uno stato all'altro basta simulare di effettuare tutte e 3 le mosse e salvarsi la migliore. Ovviamente la mossa *I* solo quando abbiamo abbastanza denaro. Per quanto questo approccio risulti corretto non è ottimale visto che la velocità non ha limiti e la somma di denaro che si può ottenere può arrivare fino a $100\,000 \times N$.

Un secondo approccio di memorizzare lo stato di un sotto-problema è il seguente: $dp[albero][velocità]$. L'osservazione fondamentale è che la somma di denaro guadagnata fino al i-esimo albero è uguale a $\sum_{j=1}^{i-1} H_i$ meno il numero di raddoppi che si sono effettuati per avere la velocità corrente. Per calcolare il numero di raddoppi basta raddoppiare la velocità iniziale fin quando non si raggiunge la velocità corrente. Per ridurre la complessità data dal calcolo del quantitativo di denaro si possono utilizzare le somme prefisse sulle altezze degli alberi. Questo approccio risulta corretto ma non ottimale in quanto la velocità non ha limiti.

Nei due precedenti approcci la velocità è sempre stato uno ostacolo alla memorizzazione del nostro stato ed è giunto il momento di limitarla. Una prima limitazione è che una volta raggiunta la velocità dell'albero più alto non ha più senso raddoppiare poiché non comporterebbe a nessun vantaggio. Sapendo che non ha più senso raddoppiare se si è raggiunti un certo limite di velocità, quante volte dobbiamo raddoppiare per raggiungerlo? Consideriamo il caso peggiore in cui la velocità inizi da 1: $1, 2, 4, 8, 16, \ldots, 65\,536, 131\,072$. Abbiamo impiegato un numero di volte pari a $log_2(H_i)$ volte per raggiungerlo. Siccome il numero di volte è molto ridotto rispetto al valore che la velocità assume possiamo memorizzarlo tranquillamente. Ora lo stato lo si può rappresentare con $dp[albero][numero\_raddoppi]$ e la velocità corrente si otterrà con $V \times 2^{numero\_radoppi}$. Raggiungendo cosi la complessità ottimale.

Ora non ci resta che ricavare le mosse da effettuare per restituirle. Se si è strutturata la dp in maniera ricorsiva basterà prima di restituire il valore ottimo per quel sotto-problema tenersi un'ulteriore tabella dove indicheremo la mossa da effettuare. Non ci resta che navigarla modificando lo stato a seconda della mossa effettuata!

Notare che questo problema offre punteggi parziali per **soluzioni greedy**, in una gara anche un singolo punto in più fa la differenza. Quindi si potevano provare varie soluzioni sperando di ottenere un buono score tipo:

- Abbattere sempre l'albero.
- Prova a usare sempre esplosivi.
- Provare a incrementare sempre.
- Provare a combinare i 3 approcci sopra elencati.

Si può ottenere circa **12pt** con l'ultima strategia.

Esempio di soluzione in c++
Complessità per testcase: $O(Nlog(H_i))$

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 1e4 + 2;
const int MAXV = 32;

int N, Y, K, *H;
int memo[MAXN][MAXV];
int mosse[MAXN][MAXV];
int vel[MAXV];
int guadagno[MAXN];
int MAXJ;

int tempo(int H, int vel){
  return (H + vel -1) / vel;
}

int dp(int i, int j){
  if(i >= N)return 0;
  int &ret = memo[i][j];
  if(ret != INT_MAX)return ret;
  pair <int, char> ans(INT_MAX, '#');
  // abbatto
  int A = dp(i + 1, j) + tempo(H[i], vel[j]);
  // esplodo
  int E = dp(i + 3, j) + Y;
  // incremento se posso
  int I = INT_MAX;
  if(j + 1 < MAXJ && guadagno[i] - ((j + 1) * K) >= 0){
    I = dp(i + 1, j + 1) + tempo(H[i], vel[j + 1]);
```

```
    }
    ans = min(ans, {A, 'A'});
    ans = min(ans, {E, 'E'});
    ans = min(ans, {I, 'I'});
    mosse[i][j] = ans.second;
    return ret = ans.first;
  }
  string Abbatti(int N, int V, int Y, int K, vector <int> &H){
      // Resetto le variaibli globali
      ::N = N;
      ::H = &H[0];
      ::Y = Y;
      ::K = K;
      fill(&memo[0][0], &memo[MAXN - 1][MAXV - 1], INT_MAX);
      fill(&mosse[0][0], &mosse[MAXN - 1][MAXV - 1], '#');
      // Calcolo della sommma dei preffisi del guadagno
      for(int i = 0; i < N; i++){
        guadagno[i + 1] = guadagno[i] + H[i];
      }
      // Calcolo delle velocità
      vel[0] = V;
      int MAXH = *max_element(H.begin(), H.end());
      MAXJ = 1;
      while(vel[MAXJ - 1] <= MAXH){
        vel[MAXJ] = vel[MAXJ - 1] * 2;
        MAXJ = MAXJ + 1;
      }
      // lancio la dp
      int ans = dp(0, 0);
      string ret;
      int i = 0;
      int j = 0;
      while(i < N){
        switch(mosse[i][j]){
          case 'A': i++; ret.push_back('A'); break;
          case 'E': i += 3; ret.push_back('E'); break;
          case 'I': i++; j++; ret.push_back('I'); break;
        }
      }
      return ret;
  }

  int main(){
    int T;
    assert(cin >> T);
    for(int t = 1; t <= T; t++){
      int N, V, Y, K;
```

```
    assert(cin >> N >> V >> Y >> K);
    vector <int> H(N);
    for(int i = 0; i < N; i++){
      assert(cin >> H[i]);
    }
    string ans = Abbatti(N, V, Y, K, H);
    cout << "Case #" << t << ": " << ans << '\n';
  }
  return 0;
}
```