A - Kuber

Bara gör exakt det som står, summera x^3 från x=1 till x=n.

Enradslösning i python:

```
print(sum([x**3 for x in range(1,int(input())+1)]))
```

B - Studschiffret

Först simulerar vi studsandet och placerar ut talen 0 till `str.size()-1` i rutnätet. Därefter läser vi av rutnätet rad för rad och lägger talen i en lista `encNum`. Denna lista beskriver nu för varje bokstav i den krypterade strängen vilket index den bokstaven har i den ursprungliga strängen, så nu går vi helt enkelt igenom alla bokstäver i den kryptarade strängen och placerar varje bokstav på rätt plats i den ursprungliga strängen.

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

using namespace std;

int main(){
   cin.sync_with_stdio(false);
   int n,m;
   string str;
   cin>>n>>m>>str;

   vector<vi> g(n,vi(m,-1));

   int r = 0;
   int c = 0;
   int dr = 1;
   int dc = 1;
   int ind = 0;
   //Simulera:
   while(ind<str.size()){
      if(g[r][c]==-1){
         g[r][c] = ind;
         ind++;
      }

      r+=dr;
      c+=dc;

      if(r<0){ r = 1; dr = 1; }
      if(c<0){ c = 1; dc = 1; }
```

```
        if(r>=n){ r = n-2; dr = -1; }
        if(c>=m){ c = m-2; dc = -1; }
    }

    //Läs av rad för rad:
    vi encNum;
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++)
            if(g[i][j]!=-1)
                encNum.push_back(g[i][j]);

    //Rekonstruera den ursprungliga strängen:
    string dec(str.size(),'.');
    for(int i = 0;i<str.size();i++)
        dec[encNum[i]] = str[i];

    cout<<dec<<endl;
}
```

C - Renovering

Vi vill para ihop maximalt antal spikar som vi redan har med spikar som vi behöver, på ett sådant sätt så att i varje par är spiken vi behöver kortare eller lika lång som den vi redan har. Säg att den längsta spiken som vi redan har har längd L. Den vill vi para ihop med den längsta spiken som vi behöver, som uppfyller att dess längd $\leq L$, eftersom ju längre spik desto större krav ställer spiken på vilka den kan paras ihop med. När vi parat ihop den längsta spiken vi redan har med den längsta vi behöver som den kan paras ihop med så tar vi bort dessa två och upprepar. När den längsta spiken vi har inte längre kan paras ihop med någon, eller när alla våra spikar är ihopparade, är vi färdiga.

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

using namespace std;

int main(){
    cin.sync_with_stdio(false);
    int n,m;
    cin>>n>>m;
    vector<int> x(n);
    vector<int> y(m);
    for(int i = 0; i<n; i++) cin>>x[i];
    for(int i = 0; i<m; i++) cin>>y[i];

    //Sortera så att de längsta spikarna hamnar sist
```

```
    sort(all(x));
    sort(all(y));

    vector<int> toBuy;

    while(x.size()){
        if(y.size()&&x[x.size()-1]<=y[y.size()-1]){ //
            Ifall den längsta spiken vi behöver är
            kortare än den längsta vi har
             //Då tar vi bort båda
            x.pop_back();
            y.pop_back();
        } else {
            //Annars kan den längsta vi behöver omöjligt
               paras ihop med någon, så vi säger att vi m
               åste köpa den och tar sedan bort den.
            toBuy.push_back(x[x.size()-1]);
            x.pop_back();
        }
    }

    sort(all(toBuy));

    cout<<toBuy.size()<<endl;
    for(int i = 0; i<toBuy.size();i++){
        cout<<toBuy[i]<<"␣";
    }
    cout<<endl;
}
```

D - Multationer

Här hinner vi testa alla möjliga multationssekvenser och kan skriva ut den kortaste. Först genererar vi rekursivt alla möjliga multationer och lägger i en lista `subs`. Därefter testar vi med funktionen `rec` rekursivt alla möjliga sekvenser av multationer, och sparar den bästa i `bestSeq`.

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

using namespace std;

vector<pair<char,string> > subs;

int n = 3; //Alfabetets längd
int m = 3; //Maximalt antal byten
```

```
int k = 3; //Maximal längd på sträng man byter ut mot

//Generera rekursivt alla strängar vi kan byta ut mot
void genSubs(string replaceWith){
   if(replaceWith.size()!=0)
      for(char x = 'A'; x<'A'+n;x++) subs.emplace_back
         (x,replaceWith);
   if(replaceWith.size()<k)
      for(char x = 'A'; x<'A'+n;x++) genSubs(
         replaceWith+char(y));
}

//Utför det faktiska bytet
string replace(string str, char from, string to){
   string out = "";
   rep(i,0,str.size()){
      if(str[i]==from) out += to;
      else out += str[i];
   }
   return out;
}


string str1; //Strängen vi börjar med
string str2; //Strängen vi ska till
vector<int> bestSeq; // Kortaste giltiga sekvensen vi
   hittat hittils

//str är hur strängen ser ut just nu, history är vilka
    multationer som hittils utförts
void rec(string str, vector<int> history){
   if(history.size()>=bestSeq.size()) return;
   if(str == str2){
      bestSeq = history;
      return;
   }
   if(history.size()<m){
      for(int i = 0; i<subs.size();i++){ //Testa alla
         möjliga multationer
         vector<int> newSeq = history;
         newSeq.push_back(i);
         // Utför multationen med hjälp av funktionen
         // "replace", och rekursera sedan
         rec(replace(str,subs[i].first,subs[i].second)
            ,newSeq);
      }
```

```
    }
}

int main(){
    cin.sync_with_stdio(false);
    cin>>str1>>str2;

    genSubs("");

    bestSeq = {0,0,0,0,0,0,0,0};

    rec(str1,vector<int>(0));

    for(int i =0; i<bestSeq.size();i++){
        cout<<subs[bestSeq[i]].first<<"␣"<<subs[bestSeq[
            i]].second<<endl;
    }
}
```

E - Robottävlingen

Att hitta maximala antalet är enkelt, det är bara att för varje ruta i rutnätet ta minimumvärdet av kolumnvärdet och radvärdet.

Att hitta minimala antalet är svårare. Vi börjar med att betrakta hur många femmor vi måste sätta ut, om vi vill ha så få som möjligt. Vi måste såklart minst lägga en femma i varje kolumn som har värdet 5 (hur skulle kolumnen fått värdet 5 om den inte innehåller en femma?), och vi måste på samma sätt även lägga en femma i varje rad. Om x_5 är antalet rader med värdet 5 och y_5 antalet kolumner med värdet 5 behöver vi därför minst $max(x_5, y_5)$ stycken femmor. Så länge det finns både någon 5-kolumn utan femma i sig och en 5-rad som inte innehåller en femma så kan vi placera en femma i den rutan som kolumnen och raden delar. Ifall vi bara har en 5-rad så kan vi placera ut en femma i den raden i någon kolumn som redan har en femma i sig, och på samma sätt kan vi göra om vi bara har en 5-kolumn. Alltså går $max(x_5, y_5)$ antal femmor alltid att uppnå.

Samma argument kan vi använda för 4,3 och 2 får att bevisa att minsta antal fyror, treor och tvåor som går att uppnå är $max(x_4, y_4)$, $max(x_3, y_3)$ och $max(x_2, y_2)$ respektive. Skillnaden är att man måste notera att när man till exempel har en 4-rad som man måste placera en fyra i, men inga 4-kolumner, så kan vi placera fyran i någon av 5-kolumnerna istället. Det kommer alltid antingen finnas en 4-kolumn eller 5-kolumn att placera i, eftersom det måste enligt raden vi jobbar med finnas en fyra någonstans, så någon kolumns maxvärde är minst 4.

Alla rutor som vi inte redan placerat något i placerar vi ettor i.

Notera att i lösningen nedan placerar vi aldrig faktiskt ut något, utan vi bara räknar direkt ut vad summan blir om vi hade placerat ut det.

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

using namespace std;

int main(){
   cin.sync_with_stdio(false); cin.tie(0);

   int n;
   cin>>n;
   vector<int> r(n);
   vector<int> c(n);
   for(int i = 0;i<n;i++) cin>>r[i];
   for(int i = 0;i<n;i++) cin>>c[i];

   int mx = 0;
   for(int i = 0;i<n;i++)
      for(int j = 0;j<n;j++)
         mx += min(r[i],c[j]);

   map<int,int> x;
   map<int,int> y;

   for(int i = 0;i<n;i++){
      x[r[i]]++;
      y[c[i]]++;
   }

   int mn = 0;
   int numPlaced = 0;

   for(int i = 1;i<6;i++){
      //Placera ut max(y[i],x[i]) stycken av sifran "i
         "
      mn += max(y[i],x[i])*i;
      numPlaced += max(y[i],x[i]);
   }

   mn += n*n - numPlaced; //Lägg till alla ettor


   cout<<mn<<"␣"<<mx<<endl;
```

```
}
```

F - Avslutningscermonin

Vi definerar en funktion $f(last, index)$ som löser uppgiften från $index$ och framåt. f svarar alltså på frågan:

Vilket är det största antalet intillsittande par vi kan uppnå bland platserna $index-1, index, index+1, ..., str.size()-1$ (platserna är 0-indexerade), givet att personen på position index-1 är från sällskap $last$?

Det fiffiga med att definera en sådan funktion är att vi kan räkna ut värdet på den rekursivt genom att testa alla olika möjligheter för hur personen på position index kan byta sin plats, räkna hur många nya intillsittande par det gav upphov till, och sedan rekursera.

Det finns 4 olika fall vi måste ta hänsyn till: ($a[i]$ beskriver vilket sällskap som urspungligen satt på position i)

Fall 1: Personen på plats index sitter kvar.
Vi räknar antal par bland $index-1$ och $index$ och rekurserar från $f(a[index], index+1)$

Fall 2: Personen på plats index byter ett steg till höger
Vi utför bytet, räknar antal par bland $index-1, index, index+1$ och rekurserar från $f(a[index], index+2)$

Fall 3: Personen på plats index byter två steg till höger (ifall man får byta två steg), och personen på $index+1$ sitter kvar
Vi utför bytet, räknar antal par bland $index-1, index, index+1, index+2$ och rekurserar från $f(a[index], index+3)$

Fall 4: Personen på plats index byter två steg till höger, och personen på index+1 byter också plats två steg till höger.
Vi utför de två bytena, räknar antal par bland $index-1, index, index+1, index+2, index+3$ och rekurserar från $f(a[index+1], index+4)$

Värdet av f beror endast av parametrarna last och index. Därför kan vi "memoize:a", det vill säga vi kommer ihåg svaret för ett givet par av parametrar, så behöver vi inte räkna ut det igen. Det gör att rekursionsträdet, som annars skulle växt exponetiellt, begränsas till att antalet noder blir maximalt antalet sätt att välja parametrarna.

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

using namespace std;
```

```
int n,m,k;
vi a;
vector<vi> mem;

//Räknar antal intillsittande par i vektorn v
int numPairs(vi v){
    int ans = 0;
    for(int i = 0; i< v.size()-1;i++)
        ans += (v[i]==v[i+1]);
    return ans;
}

int f(int index, int last){
    if(index==n) return 0;
    //Ifall värdet inte är -1 har vi redan räknat ut
       det
    if(mem[index][last]!=-1)
        return mem[index][last];

    int best = f(index+1, a[index]) + numPairs(vi({
       last, a[index]})); //Fall 1

    if(index<n-1)
        best = max(best, f(index+2, a[index]) +
           numPairs(vi({last, a[index+1], a[index]})))
           ;//Fall 2

    if(k==2){
        if(index<n-2)
            best = max(best, f(index+3, a[index]) +
               numPairs(vi({last, a[index+2], a[index
               +1], a[index]})));//Fall 3

        if(index<n-3)
            best = max(best, f(index+4, a[index+1]) +
               numPairs(vi({last, a[index+2], a[index
               +3], a[index], a[index+1]})));//Fall 4
    }


    return mem[index][last] = best;
}

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

```
    string str; cin>>str;
    n = str.size();
    a.resize(n);
    rep(i,0,n) {
        a[i] = str[i]-'A';
    }
    cin>>k;

    mem.resize(n,vi(5,-1)); //Sätt -1 som defaultvärde

    cout<<f(0,4)<<endl;
}
```