Садржај

Глава 1

Јун 2020.

Задатак: Светиљка

У једној улици налази се n кућа. Потребно је поставити једну светиљку тако да што више кућа буде обасјано. Јачина светиљке је d, што значи да ће бити обасјане само оне куће које су на удаљености мањој или једнакој d од светиљке (и лево и десно од ње). Улица је паралелна x-оси и свака кућа је одређена својом x координатом. Написати програм који реализује алгоритам за одређивање највећег броја кућа које се могу обасјати постављањем једне светиљке на улицу.

Опис улаза

Са стандардног улаза се учитавају број кућа n ($1 \leq n \leq 10^6$) и n целих бројева из интервала $[-10^9, 10^9]$ који представљају x координате кућа. Затим се учитава позитиван цео број d ($1 \leq d \leq 5 \cdot 10^8$) који представља јачину светиљке.

Опис излаза

На стандардни излаз исписати један цео број који представља максималан број кућа које могу бити обасјане једном светиљком.

Пример

Улаз	*Излаз*
6 29 -11 15 13 -68 -4 13	4

Објашњење

Постављањем светиљке у тачку са координатом 2 обасјавају се куће на координатама -11, 15, 13 и -4.

Решење

Груба сила

За сваку кућу x_i можемо израчунати број кућа које би биле осветљене када би она била та која је последња осветљена од свих кућа. То можемо израчунати тако што пребројимо све куће које су лево од ње, на растојању мањем од $2d$ у односу на њу (светиљка која је постављена у тачку $x_i - d$ осветљава интервал $[x_i - 2d, x_i]$). Пребројавање тих кућа можемо извршити грубом силом.

Алгоритам анализира n кућа и за сваку кућу x_i од њих поново анализира свих n кућа проверавајући да ли су лево од куће x_i и да ли су на растојању мањем од $2d$ у односу на њу (што се врши у константној сложености), па му је укупна сложеност $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;

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

  int d;
  cin >> d;

  int maks = 0;
  for (int i = 0; i < n; i++) {
    int broj = 0;
    for (int j = 0; j < n; j++)
      if (x[j] <= x[i] && x[i] - x[j] <= 2*d)
        broj++;
    maks = max(maks, broj);
  }

  cout << maks << endl;
  return 0;
}
```

Два показивача

Пошто светиљка осветљава d јединица на лево и d јединица на десно, све куће које су у интервалу ширине $2d$ могу бити осветљене истом светиљком. За сваку кућу x_i у сортираном низу кућа проверавамо да ли би могла да буде најдешња кућа која је осветљена светиљком (та светиљка може бити постављена у тачку $x_i - d$ и она осветљава интервал $[x_i - 2d, x_i]$). Под претпоставком да јесте, број осветљених кућа одређујемо тако што одредимо све куће које су лево од ње на растојању највише $2d$ од ње. Довољно је да знамо прву такву кућу x_j у сортираном низу кућа. Тада светиљка постављена у тачку $x_i - d$ осветљава $i - j + 1$ кућа (кућу x_i, x_j и све куће између њих).

Када са једне куће x_i пређемо на следећу $x_{i'} = x_{i+1}$, да бисмо одредили њој одговарајућу кућу $x_{j'}$, не морамо анализирати све куће из почетка, већ можемо искористити то што знамо да је x_j прва кућа која је на растојању мањем од или једнаком $2d$ у односу на кућу x_i. Ако она на растојању мањем од или једнаком $2d$ и у односу на светиљку x_{i+1}, тада смо сигурни да је то прва таква кућа (јер је, ако постоји, кућа x_{j-1} је на растојању већем од $2d$ у односу на кућу x_i, а растојање до куће x_{i+1} је још веће). Ако није, тада настављамо да анализирамо куће x_{j+1}, x_{j+2} итд. све док не дођемо до прве куће која је на растојању мањем или једнаком од $2d$ у односу на x_{i+1}.

Алгоритам, дакле, можемо имплементирати техником два показивача. Левим показивачем i обилазимо једну по једну кућу слева надесно, претпостављајући да је x_i последња осветљена кућа здесна. Десни показивач указује на прву кућу x_j слева која је на растојању мањем или једнаком од $2d$ у односу на x_i. Иницијално можемо поставити i и j на нулу и у петљи од 0 до n анализирати све куће x_i. Када год се i повећа, повећавамо и j све док је x_j на превеликом растојању у односу на x_i. Када одредимо прву кућу x_j која је на растојању мањем или једнаком од $2d$ у односу на x_i, тада израчуавамо $i - j + 1$ и ажурирамо максималан број кућа, ако је тај број већи од дотадашњег максимума.

Прикажимо рад алгоритма на једном примеру. Нека су положаји кућа након сортирања $-68, -11, -4, 13, 15, 29$ и нека је домет светиљке $d = 13$.

- Ако је последња осветљена кућа $x_i = x_0 = -68$, тада је она једина осветљена тј. $x_j = x_0 = -68$, па је $i - j + 1 = 1$.
- Ако је последња осветљена кућа $x_i = x_1 = -11$, тада је она једина осветљена тј. $x_j = x_1 = -11$ (j се увећава на 1, јер је растојање између кућа -68 и -11 веће од $2d = 26$), па је $i - j + 1 = 1$.
- Ако је последња осветљена кућа $x_i = x_2 = -4$, тада је прва кућа која је осветљена $x_j = x_1 = -11$ (j се не увећава, јер је растојање између кућа -11 и -4 мање од $2d = 26$), па је $i - j + 1 = 2$.

- Ако је последња осветљена кућа $x_i = x_3 = 13$, тада је прва кућа која је осветљена $x_j = x_1 = -11$ (j се не увећава, јер је растојање између кућа -11 и 13 мање од $2d = 26$), па је $i - j + 1 = 3$.
- Ако је последња осветљена кућа $x_i = x_4 = 15$, тада је прва кућа која је осветљена $x_j = x_1 = -11$ (j се не увећава, јер је растојање између кућа -11 и 15 једнако $2d = 26$), па је $i - j + 1 = 4$.
- Ако је последња осветљена кућа $x_i = x_5 = 29$, тада је прва кућа која је осветљена $x_j = x_3 = 13$ (j се прво увећава на 2, јер је растојање између кућа -11 и 29 веће од $2d = 26$, па се опет увећава на 3, јер је растојање између кућа -4 и 29 веће од $2d = 26$, након чега остаје 3, јер је растојање између кућа 13 и 29 мање од $2d = 26$), па је $i - j + 1 = 3$.

Максимална вредност $i - j + 1$ једнака је 4..

Иницијално сортирање кућа захтева време $O(n \log n)$. Након тога обилазак вршимо техником два показивача који се крећу у истом смеру и могу начинити највише $2n$ корака, при чему се у сваком кораку врши константан број операција, па је сложеност $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;

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

  int d;
  cin >> d;

  sort(x.begin(), x.end());

  int maks = 0, j = 0;
  for (int i = 0; i < n; i++) {
    while(x[j] < x[i] - 2*d)
      j++;
    maks = max(maks, i - j + 1);
  }

  cout << maks << endl;
  return 0;
}
```

Задатак: Зли учитељ

Зли учитељ Нави одлучио је да се освети својим ученицима за ометање часа. Следећи домаћи задатак који им буде задао радиће се у паровима. Учитељ за сваког ученика зна колико сати му је потребно да уради свој део задатка, а време потребно пару ученика да уради задатак је збир времена та два ученика. Учитељ жели да одреди колико највише сати може да зада за израду домаћег задатка (зато што жели да прикрије своје зле намере и да делује фер) тако да бар један пар ученика не може да стигне да уради задатак, без обзира на то како се ученици поделе у парове.

Опис улаза

Са стандардног улаза се учитава паран број ученика n ($1 \leq n \leq 10^6$). Затим се учитава n природних бројева који представљају бројеве дана потребних за израду задатка сваког ученика.

Опис излаза

На стандардни излаз исписати један природан број који представља број дана који учитељ треба да постави као рок за израду задатка.

Пример 1

Улаз	*Излаз*
6 5 7 2 6 4 6	10

Пример 2

Улаз	*Излаз*
6 2 4 1 16 7 11	16

Решење

Потребно је одредити упаривање ученика такво да је највеће време израде домаћег задатка најмање могуће. Другим речима, у низу бројева, потребно је пронаћи оно упаривање које даје најмањи могући (у односу на сва упаривања) максимални збир пара елемената (у односу на све парове). Учитељ може ученицима да остави рок који је за један мањи од тог минималног максималног збира и биће сигуран да неће сви ученици моћи да на време заврше домаћи (и то ће бити најдужи такав рок).

Минимални максимални збир пара можемо одредити грамзивим алгоритмом који упарује најспоријег и најбржег ученика и тиме своди проблем на проблем мање димензије (који се рекурзивно решава на исти начин). Када се исти принцип примени на преостале ученике, закључујемо да је један начин да добијемо оптимално упаривање да се други најбржи упари са другим најспоријим, трећи најбржи са трећим најспоријим и тако даље. Овим смо у потпуности одредили упаривање које може дати минималну максималну разлику и њену вредност одређујемо испитивањем збирова времена свих парова ученика.

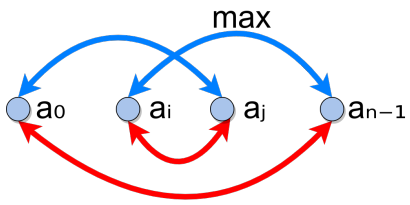
Имплементација се може направити итеративно. Прво сортирамо времена потребна да се уради домаћи неопадајуће. Након тога за свако $0 \leq i < \frac{n}{2}$ одређујемо збир $a_i + a_{n-1-i}$ и проналазимо максимум тих збирова.

```
int minimalniMaksimalniZbir(const vector<int>& a) {
  // pravimo sortiranu kopiju svih vremena
  auto as = a;
  sort(begin(as), end(as));
  // broj vremena
  int n = as.size();
  // minimalni maksimalni zbir se dobija u uparivanju u kojem je prvi
  // ucenik uparen sa poslednjim, drugi sa pretposlednjim itd.
  // odredjujemo maksimalni zbir nekog para u tom uparivanju
  int m = 0;
  for (int i = 0; i < n / 2; i++)
    m = max(m, as[i] + as[n - 1 - i]);

  return m;
}
```

Нека је низ $a_0, \ldots, a_{n-1}$ неопадајуће сортиран низ времена потребних да се уради домаћи. Тврдимо да постоји упаривање у ком се постиже минимални максимални збир, а у коме су најбржи и најспорији ученик упарени.

Размотримо упаривање у коме је најспорији ученик a_{n-1} упарен са неким учеником a_i, али који није најбржи ($i > 0$). Тада је најбржи ученик a_0 упарен са неким учеником a_j, али који није најспорији ($j < n - 1$). Ако сада направимо размену тако да упаримо ученике a_0 и a_{n-1} (најбржег и најспоријег) и ученике a_i и a_j, максимум се може само смањити. Остали парови остају непромењени, па је потребно посматрати само упаривања ова 4 ученика.



Стање пре размене и након размене. Збир $a_i + a_{n-1}$ је већи или једнак од $a_0 + a_j$, $a_0 + a_{n-1}$ и $a_i + a_j$, тако да се након размене максимум не може повећати

Важи да је $\max(a_0 + a_j, a_i + a_{n-1}) = a_i + a_{n-1}$, јер је $a_0 \leq a_i$ и $a_j \leq a_{n-1}$. Пошто је $a_0 \leq a_i$, важи да је $a_0 + a_{n-1} \leq a_i + a_{n-1}$, а пошто је $a_j \leq a_{n-1}$, важи да је $a_i + a_j \leq a_i + a_{n-1}$. Дакле, оба броја $a_0 + a_{n-1}$ и $a_i + a_j$ су мања или једнака од вредности $a_i + a_{n-1} = \max(a_0 + a_j, a_i + a_{n-1})$, па важи $\max(a_0 + a_{n-1}, a_i + a_j) \leq \max(a_0 + a_j, a_i + a_{n-1})$.

Дакле, ако се у неком оптималном упаривању изврши размена тако да се први и последњи елемент упаре, упаривање остаје оптимално (максимум се не може смањити, јер је полазно упаривање оптимално, а на основу доказаног не може се ни повећати). Према томе, постоји оптимално упаривање у коме су најбржи и најспорији ученик упарени.

Након сортирања које се врши у времену $O(n \log n)$, упаривање се одређује у времену $O(n)$.

Задатак: Игра - збир суседних пун квадрат

Задат је низ поља дужине n. У неким пољима је уписан по један позитиван цео број, док су остала поља празна. Потребно је попунити сва празна поља позитивним целим бројевима тако да је збир бројева у свака суседна два поља потпун квадрат. Написати програм који реализује алгоритам за одређивање лексикографски најмањег таквог решења, уколико решење постоји.

Опис улаза

Са стандардног улаза се учитавају дужина низа n ($1 \leq n \leq 22$) и максимални број који се може уписати у било које поље m ($1 \leq m \leq 10^4$). Затим се учитава n бројева уписаних у поља, при чему 0 означава празно поље.

Опис излаза

На стандардни излаз исписати лексикографски најмање решење игре уколико постоји, односно -1 уколико не постоји.

Пример

Улаз	*Излаз*
`5 20` `0 11 0 0 7`	`5 11 14 2 7`

Решење

Задатак можемо решити претрагом са повратком и одсецањем. На прву позицију у низу, ако већ није попуњена, могуће је уписати све вредности од 1 до m. Испробавамо једну по једну, док не наиђемо на прво решење. Након уписа вредности на почетну позицију, позивамо рекурзивну функцију која треба да попуни остатак низа.

Она прво проверава да ли је низ цео попуњен и ако јесте, прекида се поступак јер је пронађено тражено решење. Ако није, проверава се да ли је наредно поље које треба попунити већ попуњено тј. да ли на њему пише нешто различито од 0. Ако јесте, проверава се да ли је збир те вредности и претходне вредности потпун квадрат, и ако није, тренутна грана претраге се одсеца, а ако јесте, рекурзивно се прелази на попуњавање наредне позиције. Ако поље које треба попунити није већ попуњено, оно се може попунити свим вредностима од 1 до m. Може се покушати уписивање једне по једне од тих вредности. Када се упише вредности, проверава се да ли са вредношћу на претходној позицији чини потпун квадрат и ако чини, врши се наредни рекурзивни позив. Ово се може оптимизовати ако се уместо вредности a_i таквих да је $1 \leq a_i \leq m$ и да је $a_{i-1} + a_i = j^2$, за неко j размотре све вредности j такве да је $1 \leq j^2 - a_{i-1} \leq m$, тј. све вредности $\sqrt{1 + a_{i-1}} \leq j \leq \sqrt{m + a_{i-1}}$.

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

bool potpun_kvadrat(int x) {
  int koren = (int)round(sqrt(x));
  return koren * koren == x;
}
```

```
bool resi(vector<int>& v, int i, int m) {
  if (i == v.size())
    return true;

  if (v[i] > 0) {
    if (potpun_kvadrat(v[i] + v[i-1]))
      return resi(v, i + 1, m);
    else
      return false;
  }

  for (int j = (int)(sqrt(v[i-1])) + 1; j * j - v[i-1] <= m; j++) {
    v[i] = j * j - v[i-1];
    if (resi(v, i + 1, m))
      return true;
  }
  v[i] = 0;

  return false;
}

bool resi(vector<int>& v, int m) {
  if (v[0] > 0)
    return resi(v, 1, m);

  for (int j = 1; j <= m; j++) {
    v[0] = j;
    if (resi(v, 1, m))
      return true;
  }
  return false;

}

int main() {
  int n, m;
  cin >> n >> m;

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

  if (resi(v, m))
    for(int x : v)
      cout << x << ' ';
  else
    cout << -1;
  cout << '\n';

  return 0;
}
```

Глава 2

Септембар 2020.

Задатак: XOR

Задат је низ различитих неозначених целих бројева дужине n и неозначен цео број t. Потребно је одредити колико постоји парова бројева у низу таквих да је њихова ексклузивна дисјункција (*xor*) једнака управо броју t. Написати програм који реализује алгоритам за одређивање траженог броја парова. Временска сложеност алгоритма треба да буде $O(n \log n)$, а просторна $O(n)$.

За рачунање ексклузивне дисјункције у C++ користити бинарни оператор ^.

Опис улаза

Са стандардног улаза се учитавају дужина низа n ($n \leq 10^5$). Затим се учитава n неозначених целих бројева мањих од 2^{30} који представљају елементе низа. На крају се учитава и број t ($0 < t < 2^{30}$).

Опис излаза

На стандардни излаз исписати један број који представља тражени број парова.

Пример

Улаз	*Излаз*
5 1 4 5 2 6 3	2

Објашњење

Постоје 2 таква пара: $1 xor 2 = 3$ и $5 xor 6 = 3$.

Решење

Груба сила

Задатак се може једноставно решити грубом силом, тако што се испитају сви парови, израчуна вредност њихове екслузивне дисјунције и то упореди са жељеним бројем t.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

  vector<int> v(n);
  for(int i = 0; i < n; i++)
```

```
    cin >> v[i];

  int t;
  cin >> t;

  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      if ((v[i] ^ v[j]) == t) {
        broj++;
        break;
      }

  cout << broj << endl;
  return 0;
}
```

Бинарна претрага

Обежелимо операцију ексклузивне дисјункције са $\oplus$. Лако се утврђује да је то асоцијативна операција и да за свако x важи $x \oplus x = 0$ и $0 \oplus x = x$. Зато, ако важи $x \oplus y = t$, важи и $x \oplus (x \oplus y) = x \oplus t$. Важи да је $x \oplus (x \oplus y) = (x \oplus x) \oplus y = 0 \oplus y = y$. Зато из $x \oplus y = t$ следи да је $y = x \oplus t$, што значи да за свако x у низу морамо проверити да ли низ садржи и вредност $x \oplus t$. Пошто је t различито од нуле, вредности x и $x \oplus t$ ће увек бити различите.

Проналажење одговарајућих парова ефикасно можемо урадити ако низ сортирамо и затим на њега применимо бинарну претрагу. Пошто ће се сваки пар пронаћи два пута (једном за први, а једном за други елемент пара), укупан број пронађених парова треба на крају поделити са 2.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

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

  int t;
  cin >> t;

  sort(v.begin(), v.end());

  int broj = 0;
  for(int i = 0; i < n; i++)
    if (binary_search(v.begin(), v.end(), t ^ v[i]))
      broj++;

  cout << broj / 2 << endl;
  return 0;
}
```

Задатак: Лексикографски наредна шетња

Шетња је низ бројева такав да је први елемент једнак 0 и да се свака два суседна елемента низа разликују за највише 1. Дата је шетња дужине n. Написати програм који реализује алгоритам за одређивање лексикографски наредне шетње. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се учитава број n ($n \leq 10^6$). Након тога се учитава n бројева који представљају елементе шетње.

Опис излаза

На стандардни излаз исписати n бројева који представљају наредну шетњу. Уколико таква шетња не постоји, исписати -1.

Пример

Улаз	*Излаз*
`6` `0 1 0 -1 -2 -1`	`0 1 0 -1 -1 -2`

Решење

Алгоритам подсећа на алгоритам одређивања наредне варијације у лексикографском редоследу.

Анализирамо елемент по елемент низа, здесна налево и тражимо први елемент који је могуће повећати за један. Да би то могло да се уради мора да важи $a_j + 1 - a_{j-1} \leq 1$. Ако такав елемент не постоји (што ће се догодити у случају да је шетња облика $0, 1, 2, \ldots, n-1$), тада не постоји наредна варијација. У супротном увећавамо пронађени елемент a_j за један, а након њега правимо шетњу која се састоји од што мањих бројева (да би била лексикографски најмања), тако што иза a_j постављамо редом елементе $a_j - 1$, $a_j - 2$, ….

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

using namespace std;

bool naredna_setnja(vector<int>& a) {
  int n = a.size();
  int j = n-1;
  while (j > 0 && a[j] + 1 - a[j-1] > 1)
    j--;
  if (j == 0)
    return false;
  a[j]++;
  for (int k = j+1; k < n; k++)
    a[k] = a[k-1] - 1;
  return true;
}

int main() {
  ios_base::sync_with_stdio(false);

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

  if (naredna_setnja(a)) {
    for (int i = 0; i < n; i++)
      cout << a[i] << " ";
    cout << endl;
  } else
```

```
    cout << "-1" << endl;

  return 0;
}
```

Задатак: Једнобојан квадрат

Марко је желео да купи једнобојан тепих у облику квадрата, али како су у радњи имали само шарене тепихе у облику правоугаоника одлучио је да купи један такав и да из њега исече један једнобојан квадратни део. Он жели да га исече тако да површина добијеног тепиха буде што већа. Тепих је задат матрицом димензија $n \times m$ при чему је у свако поље матрице уписан један број који представља боју тог поља. Написати програм који реализује алгоритам за одређивање дужине странице највећег једнобојног квадрата који се може пронаћи на датом тепиху. Временска и просторна сложеност алгоритма треба да буду $O(n \cdot m)$.

Опис улаза

Са стандардног улаза се учитавају бројеви n и m ($n, m \leq 1000$). Затим се учитава матрица бројева величине $n \times m$ при чему је сваки елемент матрице неозначен цео број мањи или једнак 10^9.

Опис излаза

На стандардни излаз исписати један број који представља дужину странице највећег једнобојног квадрата.

Пример

Улаз	*Излаз*
3 4 1 5 3 0 8 3 3 4 2 3 3 4	2

Објашњење

Највећи једнобојан квадрат је боје 3 и његова страница је дужине 2.

Решење

Динамичко програмирање

За сваку позицију у матрици у другој матрици памтимо димензију максималног квадрата коме је доње десно теме на тој позицији. За поља у првој колони и првој врсти матрице, важи да су максималне димензије таквих квадрата једнаке 1. Да би се на позицији (i, j) завршавао квадрат димензије веће од 1, потребно је да су сва поља $(i-1, j)$, $(i, j-1)$ и $(i-1, j-1)$ обојене истом бојом као поље (i, j). Тада дмензију максималног квадрата добијамо тако што за 1 увећамо димензију најмањег од три квадрата који имају доње десно теме на пољима $(i-1, j)$, $(i, j-1)$ и $(i-1, j-1)$. Заиста, димензија квадрата на пољу $(i-1, j)$ нам говори колико квадрат може да се прошири навише, димензија квадрата на пољу $(i, j-1)$ нам говори колико квадрат може да се прошири налево, док нам димензија квадрата на пољу $(i-1, j-1)$ говори да ли квадрат може да дохвати и поље које је дијагонално.

У наредном примеру је максимални квадрат димензије 4, јер се на пољу горе-лево од њега налази 3 (остала два поља допуштају и веће квадрате). Иако се квадрат димензије 5 пута 5 може направити ако се гледају поља горе и поља лево, дијагонално поље које недостаје то спречава.

```
....xxxx...   ....1111...
...xxxxx...   ...11222...
...xxxxx...   ...12233...
...xxxxx...   ...12334...
...xxxxx...   ...12344...
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

```cpp
int main() {
  int n, m;
  cin >> n >> m;

  vector< vector<int> > mat(n, vector<int>(m));
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      cin >> mat[i][j];

  vector<vector<int>> dp(n, vector<int>(m));
  for (int i = 0; i < n; i++)
    dp[i][0] = 1;
  for (int j = 0; j < m; j++)
    dp[0][j] = 1;
  for (int i = 1; i < n; i++)
    for (int j = 1; j < m; j++)
      if (mat[i-1][j]   == mat[i][j] &&
          mat[i][j-1]   == mat[i][j] &&
          mat[i-1][j-1] == mat[i][j])
        dp[i][j] = 1 + min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
      else
        dp[i][j] = 1;

  int maxd = 0;
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      maxd = max(maxd, dp[i][j]);

  cout << maxd << '\n';
  return 0;
}
```

Глава 3

Октобар 2020.

Задатак: Кајмак

Алекса жели да отпутује један дан до Златибора да купи кајмак. Он толико воли кајмак да ће сваки дан током свог боравка на Златибору купити сав кајмак са пијаце. Пошто путује аутомобилом у који може да смести строго мање од t килограма кајмака, на Златибору ће остати до оног дана када би куповина кајмака препунила ауто. Написати програм који одређује највећи број килограма кајмака који Алекса може да купи.

Опис улаза

Са стандардног улаза се учитавају број n ($n \leq 10^6$) и након тога се учитава n бројева који представљају колико ће килограма кајмака бити на пијаци ког дана. Збир тих бројева није већи од 10^9. На крају се учитава број t ($t \leq 10^9$), који представља носивост аутомобила.

Опис излаза

На стандардни излаз исписати један број који представља колико највише килограма кајмака Алекса може да донесе са Златибора.

Пример

Улаз	*Излаз*
6 4 2 3 2 10 1 9	7

Објашњење

Објашњење

Алекса ће највише кајмака купити ако оде на Златибор другог дана, где ће дан за даном купити 2, 3 и 2 килограма, након чега се враћа за Београд.

Решење

Груба сила

Задатак грубом силом решавамо тако што за сваку почетни дан одређујемо количину кајмака која се може купити ако би летовање кренуло баш тог дана. То радимо тако што сабирамо све количине кајмака, док се не достигне вредност t.

Сложеност овог решења је $O(n^2)$.

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

using namespace std;

int main() {
```

```
  ios_base::sync_with_stdio(false);

  int n;
  cin >> n;

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

  int t;
  cin >> t;

  int max_s = 0;
  for(int i = 0; i < n; i++) {
    int s = 0;
    for(int j = i; j < n; j++) {
      s += v[j];
      if (s < t)
        max_s = max(max_s, s);
      else
        break;
    }
  }

  cout << max_s << endl;
  return 0;
}
```

Два показивача

Задатак можемо решити ефикасно техником два показивача. Одржаваћемо текући распон дана $[j, i]$ и укупну количину кајмака у тим данима. Иницијално ћемо посматрати интервал $[0, 0]$. Док год је укупна количина кајмака у тим данима строго мања од t помераћемо десни показивач i и ажурираћемо максимум ако је то потребно. Када количина постане већа или једнака t, помераћемо леви показивач j све док се количина не спусти испод t или док j не достигне i.

Овим се врши значајно одсецање, јер се за многе потенцијалне почетке j не одређује која је максимална укупна количина кајмака била када би тај j био први дан. То је безбедно урадити, јер смо сигурни да се максимум не може достићи за тако елиминисане почетке. Наиме, у ситуацији када померамо j, знамо да је збир елемената из интервала $[j, i]$ већи или једнак t, док је збир елемената из интервала $[j, i-1]$ строго мањи од t (у супротном не би био померен десни показивач i). Када се j помери на позицију $+1$ збир свих елемената из интервала $[j+1, i]$ може бити и строго мањи, али и већи или једнак од t. Ако је строго мањи, тада га упоређујемо са максимумом и ажурирамо максимум ако је то потребно. Ако је већи или једнак, то значи да се максимум кајмака кренувши од дана $j+1$ постиже закључно са неким даном који је или $i-1$ или неки раније. Међутим, тај интервал је подинтервал интервала $[j, i-1]$, па му збир не може бити већи од збира елемената интервала $[j, i-1]$ који је раније обрађен. Стога тај збир не треба прецизно одређивати и може се одмах прећи на наредну вредност j.

Пошто се оба показивача померају само у једном смеру и пошто се у сваком кораку врши само константан број операција сложеност овог решења је $O(n)$.

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int n;
```

```
  cin >> n;

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

  int t;
  cin >> t;

  int s = 0, j = 0, maks = 0;
  for(int i = 0; i < n; i++) {
    s += v[i];
    while(j <= i && s >= t)
      s -= v[j++];
    maks = max(maks, s);
  }

  cout << maks << '\n';
  return 0;
}
```

Префиксни збирови и два показивача

Задатак се ефикасно може решити тако што се израчунају збирови свих префикса низа. Ако је Z_i збир првих i елемената низа, тада се збир елемената на позицијама из интервала $[i, j]$ може израчунати као $Z_{j+1} - Z_i$. Дакле, потребно је пронаћи највећу разлику два елемента низа која је строго мања од t. То је могуће ефикасно урадити техником два показивача. Пошто су елементи низа позитивни, низ збирова префикса је сортиран растући. Одржавамо два показивача i и j које иницијализујемо на 0. Када је разлика $Z_j - Z_i$ (а то је збир елемената интервала $[i, j-1]$ већи од или једнак t, померамо показивач i (то се не може догодити када је $i = j$, тј. када је интервал $[i, j-1]$ празан). У супротном ажурирамо максимум ако је то потребно и проширујемо интервал померајући десни показивач j.

Низ збирова префикса се формира у времену $O(n)$. Пошто се оба показивача померају само у једном смеру и пошто се у сваком кораку врши само константан број операција сложеност овог решења је $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int n;
  cin >> n;

  vector<int> ps(n+1);
  ps[0] = 0;
  for(int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    ps[i] = ps[i-1] + x;
  }

  int t;
  cin >> t;

  int maks = 0, i = 0, j = 0;
  while (j <= n) {
```

```
    if (ps[j] - ps[i] >= t) {
      i++;
    } else {
      maks = max(maks, ps[j] - ps[i]);
      j++;
    }
  }

  cout << maks << '\n';
  return 0;
}
```

Префиксни збирови и бинарна претрага

Задатак можемо решити тако што се за сваки леви крај i одреди највећи десни крај j такав да је збир елемената у интервалу $[i, j]$ строго мањи од t. Ако се формира низ збирова префикса (који је растући сортиран, јер су количине кајмака позитивне), задатак се своди на то да се за свако i одреди највеће вредности $j \geq i$ тако да је $z_j - z_j < t$, што се може урадити бинарном претрагом и проналажењем најмање вредности $j \geq i$ такве да је $z_j - z_i \geq t$ тј. да је $z_j \geq z_i + t$.

Низ збирова префикса се формира у времену $O(n)$. Након тога се врши n бинарних претрага, свака у сложености $O(\log n)$, па је укупна сложеност $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int n;
  cin >> n;

  vector<int> ps(n+1);
  ps[0] = 0;
  for(int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    ps[i] = ps[i-1] + x;
  }

  int t;
  cin >> t;

  int maks = 0;
  for (int i = 0; i <= n; i++) {
    int psj = *prev(lower_bound(next(begin(ps), i), end(ps), ps[i] + t));
    maks = max(maks, psj - ps[i]);
  }

  cout << maks << '\n';
  return 0;
}
```

Задатак: Кретање низ матрицу

Дата је матрица целих бројева димензије $n \times m$. Могуће је започети кретање из било ког поља прве врсте матрице и завршити га у било ком пољу последње врсте. У сваком кораку је могуће прећи само на поље доле, доле-десно или доле-лево у односу на тренутно. Написати програм који реализује алгоритам за одређивање максималног збира бројева у пољима на путу који се може остварити током једне шетње. Временска и просторна сложеност алгоритма треба да буду $O(nm)$.

Опис улаза

Са стандардног улаза се учитавају бројеви n и m ($n, m \leq 1000$). Након тога се учитава матрица целих бројева димензије $n \times m$. Бројеви су мањи од 10^8.

Опис излаза

На стандардни излаз исписати један број који представља тражени максимални збир.

Пример

Улаз	*Излаз*
4 3 1 2 3 2 -4 2 3 0 -1 1 2 3	9

Објашњење

Највећи збир се остварује кретањем редом кроз друго поље прве врсте (2), прво поље друге врсте (2), прво поље треће врсте (3) и коначно друго поље четврте врсте (2).

Решење

Задатак решавамо динамичким програмирањем. У матрици динамичког програмирања, на пољу (i, j) чувамо максимални збир који се може покупити ако се крене из било ког поља прве врсте и заврши се на пољу (i, j). Прва врста матрице динамичког програмирања се поклапа са првом врстом матрице (пошто у сваком кораку морамо да се спустимо на следећу врсту, ако кретање завршавамо на пољу из прве врсте, тада не правимо ни један корак, па само купимо вредност која се налази на том пољу). Што се осталих поља тиче, анализирамо могућност да на њих стигнемо са поља горе-лево ($(i-1, j-1)$), поља горе ($(i-1, j)$) и поља горе-десно ($(i-1, j+1)$) и најбољу од те три могућности увећавамо за вредност на тренутном завршном пољу (i, j). При том морамо водити рачуна о рубним случајевима (прва и последња колона), када једна од те три могућности није доступна.

Пошто се кретање може завршити у било ком пољу последње врсте, крајњи резултат се одређује као максимум вредности уписаних у последњу врсту матрице динамичког програмирања.

Приказана је улазна матрица и њој одговарајућа матрица динамичког програмирања.

```
1  2  3        1  2  3
2 -4  2        4 -1  5
3  0 -1        7  5  4
1  2  3        8  9  8
```

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n, m;
  cin >> n >> m;

  vector<vector<int>> mat(n, vector<int>(m));
```

```
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      cin >> mat[i][j];

  vector<vector<int>> dp(mat);
  for (int i = 1; i < n; i++)
    for (int j = 0; j < m; j++) {
      int mp = dp[i - 1][j];
      if (j > 0)
        mp = max(mp, dp[i - 1][j - 1]);
      if (j < m - 1)
        mp = max(mp, dp[i - 1][j + 1]);
      dp[i][j] = mp + mat[i][j];
    }

  int res = *max_element(begin(dp[n - 1]), end(dp[n - 1]));
  cout << res << endl;

  return 0;
}
```

Задатак: Распоред са најмањим максималним збиром три узастопна

Ана и Бобан играју игру. Игра се игра са n папирића и на сваком је написан по један број. Ана почиње партију тако што папириће поређа у круг у произвољном редоследу. Бобан затим бира нека три узастопна папирића и рачуна збир бројева написаних на њима. Бобан осваја број бодова једнак том збиру и циљ му је да одабере три папирића тако да освоји што више бодова. Са друге стране Ана распоређује папириће тако да Бобан може да оствари што мање бодова. Написати програм који реализује алгоритам за одређивање броја бодова које ће Бобан освојити уколико оба играча играју савршено. Временска сложеност алгоритма треба да буде $O((n-1)!)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се учитава број n ($n \leq 12$). Затим се учитава n различитих позитивних бројева поређаних у неопадајући поредак који представљају бројеве написане на папирићима. Бројеви нису већи од 10^8.

Опис излаза

На стандардни излаз исписати један број који представља максималан број бодова које ће Бобан освојити.

Пример

Улаз	*Излаз*
5 2 3 4 5 6	13

Објашњење

Ана може да распореди папириће на следећи начин: $[2, 6, 4, 3, 5]$. Бобан тада може на два начина да оствари збир 13, као $5 + 2 + 6$ и као $6 + 4 + 3$. Не постоји распоред у ком Бобан може максимално да скупи мање од 13 бодова.

Решење

Испитивање свих пермутација

Решење грубом силом подразумева да се испитају све пермутације и да се за сваку пермутацију испитају све тројке узастопних елемената.

Сложеност овог решења је $O(n \cdot n!)$.

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

```cpp
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

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

  int res = v[n - 1] + v[n - 2] + v[n - 3];
  do {
    int maxsum = 0;
    for (int i = 0; i < n; i++)
      maxsum = max(maxsum, v[i] + v[(i + 1) % n] + v[(i + 2) % n]);
    res = min(res, maxsum);
  } while (next_permutation(v.begin(), v.end()));

  cout << res << '\n';
  return 0;
}
```

Оптимизовано испитивање свих пермутација

Размотримо све пермутације 4 елемента.

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

Иако су све ове пермутације различите, сваки кружни распоред папирића је представљен са 4 различите пермутације. На пример, пермутације `1 2 3 4`, затим `2 3 4 1`, затим `3 4 1 2` и на крају `4 1 2 3` све представљају исти распоред папирића. Време се може уштедети ако се ово избегне и ако се за сваки распоред анализира само једна пермутација. То на пример увек може бити она која има број 1 на почетку.

Сложеност овог решења је $O(n \cdot (n-1)!)$. Да би се смањио константни фактор у имплементацији је пожељно избегавати целобројно дељење.

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

using namespace std;

int main() {
  int n;
  cin >> n;

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

  int res = v[n - 1] + v[n - 2] + v[n - 3];
  do {
    int maxsum = 0;
    for (int i = 0; i < n; i++) {
```

```
      int i1 = i + 1;
      if (i1 >= n)
        i1 -= n;
      int i2 = i + 2;
      if (i2 >= n)
        i2 -= n;
      maxsum = max(maxsum, v[i] + v[i1] + v[i2]);
    }
    res = min(res, maxsum);
  } while(next_permutation(v.begin() + 1, v.end()));

  cout << res << '\n';
  return 0;
}
```

Глава 4

Јануар 2021. - група А

Задатак: Не-нула збир

Написати програм који за унети низ целих бројева a дужине n одређује колико постоји парова позиција i и j ($i < j$) за које важи $a_i + a_j \neq 0$. Временска сложеност алгоритма треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$), а затим и n целих бројева a_i ($-10^9 \leq a_i \leq -10^9$).

Опис излаза

На стаднардни излаз исписати тражени број парова. Користити 64-битни тип података.

Пример

Улаз	*Излаз*
4	4
3 4 -3 -3	

Објашњење

У питању су парови $(0, 1)$ $(3 + 4)$, $(1, 2)$ $(4 + (-3))$, $(1, 3)$ $(4 + (-3))$ и $(2, 3)$ $((-3) + (-3))$.

Решење

Груба сила

Решење грубом силом подразумева испитавање свих парова елемената.

Сложеност овог решења је (n^2) и оно је јако неефикасно.

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

using namespace std;

int main() {
  int n;
  cin >> n;

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

  int64_t broj = 0;
  for(int i = 0; i < n; i++)
    for(int j = i + 1; j < n; j++)
```

```
      if(v[i] + v[j] != 0)
        broj++;

  cout << broj << '\n';
  return 0;
}
```

Бројање појављивања елемената

Уместо да одредимо колико има парова где збир није нула, лакше можемо да одредимо колико има парова где збир јесте нула. За сваки елемент $_i$ важи да је број позиција j пре њега таквих да је $a_i + a_j = 0$ једнак је броју појављивања вредности $-a_i$ у делу низа на позицијама $[0, i)$. Због тога користимо асоцијативни низ (мапу тј. речник) помоћу којег бројимо појављивања свих елемената низа. Када израчунамо број парова који дају збир нула, број парова који не дају збир нула можемо добити одузимањем тог броја од укупног броја парова који је једнак $\binom{n}{2} = \frac{n(n-1)}{2}$.

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;

  unordered_map<int, int> bp; // broj pojaviljivanja do sada ucitanih elemenata
  int64_t broj_parova_0 = 0; // broj parova ciji je zbir jednak nuli
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    broj_parova_0 += bp[-x];
    bp[x]++;
  }
  int64_t ukupan_broj_parova = (int64_t)n * (n - 1) / 2;
  int64_t broj_parova_ne_0 = ukupan_broj_parova - broj_parova_0;
  cout <<  broj_parova_ne_0 << '\n';
  return 0;
}
```

Задатак: Скраћивање ниске

Над ниском се извршава следећа операција докле год је то могуће: одаберу се два једнака узастопна слова и обришу се из ниске. Написати програм који за унету ниску дужине n одређује ниску која се добија након извршења свих операција. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уноси ниска састављена од малих слова енглеског алфабета дужине n ($1 \leq n \leq 10^6$).

Опис излаза

На стандардни излаз исписати резултујућу ниску.

Пример

Улаз	*Излаз*
`abceeccbaaba`	`abca`

Решење

Брисање карактера са почетка или из средине ниске захтева померање осталих карактера, што доводи до неефикасног програма. Уместо тога, могуће је креирати нову ниску обрађујући један по један карактер полазне

ниске. Претпоставићемо да смо обрадили првих k карактера ниске и да смо брисањем свих појављивања узастопних једнаких карактера добили ниску t. Ако је ниска t празна или ако је последњи карактер ниске t различит од текућег карактера s_k ниске s (оног на позицији k), карактер s_k треба додати на t, док у супротном (ако је последњи карактер ниске t једнак карактеру s_k) треба уклонити последњи карактер ниске t и тиме (у оба случаја) добијамо резултат обраде првих $k+1$ карактера ниске s. Приметимо да се ниска t понаша као стек (карактери јој се додају и уклањају са десног краја). Тип `string` у језику C++ подржава методе `empty`, `back`, `push_back` и `pop_back` који се извршавају у сложености $O(1)$, па се тај тип може користити као стек карактера.

Пошто су све операције за рад са стеком сложености $O(1)$ и сваки карактер се највише једном може додати и једном може уклонити са стека, сложеност овог алгоритма је $O(n)$. И меморијска сложеност је такође $O(n)$.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string str;
  cin >> str;

  string s;
  for(char c : str)
    if(s.empty() || s.back() != c)
      s.push_back(c);
    else
      s.pop_back();

  cout << s << endl;

  return 0;
}
```

Задатак: Степенасти низ

Написати програм који исписује све степенасте низове дужине n у лексикографском поретку. Низ је степенаст ако почиње бројем 1 и сваки наредни елемент низа је или једнак претходном или за један већи од њега.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 18$).

Опис излаза

На стандардни излаз исписати све степенасте низове дужине n, сваки у засебном реду.

Пример

Улаз	*Излаз*
3	1 1 1
	1 1 2
	1 2 2
	1 2 3

Решење

Задатак решавамо рекурзивним набрајањем тражених комбинаторних објеката. Техника је веома слична набрајању варијација. На прву позицију постављамо 1, а на сваку наредну или елемент једнак оном на претходној позицији или елемент за један већи од њега. Након постављања елемента рекурзивно настављамо попуњавање низа све док не буде потпуно попуњен, када га исписујемо.

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

```cpp
using namespace std;

void ispisi(const vector<int>& v) {
  for(int x : v)
    cout << x << ' ';
  cout << '\n';
}

void stepenasti(vector<int>& v, int i) {
  // ceo niz je popunjen, pa ga ispisujemo
  if (i == v.size()) {
    ispisi(v);
    return;
  }

  // niz nije popunjen, pa ga dopunjujemo narednim elementom

  // naredni element je jednak prethodnom
  v[i] = v[i - 1];
  stepenasti(v, i + 1);

  // naredni element je od prethodnog veci za jedan
  v[i] = v[i - 1] + 1;
  stepenasti(v, i + 1);
}

// stampa sve stepenaste nizove duzine n
void stepenasti(int n) {
  vector<int> v(n);
  v[0] = 1;
  stepenasti(v, 1);
}

int main() {
  int n;
  cin >> n;
  stepenasti(n);

  return 0;
}
```

Задатак: Распоред кућа

Дато је n плацева поређаних у низ и за сваки је позната приход од издавања кућа изграђених на њему. Свака кућа је дужине 1 или 2 плаца. Приход од издавања куће одређен је производом њене дужине и највећег прихода плацева на којима се налази. У циљу што већих профита потребно је распоредити куће тако да сви плацеви буду заузети и да укупан приход од издвања свих кућа буде што већи.

Написати програм који за задати низ плацева дужине n одређује максимални укупан приход. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($2 \leq n \leq 10^6$). Затим се уноси n бројева p_i ($1 \leq p_i \leq 100$) који представљају приходе од издавња сваког плаца.

Опис излаза

На стандардни излаз исписати један број који представља тражени максимални приход.

Пример

Улаз	*Излаз*
5	16
1 4 2 1 3	

Објашњење

Најбоље је поставити прво кућу дужине 2, затим кућу дужине 1 и на крају поново кућу дужине 2. Прва кућа има приход 8, друга има приход 2, а трећа има приход 6.

Решење

Задатак можемо решити динамичким програмирањем. За сваки плац одређујемо максимални приход који би се могао добити ако би тај плац био последњи у низу.

Ако имамо само један плац, тада је укупан приход једнак приходу тог плаца.

Ако имамо само два плаца, тада се највише исплати направити кућу дужине 2 (јер је приход појединачних кућа једнак $p_0 + p_1$, док је приход једне куће $2 \max(p_0, p_1)$, што сигурно није мање (јер је један сабирак исти као у претходном збиру, а други већи или једнак).

Ако имамо више од два плаца, резултат одређујемо индуктивно. Ако на плацу на позицији i направимо кућу дужине 1, онда је укупан приход једнак збиру прихода тог плаца и максималног прихода који се може добити завршно са претходним плацом (а за њега можемо сматрати да га већ имамо на располагању). Ако направимо кућу дужине 2, онда је укупан приход једнак збиру двоструког максималног прихода плацева на позицијама $i-1$ и i и максималног прихода који се може добити ако се разматрају плацеви закључно са оним на позицији $i-2$ (а и за тај приход можемо сматрати да га већ имамо на располагању).

За низ прихода `1 4 2 1 3` добија се следећи садржај низа динамичког програмирања тј. следеће максималне вредности прихода за сваку позицију завршног плаца.

- Ако имамо само први плац, приход је 1.
- Са два плаца можемо направити кућу дужине 2 и добити приход $2 \cdot 4 = 8$.
- Ако на трећем плацу направимо самосталну кућу, добијамо приход $8 + 2 = 10$, а ако направимо кућу дужине 2, добијамо приход $1 + 2 \cdot 4 = 9$. Дакле, боље је направити једноструку кућу.
- Ако на четвртом плацу направимо самосталну кућу, добијамо приход $10 + 1 = 11$, а ако направимо кућу дужине 2, добијамо приход $8 + 2 \cdot 2 = 12$. Дакле, боље је направити двоструку кућу.
- Ако на петом плацу направимо самосталну кућу, добијамо приход $12 + 3 = 15$, а ако направимо кућу дужине 2, добијамо приход $10 + 2 \cdot 3 = 16$. Дакле, боље је направити двоструку кућу и коначан резултат је 16.

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

using namespace std;

int main() {
  int n;
  cin >> n;

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

  vector<int> dp(n);
  dp[0] = v[0];
  dp[1] = 2 * max(v[0], v[1]);
  for(int i = 2; i < n; i++)
    dp[i] = max(v[i] + dp[i - 1], 2 * max(v[i], v[i - 1]) + dp[i - 2]);

  cout << dp[n - 1] << '\n';
```

```
  return 0;
}
```

Глава 5

Јануар 2021. - група Б

Задатак: Број елемената који се појављују паран број пута

Написати програм који за задати низ целих бројева дужине n одређује колико постоји оних бројева који се у том низу појављују паран број пута. Временска сложеност алгоритма треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). Затим се уноси n бројева a_i ($-10^9 \leq a_i \leq 10^9$).

Опис излаза

На стандардни излаз исписати број различитих бројева који се у низу појављују паран број пута.

Пример

Улаз	*Излаз*
9 2 -3 1 2 -3 -3 -3 1 1	2

Објашњење

Објашњење: Бројеви 2 и -3 се једини појављују паран број пута у низу.

Решење

Пребројавање елемената

Задатак можемо решити тако што једноставно коришћењем мапе избројимо појављивања сваког броја, а затим у посебном пролазу видимо колико се од тих елемената јављало паран број пута. Задатак можемо решити и у једном пролазу тако што сваки пут пре него што увећамо бројач проверимо да ли је вредност парна или непарна – ако је парна и позитивна, број парних елемената се умањује за 1 (јер се елемент који се раније јављао паран број сада јавља непаран број пута), а ако је непарна, увећава се за један (јер се сада тај елемент јавља паран број пута).

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  int n, x;
  cin >> n;

  unordered_map<int, int> m;
  int cnt = 0;
  while(n--) {
```

```
    cin >> x;
    if (m[x] > 0) {
      if(m[x] % 2 == 0)
        cnt--;
      else
        cnt++;
    }
    m[x]++;
  }

  cout << cnt << '\n';
  return 0;
}
```

Сортирање

Пребројавање елемената се може олакшати ако се низ сортира, јер се тада једнаки елементи појављују узастопно. Након тога у једном пролазу обрађујемо серије једнаких узастопних елемената и након завршетка сваке проверавамо да ли је њена дужина била парна или не.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

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

  sort(v.begin(), v.end());

  int cnt = 0;
  int cur = 1;
  for (int i = 1; i < n; i++)
    if (v[i] == v[i-1])
      cur++;
    else {
      if(cur % 2 == 0)
        cnt++;
      cur = 1;
    }
  if (cur % 2 == 0)
    cnt++;

  cout << cnt << '\n';
  return 0;
}
```

Задатак: Индуктивни скуп

Бесконачан скуп S_n дефинисан је као најмањи скуп за који важи:

- $n \in S_n$;
- ако је $x \in S_n$, онда је $2x \in S_n$ и $4x - 3 \in S_n$.

Написати програм који за задато n и k исписује најмањих k бројева скупа S_n. Временска сложеност алгоритма треба да буде $O(k \log k)$, а просторна $O(k)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($2 \leq n \leq 100$) и k ($1 \leq k \leq 10^5$).

Опис излаза

На стандардни излаз у једном реду исписати најмањих k елемената скупа S_n, раздвојене размаком.

Пример

Улаз	*Излаз*
`2 5`	`2 4 5 8 10`

Решење

За сваки елемент x у скупу се налазе и елементи $2x$ и $4x-3$. Кренимо, на пример, од елемента 2. Он узрокује додавање елемената 4 и 5. Елемент 4 узрокује додавање елемената 8 и 13, док елемент 5 узрокује додавање елемената 10 и 17. Елемент 8 узрокује додавање елемената 16 и 29 итд. Зато су елементи низа 2, 4, 5, 8, 10, 13 итд.

Одржаваћемо скуп елемената за које смо на основу до сада обрађених елемената утврдили да треба да буду чланови низа. Најмањи елемент тог скупа је сигурно наредни елемент низа. Заиста, сви елементи у скупу ће бити већи од свих до тог тренутка исписаних и обрађених елемената низа. Лако се може доказати да су сви елементи у скупу већи или једнаки од 2, као и да за $x \geq 2$ важи да је $2x > x$ и да је $4x - 3 > x$. Зато ниједан тренутни елемент скупа не може имати наследнике који би били мањи од тренутно најмањег елемента скупа.

На основу овога можемо једноставно направити имплементацију коришћењем ефикасних структура података (скупа или реда са приоритетом). Скуп иницијализујемо на вредност n а затим му у сваком вадимо минимални елемент x и уместо њега додајемо вредности $2x$ и $4x - 3$.

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main() {
  int n, k;
  cin >> n >> k;
  priority_queue<int, vector<int>, greater<int> > pq;
  pq.push(n);
  for (int i = 0; i < k; i++) {
    int x = pq.top();
    pq.pop();
    cout << x << ' ';
    pq.push(2 * x);
    pq.push(4 * x - 3);
  }
  return 0;
}
```

Задатак: Бројеви у основи 4 без суседних непарних цифара

Написати програм који исписује у лексикографском поретку све бројеве са n цифара у систему са основом 4 који немају две суседне непарне цифре.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 12$).

Опис излаза

На стандардни излаз исписати све тражене бројеве, сваки у засебном реду.

Пример

Улаз	*Излаз*
2	10 12 20 21 22 23 30 32

Решење

Задатак се једноставно решава рекурзивним набрајањем тражених комбинаторних објеката. Алгоритам је једноставна модификација алгоритма за генерисање варијација.

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

using namespace std;

void ispisi(const vector<int>& v) {
  for(int x : v)
    cout << x;
  cout << '\n';
}

void generisi(vector<int>& v, int i) {
  // niz od n cifara je popunjen, pa ispisujemo broj
  if (i == v.size()) {
    ispisi(v);
    return;
  }

  // niz nije popunjen do kraja, pa popunjavamo poziciju i, pa
  // rekurzivno popunjavamo ostatak

  // na mesto i stavljamo cifre 0, 1, 2 i 3, pri čemu cifru 0 ne smemo
  // staviti na početnu poziciju
  for(int j = (i == 0 ? 1 : 0); j < 4; j++) {
    v[i] = j;
    // nije dozvoljeno da dve uzastopne cifre budu neparne
    if(i > 0 && v[i] % 2 == 1 && v[i - 1] % 2 == 1)
      continue;
    // rekurzivno popunjavamo ostatak
    generisi(v, i + 1);
  }
}

// ispisuje sve n-tocifrene brojeve u osnovi 4 koji nemaju dve
// uzastopne neparne cifre
void generisi(int n) {
  vector<int> v(n);
  generisi(v, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
```

```
  generisi(n);
  return 0;
}
```

Задатак: Вагони са 4 седишта

Возић у луна-парку се састоји од вагона који имају по 4 седишта. Испред возића људи чекају да се укрцају. Неки људи су сами, неки су у групама, али се зна да ни у једној групи нема више од 4 особе. У возић се укрцавају редом, при чему сви људи из исте групе морају бити у истом вагону, а допуштено је и да више група дели исти вагон. Напиши програм који одређује број начина да се они распореде.

Опис улаза

Са стандардног улаза се учитава број група људи n ($1 \leq n \leq 10^5$), а затим из наредног реда и број људи у свакој од n група (природни бројеви између 1 и 4).

Опис излаза

Пошто број распореда може бити јако велики, на стандардни излаз исписати његов остатак при дељењу са $10^9 + 7$.

Пример

Улаз	*Излаз*
`6` `1 2 3 1 2 3`	`6`

Објашњење

Бројеви људи у вагонима могу бити `1 2 3 1 2 3`, `1 2 3 3 3`, `1 2 4 2 3`, `3 3 1 2 3`, `3 3 3 3`, `3 4 2 3`.

Решење

Рекурзивна формулација

Задатак се рекурзивно може формулисати на следећи начин.

Основна идеја је да сместимо неке групе људи са почетка низа у вагоне и да онда рекурзивно смештамо преостале групе људи. Рекурзивна функција зато прима позицију прве нераспоређене групе људи и има задатак да израчуна број начина да се све нераспоређене групе распореде.

Излаз из рекурзије је када нема више нераспоређених група, тј. када је индекс k прве нераспоређене групе једнак дужини низа (празан скуп људи се може распоредити на један начин – нико не улази у вагоне). У супротном крећемо од позиције k и прво само групу a_k смештамо у вагон и рекурзивно распоређујемо остале групе (од позиције $k + 1$). Након тога, ако је то могуће, у вагон смештамо групе a_k и a_{k+1} и рекурзивно распоређујемо остале групе (од позиције $k + 2$). Прелазимо затим на три групе a_k, a_{k+1} и a_{k+2} и тако све док не дођемо у ситуацију да смо у текући вагон распоредили све преостале групе или да је текући вагон комплетно попуњен.

Коначно решење даје рекурзивни позив за $k = 0$, јер се њиме одређује број начина да се све групе распореде.

Рецимо и да би се алтернативно распоређивање могло вршити са краја низа тако што би се прво распоређивале последње групе које стоје у реду (оваква формулација би била практично дуална овој описаној, па ни једна ни друга немају неких значајних предности ни мана).

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

using namespace std;

const int MOD = 1e9 + 7;

int brojRasporeda(const vector<int>& a, int k) {
  if (k == a.size())
    return 1;
```

```
  int broj = 0;
  int uVagonu = 0;
  for (int i = k; i < a.size(); i++) {
    uVagonu += a[i];
    if (uVagonu > 4)
      break;
    broj = (broj + brojRasporeda(a, i+1)) % MOD;
  }
  return broj;
}

int brojRasporeda(const vector<int>& a) {
  return brojRasporeda(a, 0);
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << brojRasporeda(a) << endl;
  return 0;
}
```

Мемоизација

Рекурзивно решење је неефикасно, јер долази до преклапања рекурзивних позива. Ово се може поправити коришћењем мемоизације.

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

using namespace std;

const int MOD = 1e9 + 7;

int brojRasporeda(const vector<int>& a, int k, vector<int>& memo) {
  if (memo[k] != -1)
    return memo[k];

  if (k == a.size())
    return memo[k] = 1;

  int broj = 0;
  int uVagonu = 0;
  for (int i = k; i < a.size(); i++) {
    uVagonu += a[i];
    if (uVagonu > 4)
      break;
    broj = (broj + brojRasporeda(a, i+1, memo)) % MOD;
  }
  return memo[k] = broj;
}

int brojRasporeda(const vector<int>& a) {
  vector<int> memo(a.size() + 1, -1);
  return brojRasporeda(a, 0, memo);
}
```

```
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << brojRasporeda(a) << endl;
  return 0;
}
```

Динамичко програмирање навише

Уместо рекурзивне имплементације, могуће је направити и решење динамичким програмирањем навише где се у низу на позицији k налази број начина да се распореде групе од позиције k па до краја низа. Низ динамичког програмирања попуњавамо здесна налево.

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

using namespace std;

const int MOD = 1e9 + 7;

int brojRasporeda(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n + 1);
  dp[n] = 1;
  for (int k = n-1; k >= 0; k--) {
    dp[k] = 0;
    int uVagonu = 0;
    for (int i = k; i < n; i++) {
      uVagonu += a[i];
      if (uVagonu > 4)
        break;
      dp[k] = (dp[k] + dp[i+1]) % MOD;
    }
  }
  return dp[0];
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << brojRasporeda(a) << endl;
  return 0;
}
```

Меморијска оптимизација

Током одређивања вредности низа динамичког програмирања није потребно чувати целокупан низ, већ само неколико његових узастопних елемената. Потребно је дакле да имамо структуру података која може да чува сегменте тог низа и из које се могу уклањати елементи са краја и додавати на почетак, уз индексни приступ сваком елементу у текућем сегменту. Све ово нам пружа ред са два краја (`deque`).

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

```cpp
#include <deque>

using namespace std;

const int MOD = 1e9 + 7;

int brojRasporeda(const vector<int>& a) {
  int n = a.size();
  deque<int> dp;
  dp.push_back(1);
  for (int k = n-1; k >= 0; k--) {
    int broj = 0;
    int uVagonu = 0;
    int i = k;
    while (i < n) {
      uVagonu += a[i];
      if (uVagonu > 4)
        break;
      broj = (broj + dp[i-k]) % MOD;
      i++;
    }
    while (dp.size() >= i - k)
      dp.pop_back();
    dp.push_front(broj);
  }
  return dp[0];
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << brojRasporeda(a) << endl;
  return 0;
}
```

Глава 6

Фебруар 2021.

Задатак: Број сегмената малог збира

Дат је низ неозначених целих бројева a дужине n и један број t. Написати програм који одређује колико постоји сегмената датог низа чији је збир свих елемената строго мањи од t.

Опис улаза

Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$) и затим се учитава n бројева a_i ($0 \leq a_i \leq 1000$) који представљају елементе низа. На крају се учитава број t ($1 \leq t \leq 10^9$).

Опис излаза

На стандардни излаз исписати тражени број сегмената.

Пример

Улаз	*Излаз*

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

Објашњење

Сегменти са збиром елемената мањим од 3 су: `[2]`, `[2 0]`, `[0]`, `[2]` и `[1]`.

Решење

Задатак: Све путање у пуном стаблу на основу префиксног обиласка

Пуно бинарно стабло је бинарно стабло чији сви унутрашњи чворови имају тачно 2 детета. Дат је префиксни обилазак пуног бинарног стабла са n чворова где је у сваком чвору уписано по једно слово, при чему су листови означени великим словом. Написати програм који исписује све путеве од корена до листа тог стабла.

Опис улаза

Са стандардног улаза се учитава једна ниска састављена од малих и великих слова енглеског алфабета дужине n ($1 \leq n \leq 10^5$) која представља префиксни обилазак датог стабла.

Опис излаза

На стандардни излаз за сваки лист стабла у засебном реду исписати ниску која представља пут од корена стабла до тог листа.

Пример

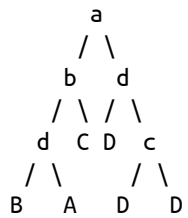
Улаз	*Излаз*

`abdBACdDcDD`	`abdB`
	`abdA`
	`abC`
	`adD`
	`adcD`
	`adcD`

Објашњење

Објашњење

Учитано стабло је приказано на слици



Решење

Конструкција бинарног дрвета

Један начин да се задатак реши је да се на основу префиксног обиласка прво експлицитно у меморији конструише бинарно дрво, а да се затим у том дрвету испишу све путање од корена до листа.

Дрво се може представити на уобичајени начин (помоћу структура које садрже карактер уписан у одговарајући чвор и два показивача ка поддрветима). Реконструкција пуног бинарног дрвета на основу префиксног обиласка се може извршити коришћењем једноставне рекурзивне функције. Она обилази ниску префиксног обиласка карактер по карактер и креира и враћа чвор дрвета који садржи текући карактер те ниске. Ако се ради о великом слову, у питању је лист и функција завршава са радом. Ако се ради о малом слову, у питању је унутрашњи чвор и функција се рекурзивно позива два пута: једном да прочита лево поддрво, а други пут да прочита десно поддрво.

Исписивање свих путања се може извршити поново помоћу једноставне рекурзивне функције. Она одржава текућу путању, приликом посете чвору дописује карактер уписан у тај чвор на крај те путање и ако се ради о великом слову тј. о листу исписује путању, а у супротном рекурзивно наставља да продужава путању на основу левог, а затим на основу десног поддрвета.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

struct cvor {
  char c;
  cvor *levo, *desno;
};

cvor* procitaj(const string& s, int& p) {
  cvor* novi = new cvor();
  novi->c = s[p++];
  if (isupper(novi->c)) {
    novi->levo = novi->desno = nullptr;
  } else {
    novi->levo = procitaj(s, p);
    novi->desno = procitaj(s, p);
  }
  return novi;
```

```
}

cvor* procitaj(const string& s) {
  int p = 0;
  return procitaj(s, p);
}

void ispisi_sve_putanje(cvor* c, string& s) {
  s.push_back(c->c);
  if (isupper(c->c))
    cout << s << endl;
  else {
    ispisi_sve_putanje(c->levo, s);
    ispisi_sve_putanje(c->desno, s);
  }
  s.pop_back();
}

void ispisi_sve_putanje(cvor* c) {
  string s;
  ispisi_sve_putanje(c, s);
}

void obrisi(cvor* c) {
  if (c != nullptr) {
    obrisi(c->levo);
    obrisi(c->desno);
    delete c;
  }
}


int main() {
  ios_base::sync_with_stdio(false);

  string p;
  cin >> p;

  cvor* c = procitaj(p);
  ispisi_sve_putanje(c);
  obrisi(c);

  return 0;
}
```

Коришћење стека

Током префиксног обиласка дрвета имплементираног нерекурзивно, уз помоћ стека, приликом сваке посете неком листу тренутни садржај стека представља путању од корена до тог листа. Пошто се сви листови обиђу (у одговарајућем редоследу, ако се прво обилази лево, па онда десно подстабло) исписивањем садржаја стека исписаће се све тражене путање.

Стога ћемо задатак решити одржавајући стек тако да његов садржај све време буде идентичан као садржај током префиксног обиласка. Анализираћемо дату ниску која садржи префиксни обилазак. Приликом сваке посете унутрашњем чвору (малом слову), ставићемо га на стек. У случају када наиђемо на велико слово, ставићемо га на стек и тада ћемо знати да се на стеку појавила наредна путања од корена до листа која се може исписати. У том тренутку претрага у дубину врши повратак. Пошто се са обрадом листа завршило, уклонићемо га са стека. Сада се јавља дилема да ли мало слово које је остало на врху стека треба уклањати или не.

- Ако смо приликом повратка први пут наишли на тај чвор, то значи да је у питању чвор чије смо само лево поддрво обрадили и да је потребно прећи на обраду његовог десног поддрвета. То значи да слово не треба скидати са стека, већ повратак треба прекинути и треба прећи на проширивање стека словима која се налазе у наставку ниске префиксног обиласка.
- Са друге стране, ако смо други пут приликом повратка наишли на тај чвор, то значи да смо наишли на чвор чија су оба поддрвета (и лево и десно) обрађена, па је у потпуности завршена обрада поддрвета којој је тај чвор корен. То значи да се слово скида са стека и наставља се фаза повратка (ако је стек празан, повратак је завршен, а ако није, онда се за мало слово које је сада на врху стека проверава да ли смо на њега наишли први или други пут приликом повратка).

Да бисмо знали да ли је неки чвор посећен први или други пут приликом повратка, на стек ћемо постављати парове где је први елемент карактер уписан у чвор дрвета, а други има вредност `false` ако чвор није до сада обрађен приликом повратка тј. `true` ако јесте. Тај податак има вредност `true` ако и само ако нема више преосталих поддрвета која треба обрадити. Приликом стављања малих слова на стек, стављаћемо их уз вредност `false`. Приликом стављања великих слова на стек, стављаћемо их уз вредност `true` (јер за лист нема поддрвета која треба обрадити). Повратак је сада могуће реализовати тако што се скидају чворови са стека, све док им је вредност друге компоненте `true` (или док се стек не испразни). Пре него што се настави обрада ниске, чвору који је преостао на врху стека након фазе повратка (ако такав постоји) постављамо вредност на `true` (јер управо прелазимо на обраду његовог последњег поддрвета, па ће у повратку он бити потпуно обрађен и биће потребно скинути га са стека).

Илуструјмо рад овог алгоритма на примеру ниске `abdBACdDcDD`. Прикажимо преостали део ниске, стање стека и путање које се исписују.

```
abdBACdDcDD          []
bdBACdDcDD           [(a, F)]
dBACdDcDD            [(a, F), (b, F)]
BACdDcDD             [(a, F), (b, F), (d, F)]
ACdDcDD              [(a, F), (b, F), (d, F), (B, T)]     abdB
                     [(a, F), (b, F), (d, T)]
CdDcDD               [(a, F), (b, F), (d, T), (A, T)]     abdA
                     [(a, F), (b, T)]
dDcDD                [(a, F), (b, T), (C, T)]             abC
                     [(a, T)]
DcDD                 [(a, T), (d, F)]
cDD                  [(a, T), (d, F), (D, T)]             adD
                     [(a, T), (d, T)]
DD                   [(a, T), (d, T), (c, F)]
D                    [(a, T), (d, T), (c, F), (D, T)]     adcD
                     [(a, T), (d, T), (c, T)]
                     [(a, T), (d, T), (c, T), (D, T)]     adcD
                     []
```

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

using namespace std;

void ispisi_putanju(const vector<pair<char, bool>>& stek) {
  for(auto x : stek)
    cout << x.first;
  cout << endl;
}

int main() {
  ios_base::sync_with_stdio(false);

  string p;
  cin >> p;
```

```
  vector<pair<char, bool>> stek;
  for (char ch : p)
    if (islower(ch))
      stek.push_back({ch, false});
    else {
      stek.push_back({ch, true});
      ispisi_putanju(stek);
      while (!stek.empty() && stek.back().second)
        stek.pop_back();
      if (!stek.empty())
        stek.back().second = true;
    }

  return 0;
}
```

Задатак: Обилазак низа скоковима

Дат је низ природних бројева a дужине n. Играч почиње на елементу са позицијом 0. У једном потезу са позиције i играч може да пређе на једну од позиција $i + a_i$ или $i - a_i$ уколико та позиција постоји. Написати програм који за задати низ исписује све могуће обиласке тог низа у којима играч свако поље посећује тачно једном.

Опис улаза

Са стандардног улаза се учитава број n ($1 \leq n \leq 100$). Затим се учитава n бројева a_i ($1 \leq a_i \leq n - 1$).

Опис излаза

На стандардни излаз исписати све обиласке низа у било ком редоследу. Сваки обилазак исписати у засебном реду као низ бројева раздвојених размаком који представљају у ком тренутку је играч стао на коју позицију.

Пример

Улаз	*Излаз*
5	0 2 1 3 4
2 2 1 1 3	0 4 1 2 3

Објашњење

У првом обиласку играч иде редом кроз позиције 0, 2, 1, 3 и 4. У другом обиласку играч иде редом кроз позиције 0, 2, 3, 4 и 1.

Решење

Задатак решавамо рекурзивним набрајањем тражених комбинаторних објеката. Дефинишемо рекурзивну функцију која прима оригнални низ дужина скокова, затим низ који садржи информацију о томе у ком кораку је неко поље посећено (-1 означава да још није посећено), затим тренутни редни број направљеног корака (бројање креће од 0) и позицију i на којој се тренутно налазимо.

На тренутну позицију i уписујемо информацију о томе да смо на њу стигли у кораку m. Ако је то последњи број корака (ако је $m = n - 1$) обишли смо цео низ и исписујемо га. У супротном покушавамо да направимо скок улево и ако је могуће, рекурзивно настављамо допуњавање са те позиције, па затим скок удесно и ако је могуће, рекурзивно настављамо допуњавање са те позиције.

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

using namespace std;

// vrsimo obilazak niza od pozicije i, pri cemu je m trenutni redni
// broj koraka, dok se u nizu v cuva informacija o tome u kom koraku
// je neko polje u nizu poseceno --- vrednost -1 oznacava da jos nije
```

```cpp
// poseceno
void search(int i, int m, vector<int>& v, const vector<int>& a) {
  // na poziciju i stavljamo element m cime oznacavamo da je pozicija i
  // posecena u koraku m
  v[i] = m;

  // niz je ceo popunjen, pa ga ispisujemo
  if(m == a.size() - 1) {
    for(int x : v)
      cout << x << ' ';
    cout << '\n';
  }

  // pokusavamo da napravimo a[i] koraka nalevo (to je moguce ako je indeks
  // i-a[i] unutar granica niza i ako je ta pozicija nije do sada posecena)
  if(i - a[i] >= 0 && v[i - a[i]] == -1)
    search(i - a[i], m + 1, v, a);

  // pokusavamo da napravimo a[i] koraka nadesno (to je moguce ako je indeks
  // i+a[i] unutar granica niza i ako je ta pozicija nije do sada posecena)
  if(i + a[i] < a.size() && v[i + a[i]] == -1)
    search(i + a[i], m + 1, v, a);

  // oznacavamo da pozicija i nije posecena
  v[i] = -1;
}

// obilazak niza a
void search(const vector<int>& a) {
  // broj elemenata niza
  int n = a.size();
  // nijedno polje do sada nije poseceno
  vector<int> v(n, -1);
  // u nultom koraku se nalazimo na polju 0
  search(0, 0, v, a);
}

int main() {
  int n;
  cin >> n;

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

  search(a);
  return 0;
}
```

Задатак: Партиционисање на збир непарних сабирака

Написати програм који за дати природни број n одређује на колико се начина он може записати као збир непарних природних бројева. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$).

Опис излаза

На стандардни излаз исписати решење по модулу 1000000007.

Пример

Улаз	*Излаз*
6	8

Објашњење

Могући збирови су:

```
1+1+1+1+1+1
1+1+1+3
1+1+3+1
1+3+1+1
1+5
3+1+1+1
3+3
5+1
```

Решење

Динамичко програмирање

Нека је $f(n)$ број начина да се број n изрази као збир непарних сабирака. Анализирањем случајева за први сабирак, видимо да он може бити $1, 3, \ldots, k$, где је k последњи непаран број мањи или једнак n. Тако добијамо везу $f(n) = f(n-1)+f(n-3)+\ldots+f(n-k)$. Излаз из ове рекурзије је случај $f(0) = 1$ (јер се нула разлаже на један начин, као збир празног скупа сабирака). Ово инсиприше решење динамичким програмирањем чија је сложеност квадратна.

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

using namespace std;

int main() {
  int n;
  cin >> n;

  vector<int> dp(n + 1);
  dp[0] = 1;
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= i; j += 2)
      dp[i] = (dp[i] + dp[i - j]) % 1000000007;

  cout << dp[n] << '\n';
  return 0;
}
```

Оптимизација

Можемо приметити да тражени број разлагања заправо представља елементе чувеног Фибоначијевог низа, па их можемо израчунати и у линеарној сложености.

```cpp
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;

  pair<int, int> dp = {0, 1};
```

```
  for(int i = 2; i <= n; i++)
    dp = {dp.second, (dp.first + dp.second) % 1000000007};

  cout << dp.second << '\n';
  return 0;
}
```

Глава 7

Јун 2022. - група А

Задатак: Компресија

Написати програм који компресује унети низ целих бројева дужине n тако да се одржи међусобан поредак тих вредности. Ранг елемента x је број елемената из тог низа који су строго мањи од x. Компресија се врши тако што се сваки елемент низа замени својим рангом. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). Затим се уноси n целих бројева из интервала $[-10^9, 10^9]$ који представљају елементе низа.

Опис излаза

На стандардни излаз исписати n бројева који представљају низ добијен компресијом унетог низа.

Пример

Улаз	*Излаз*
5 4 -20 8 4 6	1 0 4 1 3

Решење

Низ се може сортирати, а затим се позиција сваког елемента може ефикасно пронаћи бинарном претрагом.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

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

  vector<int> s = v;
  sort(s.begin(), s.end());

  for(int x : v)
    cout << (lower_bound(s.begin(), s.end(), x) - s.begin()) << ' ';
  cout << endl;
```

```
  return 0;
}
```

Задатак: Сажимање бекства из лавиринта

Робот Карел је побегао из лавиринта у ком је био заробљен. Лавиринт се састоји из више квадратних поља и свако поље је суседно са највише 4 друга поља (лево, десно, горе и доле). Познато је да између свака два поља у лавиринту постоји тачно један пут (низ различитих суседних поља) који их повезује.

Робот је тражећи пут прилично лутао (више пута се враћао на исто поље). Написати програм који за познате кораке које је робот правио одређује најкраћи редослед корака који воде од његове почетне позиције до излаза из лавиринта. Један корак подразумева прелазак са тренутног поља на једно од суседних.

Опис улаза

Са стандардног улаза се уноси ниска S која представља кораке које је робот правио. Сваки карактер ниске је једно слово које одговара једном кораку (`L` - лево, `R` - десно, `U` - горе, `D` - доле). Број корака није већи од 10^5.

Опис излаза

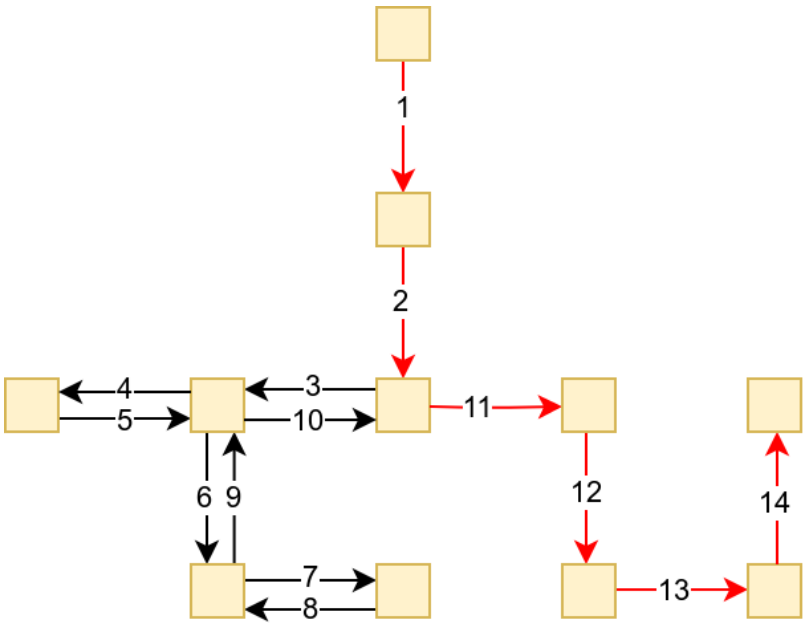
На стандардни излаз исписати најкраћи низ корака који води од почетне позиције робота до излаза из лавиринта.

Пример

Улаз	*Излаз*
`DDLLRDRLURRDRU`	`DDRDRU`

Објашњење

Путања којом је робот прошао је приказана на слици. Путања која се добије након сажимања тј. уклањања непотребних корака приказана је црвеним стрелицама.



Путања излаза из лавиринта

Решење

Непотребни кораци настају тако што робот крене у једном смеру и затим се врати у супротном. Јасно је да се такви кораци могу слободно избацити.

Прикажимо елиминисање сувишних корака на примеру ниске `DDLLRDRLURRDRU`.

- Сувшни су кораци 4 и 5 `LR`, па затим кораци 7 и 8 `RL`. Њиховим избацивањем се добија путања `DDL-DURRDRU`.

- Сада се и у овој путањи налазе сувшни кораци – то су кораци `DU` (они су били на местима 6 и 9 у оригиналној путањи). Њиховим елиминисањем добија се путања `DDLRRDRU`.
- Ова ниска такође садржи сувишне кораке `LR` (они су били на местима 3 и 10 у оригиналној путањи). Њиховим елминисањем добија се путања `DDRDRU`.
- Пошто добијена путања нема сувишних корака и она представља тражени резултат.

Задатак се зато своди на то да се итеративно проналазе узастопни парови слова који означавају супротне смерове и да се елиминишу из ниске, све док их има. Ово се може имплементирати ефикасније ако се употреби стек. Читамо ниску редом и карактере постављамо на стек. Пре постављања карактера проверавамо да ли се можда на врху стека налази њему супротан карактер и ако се налази, скидамо га (уместо да текући карактер поставимо на стек). Садржај стека на крају представља тражену сажету путању.

Пошто се на крају очекује испис путање, уместо да користимо `stack`, употребићемо обичан `string`, који методама `back`, `push_back` и `pop_back` пружа интерфејс стека.

```
#include <iostream>
#include <string>

using namespace std;

char suprotan_smer(char c) {
  switch(c) {
  case 'L': return 'R';
  case 'R': return 'L';
  case 'U': return 'D';
  case 'D': return 'U';
  }
  return 0;
}

int main() {
  string s;
  cin >> s;

  string stek;
  for (char c : s)
    if (!stek.empty() && stek.back() == suprotan_smer(c))
      stek.pop_back();
    else
      stek.push_back(c);

  cout << stek << endl;

  return 0;
}
```

Задатак: Све допуне заграда

Написати програм који за делимично попуњен низ заграда исписује све могуће начине да се тај низ допуни до коректно упареног низа заграда.

Опис улаза

Са стандардног улаза се уноси ниска која се састоји од карактера `(`, `)` и `.`, при чему `.` представља празно поље које треба попунити. Број празних поља није већи од 30.

Опис излаза

На стандардни излаз произвољним редоследом исписати све коректно упарене низове заграда који се добијају допуњавањем унете ниске, сваки у засебном реду.

Пример

Улаз	*Излаз*

`..(..)`	`(((())))`
	`()(())`
	`()()()`

Решење

Дефинисаћемо рекурзивну функцију која добија ниску која се попуњава заградама. Уз ниску се прослеђује и позиција i тако да је у сваком рекурзивном позиву првих i карактера већ попуњено заградама, а остали карактери су онакви какви су у оригиналној ниски. Уз то ћемо гарантовати да су заграде у том делу постављене исправно тј. ни у једном префиксу се не дешава да је број затворених заграда строго већи од броја отворених као и да је број отворених заграда које још нису затворене међу првих i карактера сигурно мањи или једнак од дужине непопуњеног дела ниске. Пошто нам је број отворених, а незатворених заграда на првих i позиција битан, и њега ћемо прослеђивати као параметар функције d.

Ако је i једнако дужини ниске, тада смо добили један исправан распоред заграда (затворене заграде нису претходиле отвореним, а пошто је преостало нула карактера, број отворених заграда које још нису затворене је мањи или једнак од нуле, па мора бити једнак нули).

У супротном анализирамо карактер на позицији i.

Ако је он једнак отвореној загради, тада настављамо попуњавање од наредне позиције, под условом да важи да је број преосталих позиција иза позиције i (то је $n - i$) већи или једнак броју отворених, а незатворених заграда закључно са позицијом i (ако је број отворених, а незатворених заграда пре позиције i био d, са заградом на позицији i он се пење на $d + 1$).

Ако је карактер на позицији i једнак затвореној загради, проверавамо да ли је број d отворених, а незатворених заграда у првих i карактера позитиван. Ако није, ова затворена заграда не би имала одговарајућу отворену која јој претходи, па се ниска не би могла допунити до исправне. Ако јесте, настављамо допуњавање од наредне позиције (умањујући d, број отворених, а незатворених заграда).

Ако је карактер на позицији i тачкица тада потенцијално испробавамо обе могућности за заграду на позицији i. При том водимо рачуна да отворену заграду стављамо само ако је у наставку остало довољно карактера да се на њима све досада отворене и незатворене заграде затворе, а да затворену заграду стављамо само ако међу првих i карактера има вишка отворених заграда.

```
#include <iostream>
#include <string>

using namespace std;

void dopuni_zagrade(string& s, int i, int d) {
  int n = s.length();
  if (i == n)
    cout << s << '\n';
  else {
    if (s[i] == '(') {
      if (n - i >= d + 1)
        dopuni_zagrade(s, i+1, d+1);
    } else if (s[i] == ')') {
      if (d > 0)
        dopuni_zagrade(s, i+1, d-1);
    } else if (s[i] == '.') {
      if (n - i >= d + 1) {
        s[i] = '(';
        dopuni_zagrade(s, i+1, d+1);
      }
      if (d > 0) {
        s[i] = ')';
        dopuni_zagrade(s, i+1, d-1);
      }
```

```
      s[i] = '.';
    }
  }
}

void dopuni_zagrade(string& s) {
  dopuni_zagrade(s, 0, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  string s;
  cin >> s;
  dopuni_zagrade(s);
  return 0;
}
```

Задатак: Плочице 2

Дат је низ целих бројева дужине n. На располагању су нам плочице које могу да покрију два суседна елемента. Написати програм који за задати низ одређује колики је највећи могући збир свих елемената покривених плочицама, ако је познато да сваки број може бити покривен највише једном плочицом. Временска и просторна сложеност треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). Затим се уноси n целих бројева који представљају елементе низа.

Опис излаза

На стандардни излаз исписати један цео број који представља тражени збир.

Пример

Улаз	*Излаз*
5 -1 2 -3 -2 4	3

Решење

Динамичко програмирање навише

Означимо са $f(a, i)$ максимални збир који се може добити помоћу првих $i + 1$ бројева низа, тј. покривањем бројева $a_0, \ldots, a_i$.

Ако је $i = 0$, тада постоји само један број, он се не може покрити плочицом и збир је $f(a, 0) = 0$.

Ако је $i = 1$, тада постоје два броја и или се оба могу покрити плочицом и тада је збир $a_0 + a_1$ или се плочица не ставља и тада је збир 0. Дакле, $f(a, 1) = \max(a_0 + a_1, 0)$.

Ако је $i \geq 2$, тада постоји више од два броја. Ако последњи није покривен плочицом, највећи збир се добија покривањем претходних бројева плочицама и износи $f(a, i - 1)$. Ако последњи јесте покривен, тада у збиру учествују последња два броја, а највећи збир се добија када се преостале плочице оптимално покрију, чиме се добија збир $f(a, i - 2) + a_{i-1} + ai$. Дакле, $f(a, i) = \max(f(a, i - 1), f(a, i - 2) + a_{i-1} + a_i$.

Смештањем вредности $f(a, i)$ у низ dp добијамо ефикасно решење.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
```

```
  int n;
  cin >> n;

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

  vector<int> dp(n);
  dp[0] = 0;
  dp[1] = max(a[0] + a[1], 0);
  for (int i = 2; i < n; i++)
    dp[i] = max(dp[i-1], dp[i-2] + a[i-1] + a[i]);
  cout << dp[n - 1] << endl;

  return 0;
}
```

Глава 8

Јун 2022. - група Б

Задатак: Збирови свих малих вредности

Дат је низ целих бројева дужине n и q упита. За сваки упит дат је један цео број x и потребно је одредити збир свих вредности из низа мањих или једнаких од x. Написати програм који обрађује упите.

Опис улаза

Са стандардног улаза се уноси број n ($n \leq 10^5$), након чега се уноси n бројева из интервала $[-10^9, 10^9]$ који представљају елементе низа. Затим се уноси број q ($q \leq 10^5$), након чега се уноси q бројева који представљају упите.

Напомена: Због великих вредности користити тип 64-битне целе бројеве.

Опис излаза

На стандардни излаз за сваки упит исписати по један број који представља одговор на упит.

Пример

Улаз	*Излаз*
5 9 2 -1 2 6 3 2 10 -5	3 18 0

Решење

Груба сила

Задатак се може решити грубом силом тако што се за сваки упит изнова саберу сви елементи низа који су мањи од дате границе.

Претрага и сабирање свих елемената низа мањих од дате границе се врши једним проласком кроз низ, у сложености $O(n)$. Пошто се то ради за сваки од q упита, укупна сложеност је $O(qn)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
```

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

  sort(v.begin(), v.end());

  int q;
  cin >> q;

  for (int i = 0; i < q; i++) {
    int x;
    cin >> x;

    long long zbir = 0;
    for(int j = 0; j < n && v[j] <= x; j++)
      zbir += v[j];

    cout << zbir << '\n';
  }

  return 0;
}
```

Бинарна претрага и префиксни збирови

Када се низ сортира, тада се све вредности мање од дате границе налазе на почетку низа. Бинарном претрагом је могуће ефикасно пронаћи скуп елемената које је потребно сабрати – довољно је пронаћи позицију првог елемента који је строго већи од дате границе и сабрати све елементе који се налазе испред те позиције. Да би се избегло сабирање префикса сваки пут из почетка, све збирове префикса је пожељно израчунати унапред и сместити у низ.

Збирови префикса се могу израчунати у времену $O(n)$. Након тога се q пута извршава бинарна претрага чија је сложеност $O(\log n)$ и читање из низа збирова префикса чије је сложеност $O(1)$. Укупна сложеност је стога $O(n + q \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;

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

  sort(v.begin(), v.end());

  vector<long long> p(n + 1, 0);
  partial_sum(v.begin(), v.end(), p.begin() + 1);

  int q;
  cin >> q;
```

```
  for (int i = 0; i < q; i++) {
    long long x;
    cin >> x;

    int pos = upper_bound(v.begin(), v.end(), x) - v.begin();
    cout << p[pos] << '\n';
  }

  return 0;
}
```

Глава 9

Јул 2021.

Задатак: Два блиска предајника

Постоји n локација на x-оси на које је могуће поставити предајник. На располагању су два предајника са дометом d. Потребно је поставити их тако да буду што више размакнути како би покривеност била што већа, али и да буду на раздаљини највише d како би могли међусобно да комуницирају. Написати програм који одређује максималну раздаљину између предајника. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($n \leq 10^5$) и затим се уноси n целих бројева из интервала $[-10^9, 10^9]$ који представљају координате тачака на x-оси где је могуће сместити предајнике. На крају се уноси број d ($d \leq 10^9$).

Опис излаза

На стандардни излаз исписати један број који представља тражену раздаљину.

Пример

Улаз	*Излаз*
4 7 3 1 8 3	2

Решење

Два показивача

Након што сортирамо позиције предајника, за сваку позицију десног предајника i одређујемо њему најдаљу могућу позицију левог предајника j, тако што j померамо удесно, све док је $p_j - p_i > d$. Када се i помери на следећу позицију, претрагу је довољно започети од текуће вредности j. Заиста, за све вредности $j' < j$ важи да је $p_{i+1} - p_{j'} \geq p_i - p_{j'} > d$.

Сложеношћу доминира сортирање у времену $O(n \log n)$. Пошто се након тога оба показивача крећу само удесно, сложеност друге фазе је $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
```

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

  int d;
  cin >> d;

  sort(p.begin(), p.end());

  int j = 0;
  int maxd = 0;
  for (int i = 0; i < n; i++) {
    while (j < i && p[i] - p[j] > d)
      j++;
    maxd = max(p[i] - p[j], maxd);
  }

  cout << maxd << endl;

  return 0;
}
```

Задатак: Сви распореди заграда дате дубине

Написати програм који исписује све низове коректно упарених заграда дужине n и дубине највише d.

Опис улаза

Са стандардног улаза се уносе вредности n ($1 \leq n \leq 20$), паран број) и d ($1 \leq d \leq 10$).

Опис излаза

На стандардни излаз исписати све тражене низове заграда у лексикографски растућем поретку, сваки у засебном реду.

Пример

Улаз	*Излаз*
6	(()())
2	(())()
	()(())
	()()()

Решење

Дефинишимо рекурзивну функцију којом набрајамо све распореде заграда. Она прима делимично попуњену ниску (ниску којој је попуњено првих i позиција) и допуњује на све могуће начине који дају исправне распореде заграда дате максималне дубине.

Ако је $i = n$, тада је цела ниска попуњена и исписујемо је.

Ако је $i < n$ тада одређујемо врсту заграде коју ћемо уписати на позицију i. Прво разматрамо отворену, а затим затворену заграду.

Отворену заграду можемо поставити само ако се њом не би нарушила максимална дубина (о чему можемо водити рачуна тако што функцији прослеђујемо тренутну дубину отворених заграда) и ако смо сигурни да иза текуће позиције има довољно празних позиција на које постављањем затворених заграда можемо на крају добити исправну ниску (потребно је да је тренутна дубина строго мања од дате дубине и да је број преосталих карактера након постављања заграде на позицију i (то је $n-i$) већи или једнак од увећане дубине (то је $d+1$).

Затворену заграду можемо поставити ако је тренутна дубина отворених заграда строго већа од нуле.

```
#include <iostream>
#include <string>
```

```cpp
using namespace std;

void generisi_zagrade(int d, int maxd, string& s, int i) {
  int n = s.size();

  if (i == n)
    cout << s << endl;
  else {
    if (d < maxd && n - i - 1 >= d) {
      s[i] = '(';
      generisi_zagrade(d+1, maxd, s, i+1);
    }
    if (d > 0) {
        s[i] = ')';
        generisi_zagrade(d-1, maxd, s, i+1);
    }
  }
}

void generisi_zagrade(int n, int d) {
  string s(n, ' ');
  generisi_zagrade(0, d, s, 0);
}

int main() {
  int n, d;
  cin >> n >> d;
  generisi_zagrade(n, d);
  return 0;
}
```

Глава 10

Септембар 2021.

Задатак: Симетрична разлика

Написати програм који одређује симетричну разлику два унета скупа. Временска сложеност алгоритма треба да буде $O((m+n)\log(m+n))$, а просторна $O(m+n)$ при чему m и n представљају број елемената сваког скупа.

Опис улаза

Са стандардног улаза се уноси број m, након чега се уноси m различитих целих бројева. Потом се уноси број n и n различитих целих бројева.

Опис излаза

На стандардни излаз исписати елементе добијеног скупа, уређене растуће.

Пример

Улаз	*Излаз*
4 5 2 -3 8 3 8 3 2	-3 3 5

Објашњење

Бројеви -3, 3 и 5 се једини јављају у тачно једном од унетих скупова.

Решење

Скупови

Најједноставнији начин за решавање овог задатка је коришћење библиотечких колекција за представљање скупова.

```
#include <iostream>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  set<int> A, B, C;

  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
```

```
    int x;
    cin >> x;
    A.insert(x);
  }

  int m;
  cin >> m;
  for(int i = 0; i < m; i++) {
    int x;
    cin >> x;
    B.insert(x);
  }

  for (int a : A)
    if (B.find(a) == B.end())
      C.insert(a);

  for (int b : B)
    if(A.find(b) == A.end())
      C.insert(b);

  for (int c : C)
    cout << c << ' ';
  cout << endl;

  return 0;
}
```

```
#include <iostream>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  set<int> A, B, C;

  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    A.insert(x);
  }

  int m;
  cin >> m;
  for(int i = 0; i < m; i++) {
    int x;
    cin >> x;
    B.insert(x);
  }

  for (int a : A)
    if (B.find(a) == B.end())
      C.insert(a);

  for (int b : B)
```

```
    if(A.find(b) == A.end())
      C.insert(b);

  for (int c : C)
    cout << c << ' ';
  cout << endl;

  return 0;
}
```

Сортирање и бинарна претрага

Пошто се скупови не мењају након учитавања, могуће је репрезентовати их и сортираним низовима, што омогућава да се провера припадности елементу врши бинарном претрагом.

Глава 11

Јануар 2022. - група А

Задатак: Магнет

Дат је низ од n гомила металних куглица. Могуће је поставити магнет испред било које гомиле и тада ће све куглице из првих k гомила са десне стране магнета бити привучене до њега. Потребно је поставити магнет тако да укупан пређени пут куглица буде што већи. Написати програм који одређује највећи могући укупан пређени пут. Временска и просторна сложеност треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 10^5$) и k ($1 \leq k \leq n$). Затим се уноси n бројева не већих од 10^5 који представљају величине гомила.

Опис излаза

На стандардни излаз исписати један број који представља решење задатка.

Напомена: Због великих вредности користити 64-битне типове података.

Пример

Улаз	*Излаз*
`5 3` `1 4 2 3 2`	`17`

Објашњење

Постављањем магнета испред прве гомиле куглице прелазе пут $1 \cdot 1 + 2 \cdot 4 + 3 \cdot 2 = 15$. Постављањем магнета између прве и друге гомиле куглице прелазе пут $1 \cdot 4 + 2 \cdot 2 + 3 \cdot 3 = 17$. Постављањем магнета између друге и треће гомиле куглице прелазе пут $1 \cdot 2 + 2 \cdot 3 + 3 \cdot 2 = 14$. Постављањем магнета десно од ове позиције само скраћује пређени пут, па није потребно разматрати га.

Решење

Инкременталност

Ефикасно решење задатка се може добити применом инкременталности. Наиме, директно можемо израчунати укупан пређени пут ако се магнет постави испред прве групе куглица, слева, а затим можемо разматрати све даље положаје магнета, док се не нађе испред последњих k група куглица. Даље померање магнета нема смисла, јер се број куглица које се померају, као и њихов пређени пут, смањују.

Пређени пут свих куглица за једну позицију магнета можемо употребити да бисмо одредили пређени пут свих куглица за наредну позицију магнета. Наиме, ако је магнет испред позиције i, тада је укупан пређени пут $1 \cdot a_i + 2 \cdot a_{i+1} + \ldots + k \cdot a_{i+k-1}$. Када се магнет помери тако да буде испред позиције $i+1$, тада је укупан пређени пут $1 \cdot a_{i+1} + 2 \cdot a_{i+2} + \ldots + k \cdot a_{i+k}$. Дакле, пут се скратио за $a_i + a_{i+1} + \ldots a_{i+k-1}$ и увећао се за $k \cdot a_{i+k-1}$. Зато током рада одржавамо и збир броја куглица на позицијама од i до $i + k - 1$.

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

```cpp
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int n, k;
  cin >> n >> k;

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

  // izracunavamo predjeni put i broj kuglica kada je magnet ispred
  // prve gomile
  long long predjenPut = 0;
  long long brojKuglica = 0;
  for(int i = 0; i < k; i++) {
    brojKuglica += v[i];
    predjenPut += (i + 1) * v[i];
  }

  long long maxPredjenPut = predjenPut;
  // pomeramo magnet jednu po jednu gomilu nadesno i inkrementalno
  // azuriramo podatke
  for(int i = k; i < n; i++) {
    predjenPut -= brojKuglica;
    predjenPut += k * v[i];
    maxPredjenPut = max(maxPredjenPut, predjenPut);
    brojKuglica -= v[i - k];
    brojKuglica += v[i];
  }

  cout << maxPredjenPut << endl;
  return 0;
}
```

Задатак: Најкраћи сегмент целих датог збира

Дат је низ целих бројева дужине n. Написати програм који одређује најкраћи сегмент низа чији је збир елемената једнак вредности z. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 10^5$) и z ($1 \leq z \leq 10^9$), након чега се уноси n целих бројева из интервала $[-1000, 1000]$ који представљају вредности низа.

Опис излаза

На стандардни излаз исписати вредности a и b такве да је $[a, b]$ тражени сегмент. Уколико постоји више таквих сегмената, исписати онај са најмањом вредношћу a. Ако нема таквих сегмената, исписати `nema`.

Пример

Улаз	*Излаз*
`6 2` `1 4 -3 5 4`	`2 3`

Објашњење

Сегмент $[0, 2]$ има збир $1 + 4 + (-3) = 2$. Сегмент $[2, 3]$ има збир $(-3) + 5 = 2$.

Решење

Груба сила

За сваку позицију i одређујемо први сегмент који почиње на тој позицији, а чији је збир једнак z. То радимо тако што анализирамо редом сегменте $[i, j]$, кренувши од $j = i$, па надаље, рачунамо њихов збир елемената и заустављамо се када тај збир први пут постане једнак z. Од свих тако одређених сегмената бирамо најкраћи, тј. ажурирамо глобално најкраћи сегмент, само ако се на некој наредној позицији i појави сегмент који је строго краћи неко најкраћи до тада нађен (ако се појави сегмент исте дужине, не ажурирамо минимум, дајући тако предност сегментима који раније почињу).

Пошто за сваку позицију потенцијално сабирамо све елементе до краја низа, укупна сложеност је $O(n^2)$.

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

using namespace std;

int main() {
  int n, z;
  cin >> n >> z;

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

  bool pronadjen = false;
  int minDuzina = n+1;
  int minPoc = 0;
  for (int i = 0; i < n; i++) {
    int zbir = 0;
    for (int j = i; j < n; j++) {
      zbir += a[j];
      if (zbir == z && j - i + 1 < minDuzina) {
        pronadjen = true;
        minDuzina = j - i + 1;
        minPoc = i;
        break;
      }
    }
  }

  if (pronadjen)
    cout << minPoc << ' ' << minPoc + minDuzina - 1 << '\n';
  else
    cout << ”nema” << endl;

  return 0;
}
```

Префиксни збирови и мапа

За сваку позицију j у низу одређујемо најкраћи сегмент чији је збир z а који се завршава на тој позицији. То можемо ефикасно урадити тако што применимо технику префиксних збирова. Наиме, збир елемената сегмента $[i, j]$ једнак је разлици префиксних збирова Z_{j+1} и Z_i тј. разлици између збира првих $j+1$ елемената низа, а то су елементи на позицијама $[0, j]$ и првих i елемената низа, а то су елементи на позицијама $[0, i)$. Да би важило $Z_{j+1} - Z_i = z$, потребно је да важи да је $Z_i = Z_{j+1} - z$. Дакле, потребно је да међу ранијим збировима префикса пронађемо онај који има збир $Z_{j+1} - z$ и ако има више таквих, да пронађемо онај који је последњи тј. најдужи. Да бисмо то ефикасно могли да урадимо, одржавамо асоцијативни низ (мапу тј. речник) која пресликава вредности префиксних збирова Z у максималне вредности i такве да је $Z_i = Z$.

Приликом обраде сваког новог елемента низа a_j, израчунавамо префиксни збир Z_{j+1} увећавајући претходни префиксни збир Z_j за вредност a_j (иницијализујемо вредност $Z_0 = 0$). Након тога коришћењем асоцијативног низа ефикасно проверавамо да ли постоји неко i тако да је $Z_i = Z_{j+1} - z$. Ако не постоји, тада се ниједан сегмент збира z не завршава елементом a_j. Ако постоји, тада је то уједно највеће i такво да је $Z_i = Z_{j+1} - z$, па је пронађен најкраћи сегмент који се завршава елементом a_j чији је збир z. Ако је он строго краћи од раније пронађеног таквог сегмента, тада ажурирамо најкраћи пронађени сегмент (ако није строго краћи, минимум се не ажурира, дајући тако предност сегментима који раније почињу). На крају памтимо да је последњи префикс који има збир Z_{j+1} баш онај који има $j+1$ елемент тј. завршава се елементом a_j.

Ако се користи неуређена мапа у коју се елементи убацују и која се претражује у константном времену, укупна сложеност је $O(n)$.

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  int n, z;
  cin >> n >> z;

  int zbirPrefiksa = 0;
  // za svaku vrednost zbira prefiksa pamtimo najveću dužinu prefiksa tog zbira
  // tj. najveću poziciju i takvu da zbir prvih i elemenata niza Z_i
  // ima vrednost te
  unordered_map<int, int> maksPrefiks;
  maksPrefiks[0] = 0;

  // inicijalizujemo minimalnu duzinu segmenta zbira z na +beskonacno
  // i registrujemo pocetnu poziciju najkraceg segmenta
  int minDuzina = n + 1, minPoc = -1;
  for (int j = 0; j < n; j++) {
    int x;
    cin >> x;
    zbirPrefiksa += x; // zbir Z_{j+1} = [a0, ..., aj]
    // trazimo najveće i takvo da je zbir [a0, ..., ai-1] jednak Z_{j+1} - z
    auto it =  maksPrefiks.find(zbirPrefiksa - z);
    if (it != maksPrefiks.end()) {
      int i = it->second;
      // pronasli smo najkraci segment [ai, ..., aj] čiji je zbir z
      int duzina = j - i + 1;
      // ako je on strogo kraći od prethodno najkraćeg, ažuriramo minimum
      if (duzina < minDuzina) {
        minDuzina = duzina;
        minPoc = i;
      }
    }
    // najduži segment koji ima zbir Z_{j+1} je [a0, ..., aj]
    maksPrefiks[zbirPrefiksa] = j + 1;
  }

  if (minPoc != -1)
    cout << minPoc << ' ' << minPoc + minDuzina - 1 << '\n';
  else
    cout << "nema" << endl;

  return 0;
}
```

Задатак: Боје у кругу

Аутор: Иван Дрецун

Перица и другари седе у кругу и смишљају коју ће игру следеће да играју. Договорили су се да ће се поделити у тачно три екипе (плаву, зелену и, наравно, црвену), али тако да деца која седе једна до другог морају бити у различитим екипама. Перицу и другаре занимају сви начини да направе поделу по екипама. Пошто још увек нису смислили коју ће игру следеће играти, замолили су тебе да им помогнеш и напишеш програм који ће исписати све поделе по екипама.

Опис улаза

Са стандардног улаза се уноси цео број n ($2 \leq n \leq 16$) који означава са колико другара се Перица игра.

Опис излаза

На стандардни излаз исписати онолико редова колико постоји могућих подела. Сваки ред треба да се састоји из тачно $n + 1$ карактера који редом означавају Перичину екипу и затим екипу сваког следећег друга удесно. Плаву екипу представити малим словом `p`, зелену малим словом `z` и црвену малим словом `c`. Редове исписати уређене по абецедном редоследу.

Пример

Улаз	*Излаз*
2	cpz
	czp
	pcz
	pzc
	zcp
	zpc

Решење

Приметимо да је потребно исписати сва бојења дужине $n + 1$ која испуњавају услове.

Задатак је могуће решити формирањем свих распореда боја и исписивањем само оних који испуњавају услове задатка (појављују се све три боје, сваке две суседне се разликују). Распореди се могу формирати рекурзивном претрагом - прво формирањем свих распореда бојења који почињу словом `c`, затим оних који почињу словом `p` и коначно оних који почињу словом `z`. Сложеност оваквог решења је $O(n3^n)$ и не осваја максималан број поена.

Брже решење је могуће постићи уколико формирамо само она бојења која немају две суседне једнаке боје. Можемо модификовати рекурзивну претрагу претходног решења тако да уместо генерисања бојења која почињу сваком бојом генеришемо само бојења која почињу бојом различитом од претходне. На пример, уколико је у претходном кораку претраге фиксирана боја `p`, довољно је проверити сва бојења којима је следећа боја `c` или `z`. Овим се сложеност смањује на $O(n2^n)$ што је довољно за максималан број поена.

```
#include <iostream>

using namespace std;

void generisiBojenja(int i, string& boje, bool sveBoje) {
  if (i == boje.size()) {
    if (sveBoje && boje[i - 1] != boje[0])
      cout << boje << '\n';
    return;
  }

  for(auto boja : {'c', 'p', 'z'})
    if(boje[i - 1] != boja) {
      boje[i] = boja;
      generisiBojenja(i + 1, boje,
                      sveBoje || (boja != boje[0] && boja != boje[1]));
    }
}
```

```
void generisiBojenja(int n) {
  string boje(n, ' ');
  for(auto boja : {'c', 'p', 'z'}) {
    boje[0] = boja;
    generisiBojenja(1, boje, false);
  }
}

int main() {
  int n;
  cin >> n;

  generisiBojenja(n + 1);

  return 0;
}
```

Задатак: Број неопадајућих поднизова

Написати програм који за дати низ дужине n одређује колико постоји његових поднизова (не нужно суседних елемената) који су уређени неопадајуће. Временска сложеност треба да буде $O(n^2)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 1000$). Након тога се уноси n целих бројева из интервала $[-10^9, 10^9]$ који представљају елементе низа.

Опис излаза

На стандардни излаз исписати тражени број. Пошто решење може бити веома велико, све операције вршити по модулу $10^9 + 7$.

Пример

Улаз	*Излаз*
5 1 2 -3 5 0	11

Објашњење

Неопадајући поднизови су 1, 2, -3, 5, 0, 12, 15, 25, -35, -30 и 125.

Решење

Динамичко програмирање

Кључна идеја је да за сваки елемент у низу одредимо број неопадајућих поднизова који се завршавају тим елементом. Пошто се сваки неопадајући низ завршава неким елементом, укупан број неопадајућих низова добијамо сабирањем броја неопадајућих низова за сваки завршни елемент.

Претпостављамо да у низу динамичког програмирања чувамо број неопадајућих низова за сваки завршни елемент који претходи тренутном. Тренутним елементом a_i се сигурно завршава неопадајући једночлан низ a_i, а могуће је и да елемент a_i продужава низ који се завршава неким елементом a_j, за $j < i$ и $a_j \leq a_i$. Зато анализирамо све такве j и a_j и увећавамо елемент у низу динамичког програмирања на позицији i за бројеве низова који се завршавају сваким тако пронађеним a_j.

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

using namespace std;

const int MOD = 1e9 + 7;
```

```cpp
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int ukupno = 0;
  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[j] <= a[i])
        dp[i] = (dp[i] + dp[j]) % MOD;
    ukupno = (ukupno + dp[i]) % MOD;
  }

  cout << ukupno << endl;
  return 0;
}
```

Глава 12

Јануар 2022. - група Б

Задатак: Сужавање интервала

У познатој игри *Двонедеља* налази се n играча представљених тачкама на x-оси. Простор за играње одређен је интервалом полупречника r чији је центар у координатном почетку (дакле, $[-r, r]$). Играч који се нађе изван тог интервала (на удаљености од координатног почетка строго већој од r) испада из игре. Сваког минута, полупречник се смањује за d и игра се завршава када полупречник постане строго мањи од 0. Написати програм који за сваког играча одређује колико минута ће бити у игри (могуће је да играч изгуби и на самом почетку игре). Временска и просторна сложеност треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 10^5$), r ($1 \leq r \leq 10^{18}$) и d ($1 \leq d \leq r$). Након тога се у n редова уноси по један број x који представља координату играча.

Напомена: Због великих вредности користити 64-битне типове података.

Опис излаза

На стандардни излаз исписати n бројева, по један за сваког играча, који представљају број минута које је играч провео у игри.

Пример

Улаз	*Излаз*
`4 8 3`	`1`
`7`	`2`
`-4`	`0`
`-9`	`3`
`2`	

Решење

За сваку позицију играча x лако можемо израчунати број минута колико ће играч на позицији x остати у игри. Ако је $|x| > r$ то је 0 минута. У супротном тражимо најмањи број k тако да после k минута игре играч на позицији x испадне. После k минута игре добија се интервал $[r-kd, r+kd]$, а играч испада ако је $|x| > r-kd$. Дакле, тражимо најмањи број k такав да је $k > \frac{r-|x|}{d}$. То је број $\left\lfloor \frac{r-|x|}{d} \right\rfloor + 1$.

```
#include <iostream>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  long long n, r, d, x;
  cin >> n >> r >> d;
```

```
  for (int i = 0; i < n; i++) {
    cin >> x;
    if (abs(x) > r)
      cout << 0 << '\n';
    else {
      long long k = (r - abs(x)) / d + 1;
      cout << k << '\n';
    }
  }

  return 0;
}
```

Задатак: Најдужа аритметичка прогресија

Дат је скуп величине n и цео број d. Написати програм који одређује величину највећег подскупа датог скупа таквог да његови елементи чине аритметичку прогресију са размаком d. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 10^5$) и d ($-10^5 \leq d \leq 10^5, d \neq 0$). Након тога се уноси n бројева из интервала $[-10^9, 10^9]$ који представљају елементе скупа.

Опис излаза

На стандардни излаз исписати један број који представља величину траженог подскупа.

Пример

Улаз	*Излаз*
`8 3` `5 8 4 1 9 11 2 7`	`4`

Објашњење

Највећи тражени подскуп је $2, 5, 8, 11$.

Решење

Динамичко програмирање

Низ можемо сортирати, а затим за сваки елемент редом одредити дужину најдуже аритметичке прогресије која се завршава тим елементом. Најдужа прогресија се мора завршити неким елементом, тако да њену дужину можемо одредити као максимум дужина за сваки завршни елемент. Дужина најдуже аритметичке прогресије за сваки завршни елемент x се може одредити као дужина најдуже аритметичке прогресије за елемент $x - d$. Ако се он не јавља у скупу (што ћемо знати, јер елементе обрађујемо у растућем поретку), x је први и уједно последњи елемент једночлане прогресије. За сваку вредност x у асоцијативном низу (мапи, речнику) ћемо памтити дужину најдуже аритметичке прогресије која се завршава елементом x.

Ако је d негативна вредност, можемо је заменити њој супротном (јер прогресија дужине n са разликом d постоји ако и само ако постоји прогресија дужине n са разликом $-d$ – она се добија простим обртањем редоследа елемената).

```
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n, d;
  cin >> n >> d;
```

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

  if (d < 0)
    d = -d;

  sort(begin(v), end(v));
  map<int, int> duzinaDo;
  int maxDuzina = 1;
  for (int i = 0; i < n; i++)
    if (duzinaDo.find(v[i] - d) != duzinaDo.end()) {
      duzinaDo[v[i]] = duzinaDo[v[i] - d] + 1;
      maxDuzina = max(maxDuzina, duzinaDo[v[i]]);
    } else
      duzinaDo[v[i]] = 1;

  cout << maxDuzina << endl;
  return 0;
}
```

Подела на класе еквиваленције

Елементи низа могу припадати аритметичкој прогресији чија је разлика d ако и само ако им је исти остатак при дељењу са d. Зато најдуже прогресије можемо тражити засебно унутар сваке класе еквиваленције по модулу d. Низ сортирамо на основу остатака при дељењу са d, а унутар сваке класе нумерички растуће. Након тога тражимо најдужу серију елемената такву да је сваки наредни тачно за d већи од претходног.

Пошто елементи полазног низа могу бити негативни, потребно је обратити пажњу на то да оператор остатка при дељењу може дати негативну вредност. Ако се то догоди, на вредност остатка треба додати d.

Сортирање се може извршити у времену $O(n \log n)$, након чега се најдужа серија узастопних елемената на растојању d проналази у сложености $O(n)$.

```
#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
using namespace std;

int mod(int x, int d) {
  int xd = x % d;
  return xd >= 0 ? xd : xd + d;
}

int main() {
  int n, d;
  cin >> n >> d;

  if (d < 0)
    d = -d;

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

  sort(begin(a), end(a),
       [d](int a, int b) {
         return mod(a, d) < mod(b, d) ||
```

```
                (mod(a, d) == mod(b, d) && a < b);
        });

  int maks = 1, tek = 1;
  for (int i = 1; i < n; i++) {
    if (a[i] == a[i-1] + d) {
      tek++;
      maks = max(maks, tek);
    } else
      tek = 1;
  }

  cout << maks << endl;

  return 0;
}
```

Задатак: Шпанска серија

Аутор: Иван Дрецун

Писци шпанске серије "Игра пресолца" одлучили су да у последњој епизоди направе велико окупљање на које ће бити позвани неки од n ликова из серије. Наравно, како то обично бива у шпанским серијама, постоји списак тројки ликова који се међусобно не подносе. Задатак за писце је да одаберу које од n ликова треба позвати тако да из сваке тројке са списка **тачно једна** особа буде позвана. Напиши програм који рачуна колико највише људи може бити позвано поштујући задате услове и исписује један такав списак званица.

Опис улаза

У првом реду стандардног улаза се уносе цели бројеви n ($3 \leq n \leq 14$) и k ($1 \leq k \leq 140$) који редом представљају број ликова у серији и број тројки ликова који се међусобно неподносе. У наредних k редова уносе се по три различита броја који означавају три лика од којих тачно један треба бити позван. Ликови су означени редним бројевима између 1 и n.

Опис излаза

У првом реду стандардног излаза исписати број позваних људи. У другом реду исписати редне бројеве позваних људи одвојене размаком. Уколико постоји више решења, исписати било које. Уколико не постоји списак званица који задовољава све услове, у првом реду стандардног излаза исписати 0.

Пример 1

Улаз	*Излаз*
6 2 1 2 3 1 4 5	3 2 4 6

Пример 2

Улаз	*Излаз*
4 4 1 2 3 1 2 4 1 3 4 2 3 4	0

Решење

Потребно је испробати све подскупове ликова и за сваки подскуп проверити да ли испуњава задати низ услова, односно да ли се из сваке тројке појављује тачно једна особа у подскупу. Приликом претраге подскупова потребно је памтити највећи до сада пронађен подскуп који испуњава услове - по завршетку испитивања свих подскупова тај ће бити решење.

Подскупове је могуће генерисати рекурзивним поступком: испитивањем прво свих подскупова који не садрже једног лика, а затим испитивањем свих подскупова који га садрже. Подскупове је могуће генерисати и употребом алгоритма за одређивање наредног подскупа.

Сложеност поскупка је $O(k2^n)$, што је у реду за дата ограничења променљивих.

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

```
using namespace std;

struct trojka { int a, b, c; };

int max_velicina = 0;
vector<bool> max_podskup;

bool ispunjavaUslove(vector<bool>& podskup, vector<trojka>& uslovi) {
  for(auto t : uslovi) {
    int broj_ljudi = 0;
    broj_ljudi += podskup[t.a - 1];
    broj_ljudi += podskup[t.b - 1];
    broj_ljudi += podskup[t.c - 1];
    if(broj_ljudi != 1)
      return false;
  }
  return true;
}

void sviPodskupovi(int i, vector<bool>& podskup, int velicina, vector<trojka>& uslovi) {
  if (i == podskup.size()) {
    if (velicina > max_velicina && ispunjavaUslove(podskup, uslovi)) {
      max_velicina = velicina;
      max_podskup = podskup;
    }
    return;
  }

  podskup[i] = false;
  sviPodskupovi(i + 1, podskup, velicina, uslovi);

  podskup[i] = true;
  sviPodskupovi(i + 1, podskup, velicina + 1, uslovi);
}

int main() {
  int n, k;
  cin >> n >> k;

  vector<trojka> trojke(k);
  for(int i = 0; i < k; i++)
    cin >> trojke[i].a >> trojke[i].b >> trojke[i].c;

  vector<bool> podskup(n);
  sviPodskupovi(0, podskup, 0, trojke);

  cout << max_velicina << '\n';
  if (max_velicina > 0)
    for (int i = 0; i < n; i++)
      if (max_podskup[i])
        cout << (i + 1) << ' ';
  cout << '\n';
  return 0;
}
```

Подскупове можемо представити и помоћу бинарног записа неозначених целих бројева.

```
#include <iostream>
```

```cpp
#include <vector>

using namespace std;

struct trojka { int a, b, c; };

int brojPozvanih(unsigned pozvani) {
  return __builtin_popcount(pozvani);
}

bool pozvan(unsigned pozvani, int o) {
  return pozvani & (1u << o);
}

int main() {
  int n, k;
  cin >> n >> k;

  vector<trojka> trojke(k);
  for(int i = 0; i < k; i++)
    cin >> trojke[i].a >> trojke[i].b >> trojke[i].c;

  int max_podskup = 0;
  int max_pozvanih = 0;
  for(int podskup = 0; podskup < (1 << n); podskup++) {
    if (brojPozvanih(podskup) <= max_pozvanih)
      continue;
    bool sat = true;
    for (auto t : trojke) {
      int count = 0;
      count += (int)pozvan(podskup, t.a - 1);
      count += (int)pozvan(podskup, t.b - 1);
      count += (int)pozvan(podskup, t.c - 1);
      if (count != 1) {
        sat = false;
        break;
      }
    }
    if (sat) {
      max_podskup = podskup;
      max_pozvanih = brojPozvanih(podskup);
    }
  }

  cout << max_pozvanih << '\n';
  if (max_pozvanih > 0)
    for (int o = 1; o <= n; o++)
      if (pozvan(max_podskup, o-1))
        cout << o << ' ';

  cout << '\n';
  return 0;
}
```

Задатак: Подниске несуседних елемената

Дате су ниске s и t. Написати програм који рачуна колико пута се ниска s појављује као подниска обавезно несуседних елемената ниске t. Временска и просторна сложеност треба да буду $O(n^2)$.

Опис улаза

Са стандардног улаза се уносе две ниске малих слова енглеске абецеде s и t ($1 \leq s \leq t \leq 1000$).

Опис излаза

На стандардни излаз исписати тражени број појављивања. Пошто решење може бити веома велико, све операције вршити по модулу $10^9 + 7$.

Пример

Улаз	*Излаз*
`abbc` `abbbbabcc`	`5`

Објашњење

Појављивања су `AbBbBabCc`, `AbBbBabcC`, `AbBbbaBcC`, `AbbBbaBcC`, `AbbbBaBcC`.

Решење

Рекурзивно решење

Празна ниска s се на један начин може одабрати као подниска ниске t (не одабере се ниједан карактер). Са друге стране, ако је ниска t празна, тада непразна ниска s не може бити њена подниска.

Ако су обе ниске непразне, тада разматрамо последње слово ниске t. Ако се одабере да оно није део подниске, тада се подниска тражи међу првих $|t| - 1$ карактера ниске t. Са друге стране, оно може бити део поднске само ако је једнако последњем слову ниске s. Ако јесте, онда се подниска која садржи све осим последњег карактера ниске t тражи међу првих $|t| - 2$ карактера ниске t (због услова несуседности, ако смо у подниску укључили последњи карактер ниске t, не можемо и претпоследњи). Треба обратити пажњу на случај када је t једночлана ниска, јер тада није могуће скратити је за два карактера. Једна могућност је да се допусти рекурзивни позив за ниску t дужине -1, а да се излаз из рекурзије не врши само када је дужина ниске t нула, већ када год је мања или једнака од нуле. Друга могућност која избегава негативне дужине ниски је да се у случају када је $|t| = 1$ и када су последња слова ниски s и t једнака, број начина увећа за 1 само ако је и $|s| = 1$.

```
#include <iostream>
#include <string>

using namespace std;

const int MOD = 1e9 + 7;

int nesusedne_podniske(const string& s, int ns, const string& t, int nt) {
  if (ns == 0)
    return 1;
  if (nt <= 0)
    return 0;
  int broj = nesusedne_podniske(s, ns, t, nt - 1) % MOD;
  if (s[ns-1] == t[nt-1])
    broj = (broj + nesusedne_podniske(s, ns - 1, t, nt - 2)) % MOD;
  return broj;
}

int main() {
  string s, t;
  cin >> s >> t;
  cout << nesusedne_podniske(s, s.size(), t, t.size()) << endl;
  return 0;
}
```

Динамичко програмирање навише

Пошто је рекурзивно решење неефикасно и доводи до преклапајућих рекурзивних позива, можемо формулисати решење динамичком програмирањем навише тако што у матрици памтимо вредности рекурзивних позива за све комбинације дужина ниски s и t.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

const int MOD = 1e9 + 7;

int main() {
  string s, t;
  cin >> s >> t;
  vector<vector<int>> dp(s.size() + 1, vector<int>(t.size() + 1));
  for (int j = 0; j <= t.size(); j++)
    dp[0][j] = 1;
  for (int i = 1; i <= s.size(); i++)
    dp[i][0] = 0;
  for (int i = 1; i <= s.size(); i++)
    for (int j = 1; j <= t.size(); j++) {
      dp[i][j] = dp[i][j-1];
      if (s[i-1] == t[j-1]) {
        if (j > 1)
          dp[i][j] = (dp[i][j] + dp[i-1][j-2]) % MOD;
        else if (i == 1)
          dp[i][j] = (dp[i][j] + 1) % MOD;
      }
    }

  cout << dp[s.size()][t.size()] << endl;
  return 0;
}
```

Глава 13

Фебруар 2022. - група А

Задатак: Рачуни

Аутор: Иван Дрецун

Потребно је симулирати банковни систем за k различитих корисника. Сваки корисник има рачун са почетним стањем 0. Потребно је подржати две врсте операција:

- `upit x` одређује колико постоји корисника чији рачун садржи тачно x динара
- `ime x` додаје x динара на рачун особе са именом `ime` (x може бити и негативно)

Написати програм који пордржава извршавање n оваквих операција.

Опис улаза

Са стандардног улаза се уносе бројеви n и k. Након тога се у n редова уноси по једна операција.

Опис излаза

За сваки упит (операцију првог типа) исписати одговор, сваки у засебном реду.

Пример

Улаз	*Излаз*

`6 4`
`marko 2`
`milan 5`
`dragana 4`
`upit 0`
`milan -1`
`upit 4`
	`1`
	`2`

Решење

Задатак се једноставно и лако може решити употребом две мапе тј. речника. Једна се користи да преслика име корисника у износ на његовом рачуну, а друга да преслика износ на рачуну у број рачуна који садрже тај износ.

```
#include <iostream>
#include <map>

using namespace std;

int main() {
  int n, m, x;
  cin >> n >> m;

  map<string, int> racun;
  map<int, int> brPojavljivanja;
```

```
  brPojavljivanja[0] = m;

  for (int i = 0; i < n; i++) {
    string s;
    cin >> s >> x;

    if (s == "upit")
      cout << brPojavljivanja[x] << '\n';
    else {
      brPojavljivanja[racun[s]]--;
      racun[s] += x;
      brPojavljivanja[racun[s]]++;
    }
  }

  return 0;
}
```

Задатак: К елемената најближих датој вредности у несортираном низу

Дат је низ од n бројева. Потребно је одредити k бројева у низу најближих броју x. Број a је ближи броју x од броја b ако $|a - x| \leq |b - x|$ или $|a - x| = |b - x|, a < b$. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви $n(1 \leq n \leq 10^6)$, $k(1 \leq k \leq n)$ и $x(1 \leq x \leq 10^9)$. Затим се уноси n бројева који су између 1 и 10^9.

Опис излаза

На стандардни излаз исписати тражених k бројева сортираних неопадајуће.

Пример

Улаз	*Излаз*
7 4 4 7 4 2 3 1 5 6	2 3 4 5

Решење

Покретни прозор

Задатак се брже може решити тако што се низ на почетку сортира. Након тога довољно је да тражене елементе пронађемо у једном проласку кроз низ (иако би се то бинарном претрагом могло учинити и брже). Посматрамо "покретни прозор" који садржи узастопних k елемената и померамо га од почетка низа надесно. У почетку се померањем покретног прозора надесно елементи приближавају траженој вредности x да би се непосредно након проналажења тражених k најближих вредности они почели удаљавати од x. Ако прозор почиње на позицији l, померањем надесно у прозор се додаје елемент a_{l+k} а избацује a_l. У првом тренутку када a_{l+k} није ближи вредности x од елемента a_l тј. у тренутку када је $|a_{l+k}| - x \geq |a_l - x|$ претрага престаје и најближих k елемената су елементи $a_l, \ldots, a_{l+k-1}$

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

int k_najblizih(vector<int>& v, int k, int x) {
  int n = v.size();
```

```
  sort(v.begin(), v.end());
  int l;
  for (l = 0; l + k < n; l++)
    if (abs(v[l+k] - x) >= abs(v[l] - x))
      break;
  return l;
}

int main() {
  ios_base::sync_with_stdio(false);

  int n, k, x;
  cin >> n >> k >> x;

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

  int l = k_najblizih(v, k, x);

  for(int i = l; i < l+k; i++)
    cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

Два показивача

Задатак можемо решити тако што након сортирања кренемо са два показивача са краја низа и приближавамо их ка средини, све док не обухвате сегмент од тачно k тражених елемената. У сваком кораку померамо онај показивач који указује на елемент који је даљи од вредности x (ако су оба подједнако удаљена, померамо десни показивач, јер по условима задатка предност дајемо левом елементу).

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<cmath>

using namespace std;

int k_najblizih(vector<int> &v, int k, int x) {
  int n = v.size();
  sort(v.begin(), v.end());
  int l = 0;
  int r = n-1;
  while (r-l >= k) {
    if (abs(v[l] -x) <= abs(v[r]-x))
      r--;
    else
      l++;
  }

  return l;
}


int main() {
  ios_base::sync_with_stdio(false);
```

```
  int n, k, x;
  cin >> n >> k >> x;

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

  int l = k_najblizih(v, k, x);

  for(int i = l; i < l+k; i++)
    cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

Задатак: Без претераног понављања исте цифре

Кажемо да је бинарни низ без претеривања уколико не садржи узастопно појављивање исте цифре дуже од k. Написати програм који исписује све такве низове дужине n у лексикографском поретку.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 20$) и k ($1 \leq k \leq n$).

Опис излаза

На стандардни излаз исписати све тражене бинарне низове, сваки у засебном реду. Низове исписивати без размака.

Пример

Улаз	*Излаз*
3 2	001
	010
	011
	100
	101
	110

Решење

Задатак се лако решава рекурзивно, као мала варијација алгоритма генерисања свих варијација. Рекурзивна функција прима низ коме је попуњено првих i елемената и покушава да га допуни постављајући нулу и затим јединицу на позицију i. Ако постављени елемент није једнак претходном, тада је започета нова серија једнаких цифара, а у супротном је серија продужена и проверавамо да ли је њена дужина мања или једнака k (ако није, неуспешно прекидамо текућу грану претраге).

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

using namespace std;

void ispisi(const vector<int>& v) {
  for (int x : v)
    cout << x;
  cout << '\n';
}

// dopunjava niz v od pozicije i, pri čemu br sadrži broj pojavljivanja
// iste cifre na kraju popunjenog dela niza
void generisi(vector<int>& v, int i, int br, int k) {
```

```
  if (i == v.size()) {
    ispisi(v);
    return;
  }

  for (int j = 0; j <= 1; j++) {
    v[i] = j;
    if (i == 0 || j != v[i-1])
      // započeli smo novu seriju cifara i njena dužina je 1
      generisi(v, i + 1, 1, k);
    else if (br + 1 <= k)
      // serija nije predugačka, pa možemo da je nastavimo
      generisi(v, i + 1, br + 1, k);
  }
}

void generisi(int n, int k) {
  vector<int> v(n);
  generisi(v, 0, 0, k);
}

int main() {
  ios_base::sync_with_stdio(0);

  int n, k;
  cin >> n >> k;
  generisi(n, k);

  return 0;
}
```

Задатак: Лагана шетња до врха планине

Матрицом је дата мапа планине. У сваком пољу матрице уписан је број који представља надморску висину тог дела планине. Планинар почиње из горњег левог поља и треба да стигне до врха који се налази у доњем десном пољу. Планинар са неког поља може да се помери на поље доле, десно или доле-десно, и то само ако је приликом преласка на то поље промена висине мања или једнака d. Написати програм који одређује на колико начина је могуће доћи до врха планине. Временска и просторна сложеност алгоритма треба да буду $O(n * m)$, где су n и m димензије матрице.

Опис улаза

Са стандардног улаза се уносе вредности n, m и d ($1 \leq n, m, d \leq 1000$). Затим се уноси матрица димензија $n \times m$.

Опис излаза

На стандардни излаз исписати тражени број различитих путева до врха. Како овај број може бити велик, резултат исписати по модулу $10^9 + 7$.

Пример

Улаз	*Излаз*
3 3 4 1 3 4 2 9 6 4 7 7	3

Решење

Динамичко програмирање

Задатак решавамо динамичким програмирањем. За свако поље планине, у засебној матрици памтимо број начина да се на планини стигне до тог поља. До почетног поља у горњем левом углу можемо стићи на тачно један начин. За свако од осталих поља анализирамо три могућа пута да до њега стигнемо: од горе, с лева и дијагонално, од горе-слева. За свако од та три претходна поља проверавамо да ли такво поље постоји и да ли је висинска разлика између њега и текућег поља највише d. Ако јесте, тада број начина да се стигне до текућег поља увећавамо за број начина да се стигне до тог претходног поља.

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

using namespace std;

const int MOD = 1e9 + 7;

int main() {
  ios_base::sync_with_stdio(false);
  int m, n, d;
  cin >> m >> n >> d;
  vector<vector<int>> planina(m, vector<int>(n));
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      cin >> planina[i][j];

  vector<vector<int>> dp(m, vector<int>(n, 0));
  dp[0][0] = 1;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
      if (i > 0 && abs(planina[i][j] - planina[i-1][j]) <= d)
          dp[i][j] = (dp[i][j] + dp[i-1][j]) % MOD;
      if (j > 0 && abs(planina[i][j] - planina[i][j-1]) <= d)
          dp[i][j] = (dp[i][j] + dp[i][j-1]) % MOD;
      if (i > 0 && j > 0 && abs(planina[i][j] - planina[i-1][j-1]) <= d)
          dp[i][j] = (dp[i][j] + dp[i-1][j-1]) % MOD;
    }

  cout << dp[m-1][n-1] << endl;

  return 0;
}
```

Глава 14

Фебруар 2022. - група Б

Задатак: Број парова датог збир упити

Нека је дат цео број t. Написати програм који подржава следеће операције:

- `pisi x` записује број x на таблу,
- `brisi x` брише једно појављивање броја x са табле,
- `upit` одређује колико постоји парова бројева записаних на табли таквих да је њихов збир једнак датом броју t.

Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$, где је n број упита.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 10^5$) и t ($-5 \cdot 10^8 \leq t \leq 5 \cdot 10^8$). Затим се уноси n редова који представљају извршене операције.

Опис излаза

За сваки упит исписати одговарајуће решење у засебном реду.

Пример

Улаз	*Излаз*

`10 6`
`pisi 4`
`pisi 2`
`upit`
`pisi 2`
`upit`
`pisi 3`
`pisi 3`
`upit`
`brisi 4`
`upit`
	`1`
	`2`
	`3`
	`1`

Решење

Груба сила

Решење грубом силом подразумева да елементе чувамо у вектору. Елементе додајемо на крај вектора, бришемо прво појављивање, а бројање парова вршимо угнежђеним петљама, анализирајући све парове.

Додавање се врши у сложености $O(1)$, брисање у сложености $O(n)$, а пребројавање у сложености $O(n^2)$, па је овај приступ веома неефикасан.

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

```cpp
#include <algorithm>

using namespace std;

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

  int n, t;
  cin >> n >> t;

  vector<int> tabla;
  for (int i = 0; i < n; i++) {
    string s;
    cin >> s;

    if (s == "pisi") {
      int x;
      cin >> x;
      tabla.push_back(x);
    } else if (s == "brisi") {
      int x;
      cin >> x;
      tabla.erase(find(begin(tabla), end(tabla), x));
    } else {
      int brojNacina = 0;
      for (int i = 0; i < tabla.size(); i++)
        for (int j = i+1; j < tabla.size(); j++)
          if (tabla[i] + tabla[j] == t)
            brojNacina++;
      cout << brojNacina << '\n';
    }
  }
  return 0;
}
```

Мултискупови

Брисање из низа је веома спора операција. Стога је потребно користити неку структуру података која омогућава брзо додавање и брисање, а то може бити уређени мултискуп. Он нам уједно омогућава и брзо одређивање броја појављивања било ког елемента, што омогућава да се брже преброји број парова. Број парова се одређује релативно наивно, тако што се за сваки елемент x мултискупа преброје појављивања елемената $t - x$. Притом је потребно водити рачуна о специјалном случају када је $x = t - x$.

Додавање и брисање из уређеног мултискупа су операције сложености $O(\log n)$, исто као и бројање појављивања елемената. Међутим, број парова се одређује релативно наивно, па је сложеност ове операције $O(n \log n)$.

```cpp
#include <iostream>
#include <set>

using namespace std;

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

  int n, t;
  cin >> n >> t;
```

```
  multiset<int> tabla;
  for (int i = 0; i < n; i++) {
    string s;
    cin >> s;

    if (s == "pisi") {
      int x;
      cin >> x;
      tabla.insert(x);
    } else if (s == "brisi") {
      int x;
      cin >> x;
      tabla.erase(tabla.find(x));
    } else {
      int brojNacina = 0;
      for (int x : tabla) {
        if (x != t - x)
          brojNacina += tabla.count(t - x);
        else
          brojNacina += tabla.count(t - x) - 1;
      }
      cout << brojNacina / 2 << '\n';
    }
  }
  return 0;
}
```

Мапе

Уместо да чувамо све елементе, довољно је да чувамо колико пута се сваки од њих појављује у низу, што се може урадити помоћу асоцијативног низа. Приликом додавања новог елемента број његовог појављивања се увећава за 1, а приликом брисања се умањује за 1. Пошто се збир t не мења током извршавања програма, број парова у тренутном низу можемо одржавати током рада програма и ажурирати га при свакој промени низа. Када се у низ учита нови елемент x број парова ћемо увећати за број појављивања елемента $t - x$, а када се из низа брише елемент x број парова ћемо умањити за број појављивања елемента $t - x$. Треба водити рачуна о томе да елемент може да чини пар сам са собом тј. да може важити да је $x + x = t$. Да би програм исправно радио и у овим случајевима, бројач парова ћемо увећати пре увећавања броја појављивања новог елемента x, а умањићемо га након умањивања броја појављивања уклоњеног елемента x.

Ако се користе уређене мапе, сложеност све три операције је $O(\log n)$.

```
#include <iostream>
#include <map>

using namespace std;

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

  int n, t;
  cin >> n >> t;

  map<int, int> brPojavljivanja;
  int brNacina = 0;
  for (int i = 0; i < n; i++) {
    string s;
    cin >> s;

    if (s == "pisi") {
```

```
      int x;
      cin >> x;
      brNacina += brPojavljivanja[t - x];
      brPojavljivanja[x]++;
    } else if (s == "brisi") {
      int x;
      cin >> x;
      brPojavljivanja[x]--;
      brNacina -= brPojavljivanja[t - x];
    } else {
      cout << brNacina << '\n';
    }
  }
  return 0;
}
```

Задатак: Најмања дужина k-точланих сегмената довољног збира

Дат је низ од n природних бројева и број t. Наћи најмањи број k (ако постоји) тако да за сваких k узастопних бројева у низу важи да им је збир већи или једнак t. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви $n(1 \leq n \leq 10^6)$ $t(1 \leq t \leq 10^{18})$, затим n бројева који су између 0 и 10^9.

Опис излаза

На стандардни излаз исписује се тражено k ако постоји, у супротном исписује се 0.

Пример

Улаз	*Излаз*
6 5 3 1 2 1 5 2	4

Решење

Бинарна претрага по решењу

За сваку дужину k прилично ефикасно можемо проверити да ли је збир свих k-точланих сегмената већи или једнак t. Кључно је да користимо инкременталност и да збир наредног k-точланог сегмента израчунавамо одузимајући почетни елемент претходног сегмента и додајући последњи елемент наредног.

Пошто проблем поседује својство монотоности (ако је збир свих k-точланих сегмената већи или једнак t, онда је и збир свих k'-точланих сегмената, за $k' > k$ већи или једнак t), оптималну вредност k можемо пронаћи бинарном претрагом.

Провера за сваку конкретну вредност k захтева време $O(n)$. Пошто k може да узме вредности из интервала $[1, n]$, врши се $O(\log n)$ провера, па је укупна сложеност $O(n \log n)$.

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

using namespace std;

bool proveri_zbirove(const vector<int>& a, int k, long long t) {
  long long zbir = 0;
  for (int i = 0; i < a.size(); i++) {
    zbir += a[i];
    if (i >= k - 1) {
      if (zbir < t)
        return false;
```

```
      zbir -= a[i - k + 1];
    }
  }
  return true;
}

int main() {
  ios_base::sync_with_stdio(false);

  int n;
  long long t;
  cin >> n >> t;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int l = 1, d = n;
  while (l <= d) {
    int s = l + (d - l) /2;
    if (proveri_zbirove(a, s, t))
      d = s - 1;
    else
      l = s + 1;
  }

  if (l == n + 1)
    cout << 0 << endl;
  else
    cout << l << endl;
  return 0;
}
```

Два показивача

Задатак може да се ефикасно реши техником два показивача. Одржавамо показиваче l и d и збир елемената који су на позицијама из сегмента $[l, d]$. Док је тај збир премали, увећавамо десни показивач (проширујемо сегмент), ажурирајући, ако је потребно, максималну дужину сегмента $[l, d]$, после сваког његовог проширивања. Када збир постане превелики, повећавамо леви показивач, скраћујући сегмент. Резултат је најдужи сегмент $[l, d]$ настао током овог поступка.

Сложеност овог поступка је очигледно $O(n)$.

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int n;
  long long t;
  cin >> n >> t;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  long long zbir = a[0];
  int l = 0, d = 0;
  int mink = 0;
```

```
  while (d < n) {
    if (zbir < t) {
      d++;
      zbir += a[d];
      mink = max(mink, d - l + 1);
    } else {
      zbir -= a[l];
      l++;
    }
  }

  if (mink == n+1)
    cout << 0 << endl;
  else
    cout << mink << endl;


  return 0;
}
```

Задатак: Балансирани бинарни низови

Бинарни низ је балансиран ако је разлика у броју појављивања цифара 0 и 1 највише k. Написати програм који исписује све овакве низове дужине n.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 20$) и k ($1 \leq k \leq n$).

Опис излаза

На стандардни излаз исписати тражене бинарне низове, сваки у засебном реду. Низове исписивати без размака између цифара.

Пример

Улаз	*Излаз*
3 1	001
	010
	011
	100
	101
	110

Решење

Задатак решавамо бектрекинг претрагом са одсецањем. Дефинишемо рекурзивну функцију чији је задатак да допуни тренутни низ нулама и јединицама од позиције i надаље. Функција прима и разлику између броја нула и јединица у тренутно попуњеном почетном делу низа. Та разлика може бити повећана или смањена за највише $n - i$ (тако што се стави $n - i$ нула тј. $n - i$ јединица). Ако би се и након такве поправке добио низ који има разлику ван интервала $[-k, k]$, тада се текући низ не би могао допунити до балансираног, па се тренутна грана претраге може исећи. У супротном проверавамо да ли је цео низ попуњен и ако јесте исписујемо га. Ако није, покушавамо да на позицију i упишемо прво нулу, а затим јединицу и рекурзивно настављамо допуњавање.

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

using namespace std;

void ispisi(const vector<int>& v) {
  for(int x : v)
    cout << x;
```

```
  cout << '\n';
}

void generisi(vector<int>& v, int i, int razlika, int k) {
  int n = v.size();

  if (razlika + n - i < -k || razlika - (n - i) > k)
    // ekvivalentno sa abs(razlika) > k + n - i
    return;

  if (i == n) {
    ispisi(v);
    return;
  }

  v[i] = 0;
  generisi(v, i + 1, razlika - 1, k);

  v[i] = 1;
  generisi(v, i + 1, razlika + 1, k);
}

void generisi(int n, int k) {
  vector<int> v(n);
  generisi(v, 0, 0, k);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n, k;
  cin >> n >> k;
  generisi(n, k);
  return 0;
}
```

Задатак: Максимални збир заједничког подниза два низа

Дата су два низа целих бројева. Написати програм који одређује заједнички подниз (не нужно суседних елемената) максималног збира елемената и исписује тај збир. Временска и просторна сложеност алгоритма треба да буду $O(nm)$, где су n и m дужине низова.

Опис улаза

Са стандардног улаза се уноси n ($1 \leq n \leq 1000$), након чега се уноси n бројева који представљају елементе првог низа. Затим се уноси број m ($1 \leq m \leq 1000$), након чега се уноси m бројева који представљају елементе другог низа.

Опис излаза

На стандардни излаз исписати један број који представља решење задатка.

Пример

Улаз	*Излаз*
4 1 -1 4 3 5 1 5 -1 2 3	4

Решење

Рекурзивна формулација

Ако је било који од два низа празан, тада низови немају ниједан заједнички подниз и решење је 0.

Ако се оба низа завршавају истим, позитивним елементом, тада тај елемент треба да буде укључен у подниз, што значи да проблем сводимо на проналажење максималног заједничког подниза префикса тих низова који се на крају проширује последњим елементом.

Ако се оба низа завршавају истим негативним елементом, тада тај елемент не треба да буде укључен у подниз, што значи да проблем сводимо само на проналажење максималног подниза заједничког подниза префикса тих низова.

Ако су елементи различити они не могу истовремено да буду део заједничког подниза, па разматрамо варијанте у којима је искључен један од њих и гледамо бољу од њих, што значи да се проблем своди на проналажење максималног заједничког подниза целог првог низа и другог низа без последњег елемента и на проналажење максималног заједничког подниза целог другог низа и првог низа без последњег елемента и узимање бољег од њих.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int maks_zbir(const vector<int>& a, int na,
              const vector<int>& b, int nb) {
  if (na == 0 || nb == 0)
    return 0;
  if (a[na-1] == b[nb-1])
    return maks_zbir(a, na-1, b, nb-1) + max(0, a[na-1]);
  else
    return max(maks_zbir(a, na-1, b, nb), maks_zbir(a, na, b, nb-1));
}

int main() {
  int na;
  cin >> na;
  vector<int> a(na);
  for (int i = 0; i < na; i++)
    cin >> a[i];

  int nb;
  cin >> nb;
  vector<int> b(nb);
  for (int i = 0; i < nb; i++)
    cin >> b[i];

  cout << maks_zbir(a, na, b, nb) << endl;
}
```

Динамичко програмирање

Због понављања рекурзивних позива, задатак је боље решити динамичким програмирањем. Резултате за префиксе низова можемо памтити у матрици.

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

using namespace std;

int main() {
```

```
  int na;
  cin >> na;
  vector<int> a(na);
  for (int i = 0; i < na; i++)
    cin >> a[i];

  int nb;
  cin >> nb;
  vector<int> b(nb);
  for (int i = 0; i < nb; i++)
    cin >> b[i];

  vector<vector<int>> dp(na + 1, vector<int>(nb + 1));
  for (int i = 0; i <= na; i++)
    dp[i][0] = 0;
  for (int j = 0; j <= nb; j++)
    dp[0][j] = 0;
  for (int i = 1; i <= na; i++)
    for (int j = 1; j <= nb; j++) {
      if (a[i-1] == b[j-1])
        dp[i][j] = dp[i-1][j-1] + max(a[i-1], 0);
      else
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  cout << dp[na][nb] << endl;

  return 0;
}
```

Пошто се свака врста матрице израчунава само на основу претходне, уместо матрице можемо памтити само један низ.

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

using namespace std;

int main() {
  int na;
  cin >> na;
  vector<int> a(na);
  for (int i = 0; i < na; i++)
    cin >> a[i];

  int nb;
  cin >> nb;
  vector<int> b(nb);
  for (int i = 0; i < nb; i++)
    cin >> b[i];

  vector<int> dp(nb + 1, 0);
  for (int i = 1; i <= na; i++) {
    int pp = dp[0];
    for (int j = 1; j <= nb; j++) {
      int tmp = dp[j];
      if (a[i-1] == b[j-1])
        dp[j] = pp + max(a[i-1], 0);
      else
        dp[j] = max(dp[j], dp[j-1]);
```

```
      pp = tmp;
    }
  }
  cout << dp[nb] << endl;

  return 0;
}
```

Глава 15

Јун 2022. - група А

Задатак: Круг

Дато је n природних бројева поређаних у круг. Потребно је пресећи круг на два места тако да је разлика у збировима елемената добијених делова што мања. Написати програм који одређује најмању тако добијену разлику. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$). Затим се уноси n бројева на кругу ($1 \leq a_i \leq 10^9$).

Напомена: Користити 64-битне типове података због могућих прекорачења.

Опис излаза

На стандардни излаз исписати тражену разлику.

Пример

Улаз	*Излаз*
5	1
1 2 3 2 5	

Објашњење

Круг је могуће поделити тако да се добију делови 2 3 2 и 5 1. Разлика у збировима је $(2+3+2)-(5+1) = 7-6 = 1$.

Решење

Инкрементално решење

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

using namespace std;

int main() {
  int n;
  cin >> n;

  int64_t zbirKruga = 0;
  vector<int> v(n);
  for(int i = 0; i < n; i++) {
    cin >> v[i];
    zbirKruga += v[i];
  }
```

```
  int j = 0; // [i, j]
  int64_t zbirSegmenta = v[0];
  int64_t minRazlika = zbirKruga;
  for (int i = 0; i < n; i++) {
    while (2 * zbirSegmenta < zbirKruga) {
      j++;
      if (j >= n)
        j -= n;
      zbirSegmenta += v[j];
    }
    cout << zbirSegmenta << " " << zbirKruga - zbirSegmenta << endl;
    minRazlika = min(minRazlika, 2 * zbirSegmenta - zbirKruga);
    zbirSegmenta -= v[i];
  }

  cout << minRazlika << '\n';

  return 0;
}
```

Задатак: Биоскоп

У биоскопу су седишта поређана у један низ. Сва седишта су окренута ка почетку низа, где се налази платно. Нека је видљивост у биоскопу дата бројем k. То значи да особа на седишту i неће видети платно ако на неком од седишта $i-k, i-k+1, \ldots, i-1$ седи особа која је строго виша од ње. Написати програм који за сваку особу одређује да ли она може да види платно или не. Временска сложеност алгоритма треба да буде $O(n \log k)$, а просторна $O(n+k)$.

Опис улаза

Са стандардног улаза се уносе вредности n и k ($1 \leq k \leq n \leq 2 \cdot 10^5$). Затим се уноси n бројева који представљају висине особа у биоскопу ($1 \leq m \leq 10^8$).

Опис излаза

На стандардни излаз исписати n бројева одвојених размаком. i-ти број означава да ли i-та особа може да види платно. Уколико не може да види платно исписати 1, у супротном исписати 0.

Пример

Улаз	*Излаз*
6 3 2 5 3 5 7 6	0 0 1 1 0 1

Решење

Мапа

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

int main() {
  int n, k;
  cin >> n >> k;

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

```
  multiset<int> s;
  for (int i = 0; i < n; i++) {
    if (!s.empty() && *(s.rbegin()) >= v[i])
      cout << "1 ";
    else
      cout << "0 ";

    s.insert(v[i]);
    if (i >= k)
      s.erase(s.find(v[i - k]));
  }
  cout << endl;

  return 0;
}
```

Задатак: Прорeђен низ

Низ је прорeђен уколико се свака два суседна броја разликују за барем d. Написати програм који за учитан низ различитих бројева исписује све његове прорeђене пермутације.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 20$) и d ($1 \leq d \leq 300$). Затим се уноси n бројева који представљају елементе низа ($1 \leq a_i \leq 2000$).

Опис излаза

На стандардни излаз исписати све тражене пермутације. Сваку пермутације исписати у засебном реду, а елементе сваке исписати одвојене размаком. Пермутације исписати у произвољном редоследу.

Пример

Улаз	*Излаз*
4 3	2 7 4 8
2 4 7 8	2 8 4 7
	4 7 2 8
	4 8 2 7
	7 2 8 4
	7 4 8 2
	8 2 7 4
	8 4 7 2

Решење

Рекурзивно генерисање пермутација

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void permutuj(vector<int>& v, int i, int d) {
  if (i == v.size()) {
    for(int x : v)
      cout << x << ' ';
    cout << endl;
    return;
  }

  for (int j = i; j < v.size(); j++) {
```

```
    if (i == 0 || abs(v[i - 1] - v[j]) >= d) {
      swap(v[i], v[j]);
      permutuj(v, i + 1, d);
      swap(v[i], v[j]);
    }
  }
}

int main() {
  ios_base::sync_with_stdio(0);

  int n, d;
  cin >> n >> d;

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

  permutuj(v, 0, d);
  return 0;
}
```

Задатак: Стабла

За дати број n израчунати број свих бинарних претраживачких стабала која имају n различитих чворова чије су вредности из скупа $1, 2, ..., n$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 1000$).

Напомена: Користити 64-битне типове података због могућих прекорачења.

Опис излаза

На стандардни излаз исписати тражени број по модулу $10^9 + 9$.

Пример

Улаз	*Излаз*
3	5

Решење

Мемоизација

```
#include<iostream>
#include<vector>
#include<cmath>

using namespace std;

long long MOD = 1e9 + 9;

long long num_of_trees(int n, vector<long long>& memo) {
  if (n == 0 || n == 1)
    return memo[n] = 1;

  if (memo[n] != 0)
    return memo[n];

  for (int i = 0; i < n; i++)
    memo[n] += num_of_trees(n-i-1, memo) * num_of_trees(i, memo) % MOD;
```

```
  return memo[n];
}

int main() {
  int n;
  cin >> n;

  vector<long long> memo(n+1, 0);
  cout << num_of_trees(n, memo) << endl;

  return 0;
}
```

Глава 16

Јул 2022.

Задатак: Производ

Дат је низ целих бројева a дужине n. Написати програм који рачуна збир производа $a_i a_j$ свих парова $i < j$ бројева датог низа. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Напомена: Због могућих прекорачења користити 64-битне типове података.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). Затим се уноси n бројева који представљају елементе низа ($-10^3 \leq a_i \leq 10^3$).

Опис излаза

На стандардни излаз исписати тражену вредност.

Пример

Улаз	*Излаз*
4 4 1 -2 3	3

Решење

Инкременталност

Потребно је израчунати следећи збир:

$$a_1a_0 + a_2a_0 + a_2a_1 + a_3a_0 + a_3a_1 + a_3a_2 + \ldots + a_{n-1}a_{n-2}$$

Груписањем сабирака добијамо:

$$a_1a_0 + a_2(a_0 + a_1) + a_3(a_0 + a_1 + a_2) + \ldots + a_{n-1}(a_0 + \ldots + a_{n-2})$$

Лако се примећује да збирове префикса којима се множи текући члан можемо израчунавати инкрементално, чиме се добија израчунавање линеарне сложености.

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  long long zbir = 0;
  long long zbirProizvoda = 0;
```

```
  for (int i = 0; i < n; i++) {
    long long x;
    cin >> x;
    zbirProizvoda += x * zbir;
    zbir += x;
  }
  cout << zbirProizvoda << endl;
}
```

Формула

Потребно је израчунати следећи збир:

$$S = (a_1a_0) + (a_2a_0 + a_2a_1) + (a_3a_0 + a_3a_1 + a_3a_2) + \ldots + (a_{n-1}a_0 + \ldots + a_{n-1}a_{n-2})$$

Овај бир можемо посматрати и као збир у ком се сваки елемент множи елементима лево и као збир у ком се сваки елемент множи елементима десно од њега тј. као збир:

$$S = (a_0a_1 + \ldots a_0a_{n-1}) + (a_1a_2 + \ldots + a_1a_{n-1}) + \ldots + (a_{n-2}a_{n-1})$$

Када се ови збирови саберу са збиром квадрата свих елемената

$$K = a_0a_0 + a_1a_1 + \ldots + a_{n-1}a_{n-1}$$

добија се збир у ком се јављају сви парови a_ia_j, за $0 \leq i, j < n$. Међутим, збир свих парова је очигледно једнак квадрату збира $Z = a_0 + a_1 + \ldots + a_{n-1}$. Пошто је, дакле, $2S + K = Z^2$, важи да је $S = Z^2 - K$. Збирове S и K се лако израчунавају у линеарној сложености, па се тако израчунава и тражени збир S.

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;

  int64_t zbir = 0;
  int64_t zbirKvadrata = 0;
  for(int i = 0; i < n; i++) {
    int64_t x;
    cin >> x;
    zbir += x;
    zbirKvadrata += x * x;
  }

  cout << (zbir * zbir - zbirKvadrata) / 2 << endl;
  return 0;
}
```

Задатак: Шпијуни

Како би открила шпијуне, агенција је сваком агенту (*string*) доделила *ID* агента (*string*) којег посматра. Додељен *ID* не мора да буде валидан (не мора да одговара ни једном агенту). *Шпијуни* не посматрају никога (додељен им је невалидан *ID*). *Шпијун* је откривен ако њега неко посматра. Одредити колико шпијуна је откривено.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). Затим се уноси n редова - парови "*аīенīī ID*" ($stringstring$), односно агент и *ID* (потенцијалног) агента којег посматра.

Опис излаза

На стандардни излаз исписати број откривених шпијуна.

Пример

Улаз	*Излаз*
7 agent9 agent1 agent6 agent7 agent7 agent8 agent5 agent6 agent8 agent4 agent0 agent2 agent2 agent4	2

Објашњење

Откривени шпијуни су 2 и 8. Они посматрају агента *agent4* (који не постоји), а њих редом посматрају *agent0* и *agent7*.

Решење

```cpp
#include <iostream>
#include <map>
#include <set>

int main() {
  map<string, string> posmatra;
  set<string> posmatran;

  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    string s1, s2;
    cin >> s1 >> s2;
    posmatra[s1] = s2;
    posmatran.insert(s2);
  }

  int otkrivenihSpijuna = 0;
  for (auto p : posmatra) {
    // agent p.first posmatra agenta p.second
    // ako p.second ne postoji tj. ne posmatra nikoga,
    // a neko posmatra agenta p.first onda je p.first otkriven spijun
    if (posmatra.find(p.second) == posmatra.end() &&
        posmatran.find(p.first) != posmatran.end())
      otkrivenihSpijuna++;
  }

  cout << otkrivenihSpijuna << endl;
  return 0;
}
```

Задатак: Скакач

Дата је табла димензије $n \times m$ и на сваком пољу је записан по један цео број. Скакач се на почетку налази у горњем левом углу табле. Скакач може да обиђе свако поље највише једном и може да се заустави у било

ком тренутку. Када обиђе неко поље број записан на њему се додаје на укупан збир. Написати програм који одређује највећи збир који је могуће остварити.

Опис улаза

Са стандардног улаза се уносе димензије табле n и m ($1 \leq n, m \leq 5$). Након тога се у наредних n редова уноси по m целих бројева који представљају бројеве записане на поља табле.

Опис излаза

На стандардни излаз исписати максимални тражени збир.

Пример

Улаз	*Излаз*
3 3 1 -4 3 4 0 2 0 -1 -2	7

Објашњење

Највећи збир је могуће остварити обиласком поља 1, -1, 3, 4, након чега се скакач заустављa.

Решење

Бектрекинг

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

using namespace std;

bool naTabli(int i, int j, const vector<vector<int>>& tabla) {
  return i >= 0 && j >= 0 && i < tabla.size() && j < tabla[0].size();
}

int najveciZbir(int i, int j, vector<vector<bool>>& posecen, const vector<vector<int>>& tabla) {
  int rezultat = 0;
  for(int k = 0; k < 8; k++) {
    int di[] = {1, 1, -1, -1, 2, 2, -2, -2};
    int dj[] = {2, -2, 2, -2, 1, -1, 1, -1};
    int ni = i + di[k];
    int nj = j + dj[k];
    if(naTabli(ni, nj, tabla) && !posecen[ni][nj]) {
      posecen[ni][nj] = true;
      rezultat = max(rezultat, najveciZbir(ni, nj, posecen, tabla));
      posecen[ni][nj] = false;
    }
  }
  return rezultat + tabla[i][j];
}

int najveciZbir(const vector<vector<int>>& tabla) {
  int n = tabla.size(), m = tabla[0].size();
  vector<vector<bool>> posecen(n, vector<bool>(m));
  posecen[0][0] = true;
  return najveciZbir(0, 0, posecen, tabla);
}

int main() {
  int n, m;
  cin >> n >> m;
```

```
  vector<vector<int>> tabla(n, vector<int>(m));
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      cin >> tabla[i][j];

  cout << najveciZbir(tabla) << endl;

  return 0;
}
```

Задатак: Коцкице

Бачено је n коцкица различитих боја. Свака коцкица има m страница означених бројевима од 1 до m. Написати програм који рачуна колико постоји различитих бацања таквих да је збир бројева на баченим коцкицама једнак s. Временска сложеност алгоритма треба да буде $O(nms)$, а просторна $O(ns)$.

Опис улаза

Са стандардног улаза се уносе природна броја n, m и s.

Опис излаза

На стандардни излаз исписати тражени број начина. Пошто тај број може бити јако велик потребно је вршити све операције по модулу $10^9 + 7$.

Пример

Улаз	*Излаз*
`2 6 10`	`3`

Објашњење

Бацају се две шестостране коцкице. Постоји три начина да се оствари збир $10 = 6 + 4 = 5 + 5 = 4 + 6$.

Решење

Динамичко програмирање навише

Нека је $f(n, s)$ број начина да се постигне збир s са n коцкица (од којих свака има M страна). Важи да је $f(0, 0) = 1$ и да је $f(0, s) = 0$, за $s > 0$ (без коцкица постоји 1 начин да се добије збир 0 и нема начина да се добије било који други збир. Ако се баца $n > 0$ коцкица, тада на првој коцкици може да буде било који број од 1 до m. Пошто се тражи збир s, разматрају се само она бацања за које је $m \leq s$. Свако такво бацање прве коцкице се комбинује са свим бацањима $n - 1$ коцкице која дају збир $s - m$, тј.

$$f(n, s) = \sum_{1 \leq m \wedge m \leq M \wedge m \leq s} f(n - 1, s - m)$$

На основу ове рекурентне везе можемо једноставно попунити матрицу која чува вредности функције f.

```
n
  0 1 2 3 4 5 6 7 8 9 10  s
0 1 0 0 0 0 0 0 0 0 0 0
1 0 1 1 1 1 1 1 0 0 0 0
2 0 0 1 2 3 4 5 6 5 4 3
```

Приметимо да свака врста зависи само од претходне, па нема потребе чувати истовремено целу матрицу, као и да би се сабирање могло убрзати ако би се уместо низа вредности чувале префиксне суме.

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

using namespace std;
```

```
const int MOD = 1000000000 + 7;

int main() {
  int N, M, S;
  cin >> N >> M >> S;

  vector<vector<int>> dp(N+1, vector<int>(S+1, 0));
  dp[0][0] = 1;

  for (int n = 1; n <= N; n++)
    for (int s = 0; s <= S; s++)
      for (int m = 1; m <= M; m++)
        if (m <= s)
          dp[n][s] = (dp[n][s] + dp[n-1][s-m]) % MOD;

  cout << dp[N][S] << endl;
}
```

Глава 17

Септембар 2022.

Задатак: Далековод

Дуж једне улице налази се n кућа и за сваку је позната удаљеност x_i од почетка улице. Потребно је поставити далековод негде дуж улице. Када се далековод постави, развлачи се по један кабл од далековода до сваке куће. Написати програм који одређује колика је минимална укупна дужина кабла потребна да би било могуће повезати далековод са свим кућама. Временска сложеност алгоритма треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$) и затим n различитих бројева x_i ($-10^9 \leq x_i \leq 10^9$).

Напомена: Због могућих прекорачења користите 64-битне типове података.

Опис излаза

На стандардни излаз исписати један број који представља тражено решење.

Пример

Улаз	*Излаз*
4 1 -2 3 -10	16

Решење

Инкременталност

Ефикасно решење се може добити тако што се дужина кабла израчунава инкрементално, тако што се разматра ефекат померања са једне на наредну локацију.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

  vector<int64_t> v(n);
  for(int i = 0; i < n; i++)
    cin >> v[i];

  sort(begin(v), end(v));
  int64_t duzina = 0;
```

```
  for (int i = 0; i < n; i++)
    duzina += v[i] - v[0];

  int64_t min_duzina = duzina;
  for (int i = 1; i < n; i++) {
    duzina -= (n - i) * (v[i] - v[i-1]);
    duzina += i * (v[i] - v[i-1]);
    min_duzina = min(min_duzina, duzina);
  }

  cout << min_duzina << endl;

  return 0;
}
```

Медијана

Може се доказати да се оптимално решење добија ако се далековод стави на централну локацију (или на било коју од две централне, ако је укупан број локација паран).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

  vector<int64_t> v(n);
  for(int i = 0; i < n; i++)
    cin >> v[i];

  sort(begin(v), end(v));

  int64_t duzina = 0;
  for(int i = 0; i < n; i++)
    duzina += abs(v[n / 2] - v[i]);

  cout << duzina << endl;

  return 0;
}
```

Задатак: Околина

Дат је скуп n целих бројева. Кажемо да је елемент a скупа у околини елемента b скупа ако је $|b - a| \leq d$. Написати програм који одређује који елемент скупа има највећу околину и колико она садржи елемената. Временска сложеност алгоритма треба да буде $O(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 2 \cdot 10^5$) и d ($1 \leq n \leq 2 \cdot 10^8$), а затим n различитих бројева x_i ($-10^9 \leq x_i \leq 10^9$) који представљају елементе скупа.

Опис излаза

На стандардни излаз исписати две вредности одвојене размаком. Прва вредност представља елемент скупа који има највећу околину (ако има више таквих, исписати најмањи), а друга вредност представља величину

те околине.

Пример

Улаз	*Излаз*
`7 4` `4 -6 3 12 1 -4 9`	`1 3`

Решење

Опис главног решења

У овом блоку се описује главно решење задатка.

```
#include <iostream>

using namespace std;
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n, d;
  cin >> n >> d;

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

  sort(begin(v), end(v));

  int l = 0, r = 0, sol = 0;
  int solv = v[0];
  for (int i = 0; i < n; i++) {
    while (v[i] - v[l] > d)
      l++;
    while (r < n && v[r] - v[i] <= d)
      r++;
    if (r - l > sol) {
      sol = r - l;
      solv = v[i];
    }
  }

  cout << solv << ' ' << sol << endl;

  return 0;
}
```

Задатак: Комбинације збира

Дат је број s и n сортираних различитих целих бројева. Исписати све низове састављене од датих n бројева таквих да је сума елемената низа једнака s.

Бројеви у низу се могу понављати и не мора бити укључено свих n бројева. Исписивати само различите низове, растуће сортиране (два низа су иста ако имају исте цифре које се понављају исти број пута нпр. `1 1 2` и `1 2 1`, иначе су различити).

Опис улаза

Са стандардног улаза учитава се број n, затим n различитих бројева, затим број s.

Опис излаза

На стандардни излаз исписати низове (линију по линију).

Улаз

```
3
2 3 5
13
```

Излаз

```
2 2 2 2 2 3
2 2 2 2 5
2 2 3 3 3
2 3 3 5
3 5 5
```

Решење

Бектрекинг

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

using namespace std;

void ispisi(const vector<int>& v) {
  for (int x : v)
    cout << x << " ";
  cout << endl;
}

void kombinacijaZbira(const vector<int>& v, int indeks, vector<int>& komb, int zbir) {
  if (zbir < 0)
    return;

  if (zbir == 0) {
    ispisi(komb);
    return;
  }

  for (int i = indeks; i < v.size(); i++) {
    komb.push_back(v[i]);
    kombinacijaZbira(v, i, komb, zbir - v[i]);
    komb.pop_back();
  }
}

void kombinacijaZbira(const vector<int>& v, int zbir) {
  vector<int> komb;
  kombinacijaZbira(v, 0, komb, zbir);
}


int main() {
  int n;
  cin >> n;
  vector<int> v(n);
  for (int i = 0; i < n; i++)
```

```
    cin >> v[i];
  int zbir;
  cin >> zbir;

  kombinacijaZbira(v, zbir);
  return 0;
}
```

Задатак: Битка

У једној игрици су за битку на располагању три могућа напада. Сваки напад захтева одређен број секунди t_i, по чијем истеку прави штету у износу d_i (противнику се одузима d_i поена). Потребно је одредити максималну укупну количине штете коју је могуће направити ако је познато да битка траје тачно t секунди. Временска и просторна сложеност алгоритма треба да буду $O(t)$.

Опис улаза

Са стандардног улаза се уносе бројеви $t1, t2, t3$ ($1 \leq t_i \leq t$) који представљају дужину сваког напада. Након тога се уносе бројеви $d1, d2, d3$ ($1 \leq d_i \leq 10000$) који представљају количину нанете штете сваког напада. Коначно, учитава се број t ($1 \leq t \leq 200000$) који представља дужину трајања битке.

Опис излаза

На стандардни излаз исписати један цео број који представља тражено решење.

Пример

Улаз	*Излаз*
3 4 5 5 8 11 17	35

Решење

Динамичко програмирање навише

Нека $f(t)$ представља највећи добитак ако игра траје тачно t секунди. Ако ниједан од напада не може да се изведе за тих t секунди, тада је $f(t) = 0$. Ако се на почетку изведе напад i, тада се добија награда $n_i + f(t - t_i)$. Вредност $f(t)$ се онда одређује тако што се одреди максимална вредност $n_i + f(t - t_i)$, за све нападе који могу да се изведу тј. за све нападе за које је $t_i \leq t$.

$$f(t) = max\{n_i + f(t - t_i) \mid 1 \leq i \leq 3 \wedge t_i \leq t\},$$

при чему је по договору $\max\{\} = 0$. Ова рекурентна веза омогућава решење, међутим, ако се имплементира рекурзивном функцијом, тада долази до поновљених рекурзиних позива и добија се неефикасно решење. Зато се задатак решава динамичким програмирањем навише, тако што се формира низ DP_t који чува вредности функције $f(t)$. Низ се попуњава док се не израчуна $DP[T]$, тј. $f(T)$, где је T учитано трајање игре.

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

using namespace std;

int main() {
  int t1, t2, t3;
  int d1, d2, d3;
  int t;

  cin >> t1 >> t2 >> t3;
  cin >> d1 >> d2 >> d3;
  cin >> t;
```

```
  vector<int> dp(t + 1);
  for (int i = 0; i <= t; i++) {
    dp[i] = 0;
    if (i >= t1)
      dp[i] = max(dp[i], dp[i-t1] + d1);
    if (i >= t2)
      dp[i] = max(dp[i], dp[i-t2] + d2);
    if (i >= t3)
      dp[i] = max(dp[i], dp[i-t3] + d3);
  }
  cout << dp[t] << endl;

  return 0;
}
```

Глава 18

Октобар 2022.

Задатак: Најчешћи карактер сваког подсегмента

Дата је ниска дужине n сачињена од карактера `a`, `b` и `c`, као и цео број k. Написати програм који за сваки подсегмент дужине k ниске одређује који се карактер појављује највећи број пута. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уноси број k ($1 \leq k \leq n$). Након тога се уноси ниска ($1 \leq n \leq 100000$).

Опис излаза

На стандардни излаз исписати $n-k+1$ карактера. i-ти по реду карактер представља карактер који се најчешће појављује у сегменту $[i, i+k)$ ниске. Уколико не постоји карактер који се појављује строго више од осталих, исписати `#`.

Пример

Улаз	*Излаз*
`3` `abbcaa`	`bb#a`

Објашњење

Сегменти дужине 3 су редом `abb` (најчешћи карактер `b`), `bbc` (најчешћи карактер `b`), `bca` (нема најчешћег карактера), `caa` (најчешћи карактер `a`).

Решење

Мапа

```
#include <iostream>
#include <map>

using namespace std;

int main() {
  int k;
  cin >> k;
  string s;
  cin >> s;

  map<char, int> m;
  for (int i = 0; i < k - 1; i++)
    m[s[i]]++;

  for (int i = k - 1; i < s.size(); i++) {
```

```
    m[s[i]]++;

    if (m['a'] > m['b'] && m['a'] > m['c'])
      cout << "a";
    else if (m['b'] > m['a'] && m['b'] > m['c'])
      cout << "b";
    else if (m['c'] > m['a'] && m['c'] > m['b'])
      cout << "c";
    else
      cout << "#";

    m[s[i - (k - 1)]]--;
  }

  cout << endl;
  return 0;
}
```

Низ бројача

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int k;
  cin >> k;
  string s;
  cin >> s;
  int br[3] = {0};
  for (int i = 0; i < s.size(); i++) {
    br[s[i] - 'a']++;
    if (i >= k-1) {
      if (br[0] > br[1] && br[0] > br[2])
        cout << 'a';
      else if (br[1] > br[0] && br[1] > br[2])
        cout << 'b';
      else if (br[2] > br[0] && br[2] > br[1])
        cout << 'c';
      else
        cout << '#';
      br[s[i-k+1] - 'a']--;
    }
  }
  cout << endl;

  return 0;
}
```

Задатак: Блиски

Бројеви a и b су блиски ако $|a - b| < d$. Дата су два низа бројева дужине n и број d. Одредити колико елемената првог низа има близак број у другом низу. Временска сложеност алгоритма треба да буде $(n \log n)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви n ($1 \leq n \leq 1000000$) и d, затим елементи првог низа, а онда елементи

другог низа.

Опис излаза

На стандардни излаз исписати тражени број.

Пример

Улаз	*Излаз*
6 2 5 1 7 -5 10 13 12 0 3 -2 7 9	4

Решење

Два показивача

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  int d;
  cin >> d;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    cin >> b[i];
  sort(begin(a), end(a));
  sort(begin(b), end(b));
  int brojBliskih = 0;
  int j = 0;
  for (int i = 0; i < n; i++) {
    while (j < n && b[j] <= a[i] - d)
      j++;
    if (j < n && b[j] < a[i] + d)
      brojBliskih++;
  }
  cout << brojBliskih << endl;
  return 0;
}
```

Бинарна претрага

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int bliski(int a, int b, int d) {
  return abs(a-b) < d;
}

int ima_bliskih(int x, const vector<int>& v, int d) {
  auto levo = upper_bound(begin(v), end(v), x - d);
```

```
  auto desno = lower_bound(begin(v), end(v), x + d);
  return levo < desno;
}

int main() {
  int n;
  cin >> n;
  int d;
  cin >> d;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    cin >> b[i];
  sort(begin(a), end(a));
  sort(begin(b), end(b));

  int broj_bliskih = 0;
  for(int x : a)
    if(ima_bliskih(x, b, d))
      broj_bliskih++;

  cout << broj_bliskih << endl;

  return 0;
}
```

Задатак: Збирови

Дата је матрица бројева између 1 и k, при чему су нека поља празна. За сваку колону и врсту матрице познат је тражени збир елемената. Написати програм који рачуна колико постоји различитих начина да се празна поља попуне тако да збир елемената у свакој врсти и колони буде једнак траженом збиру.

Опис улаза

Са стандардног улаза се уносе димензије матрице n и m ($1 \leq n, m \leq 10$), након чега се уноси број k ($2 \leq k \leq 4$). У наредном реду се уноси n бројева који представљају тражене збирове у свакој врсти. У реду након тог се уноси m броејва који представљају тражене збирове у свакој колони. Коначно, уноси се матрица бројева, при чему поља са вредношћу 0 представљају празно поље.

Опис излаза

На стандардни излаз исписати један број који представља колико постоји могућих решења.

Пример

Улаз	*Излаз*
3 3 3 6 6 7 6 5 8 1 2 3 0 1 0 0 2 0	2

Објашњење

Два могућа решења су:

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

и

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

Решење

Бектрекинг

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

using namespace std;

bool check(vector<vector<int>>& mat, vector<int>& row, vector<int>& col, int i, int j) {
  int sum = 0, zero = 0;
  for (int k = 0; k < row.size(); k++) {
    sum += mat[k][j];
    if (mat[k][j] == 0)
      zero++;
  }
  if (zero == 0 && sum != col[j])
    return false;

  // dodatno odsecanje kolona koje vec imaju preveliki zbir
  if (sum > col[j])
    return false;

  sum = 0;
  zero = 0;
  for (int k = 0; k < col.size(); k++) {
    sum += mat[i][k];
    if (mat[i][k] == 0)
      zero++;
  }
  if (zero == 0 && sum != row[i])
    return false;

  // dodatno odsecanje vrsta koje vec imaju preveliki zbir
  if (sum > row[i])
    return false;

  return true;
}

int zbirovi(vector< vector<int> >& mat, vector<int>& row, vector<int>& col, int k, int maks_broj) {
  int i = k / col.size();
  int j = k % col.size();

  if (i == row.size())
    return 1;

  if (mat[i][j] != 0) {
    if (!check(mat, row, col, i, j))
      return 0;
    return zbirovi(mat, row, col, k + 1, maks_broj);
  }

  int count = 0;
```

```
  for(int broj = 1; broj <= maks_broj; broj++) {
    mat[i][j] = broj;
    if (check(mat, row, col, i, j))
      count += zbirovi(mat, row, col, k + 1, maks_broj);
  }
  mat[i][j] = 0;
  return count;
}

int main() {
  int n, m, d;
  cin >> n >> m >> d;

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

  vector<int> cols(m);
  for(int i = 0; i < m; i++)
    cin >> cols[i];

  vector< vector<int> > mat(n, vector<int>(m));
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      cin >> mat[i][j];

  cout << zbirovi(mat, rows, cols, 0, d) << endl;
  return 0;
}
```

Задатак: Игра

Два играча играју игру са кулом од n новчића. Играч на потезу може да уклони тачно n_1, n_2 или n_3 новчића са врха. Игра се завршава када играч на потезу не може да направи потез, у ком случају он губи игру. Написати програм који одређује који играч има загарантовану победу. Временска и просторна сложеност алгоритма треба да буду $O(n)$.

Опис улаза

Са стандардног улаза се уноси број игара q ($1 \leq q \leq 20$). У наредних q редова се уносе по четири броја, n ($1 \leq n \leq 10000$), n_1, n_2 и n_3 ($1 \leq n_1, n_2, n_3 \leq n$) који представљају број новчића и могуће потезе једне игре.

Опис излаза

За сваку игру исписати по један број који представља који играч има загарантовану победу. Ако је у питању први играч исписати 1, иначе исписати 2.

Пример

Улаз	*Излаз*
2	1
7 1 2 3	2
8 2 4 6	

Решење

Динамичко програмирање навише

Нека функција $f(t)$ има вредност $\top$ ако и само ако играч на потезу има победничку стратегију за игру у којој се на почетку на гомили налази t новчића. Ако на гомили имамо 0 новчића, тада играч на потезу губи, па је $f(0) = \bot$. Да би играч на потезу имао победничку стратегију када је на гомили t новчића, мора да постоји

потез у ком ће он узети t_i новчића (за неко $t_i \leq t$), такав да се после тога добије ситуација у којој играч на потезу нема победничку стратегију тј.

$$f(t) = \top \iff (\exists i \in \{1, 2, 3\})(t_i \leq t \wedge f(t - t_i) = \bot)$$

Овим је потпуно дефинисана рекурзивна функција којом се израчунава тражени резултат, међутим, ако се она директно имплементира, понављаће се рекурзивни позиви и решење ће бити неефикасно. Боље решење се добија динамичким програмирањем навише, тако да се у низу DP_t чува вредност функције $f(t)$. Низ се попуњава редом, за вредности од 0 до T, где је T учитани број новчића у кули.

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

using namespace std;

int main() {
  int Q;
  cin >> Q;

  for (int q = 0; q < Q; q++) {
    int t, t1, t2, t3;
    cin >> t >> t1 >> t2 >> t3;

    vector<bool> dp(t + 1);
    for(int tt = 1; tt <= t; tt++) {
      dp[tt] = (tt >= t1 && !dp[tt - t1]) ||
               (tt >= t2 && !dp[tt - t2]) ||
               (tt >= t3 && !dp[tt - t3]);
    }

    cout << (dp[t] ? 1 : 2) << '\n';
  }

  return 0;
}
```

Глава 19

Јануар 2023.

Задатак: Флип

Дат је низ од n елемената чије су вредности 0 или 1. Флип је операција којом се 0 претвара у 1. Одредити дужину најдужег низа јединица у датом низу, ако је дозвољено највише k флипова. Временска и просторна сложеност треба да буду $O(n)$.

Опис улаза

Са стандардног улаза уноси се број n ($1 \leq n \leq 10^9$), k ($1 \leq k \leq n$), затим n елемената чије су вредности 0 или 1.

Опис излаза

На стандардни излаз исписати тражени број.

Пример

Улаз	*Излаз*
`8 2` `1 1 0 0 1 0 1 1`	`5`

Објашњење

Објашњење: Флиповањем битова на позицијама 2 и 3 или 3 и 5 добијамо низ јединица дужине 5.

Решење

Два показивача

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  int l = 0;
  int broj_nula = 0;
  int maks_flip = 0;
  for (int d = 0; d < n; d++) {
    if (a[d] == 0)
      broj_nula++;
    while (broj_nula > k) {
```

```
      if (a[l] == 0)
        broj_nula--;
      l++;
    }
    maks_flip = max(maks_flip, d - l + 1);
  }
  cout << maks_flip << endl;
  return 0;
}
```

Два показивача

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  int l = 0, d = 0;
  int broj_nula = 0;
  int maks_flip = 0;
  while (d < n) {
    if (broj_nula < k || a[d] == 1) {
      if (a[d] == 0)
        broj_nula++;
      d++;
      maks_flip = max(maks_flip, d - l);
    } else {
      if (a[l] == 0)
        broj_nula--;
      l++;
    }
  }
  cout << maks_flip << endl;
  return 0;
}
```

Задатак: Мешалица

На располагању нам је мешалица која може да меша редослед елемената низа. Мешалица елементе увек меша на исти начин. У мешалицу је убачен низ a чијим је мешањем добијен низ m. Написати програм који учитава низове a и m, као и низ b и исписује како изгледа низ b након мешања. Временска сложеност треба да буде $O(n \log n)$, а просторна $O(n)$ где је n дужина низова.

Опис улаза

Са стандардног улаза се уноси цео број n ($1 \leq n \leq 10^5$) који представља дужине низова. У наредној линији се уноси n различитих елемената низа a, одвојених размаком. У наредној линији се уноси n различитих елемената низа m, одвојених размаком. У последњој линији се уноси n различитих елемената низа b, одвојених размаком. Сви елементи низова су цели бројеви из интервала $[0, 10^9]$.

Опис излаза

На стандардни излаз у једној линији исписати n елемената низа добијеног мешањем низа b, одвојене размаком.

Пример 1

Улаз	*Излаз*
5 4 3 1 0 2 1 3 2 4 0 3 2 0 4 1	0 2 1 3 4

Пример 2

Улаз	*Излаз*
4 1 2 5 7 5 2 1 7 3 4 1 5	1 4 3 5

Решење

Мапа

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int main() {
  int n;
  cin >> n;

  // na kojoj poziciji se nalazi ucitana vrednost niza
  unordered_map<int, int> pozicija;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    pozicija[x] = i;
  }

  // na koju poziciju se preslikava pozicija i iz originalnog niza
  vector<int> mesalica(n);
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    mesalica[pozicija[x]] = i;
  }

  // promesana verzija ucitanog niza b
  vector<int> b_promesano(n);
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    b_promesano[mesalica[i]] = x;
  }

  for (int x : b_promesano)
    cout << x << " ";
  cout << endl;
}
```

Задатак: Уљез

Црвени астронаут има задатак да унесе тачну шифру како други астронаути не би сазнали да је он заправо ванземаљац. Шифра је дужине n и састоји се из цифара 1, 2 и 3. Црвени је сазнао неке делове шифре, као и да шифра садржи тачно k јединица. Написати програм који помаже Црвеном да испише све шифре које испуњавају дате услове у лексикографском поретку пре него што други астронаути посумњају да је он уљез међу њима.

Опис улаза

Са стандардног улаза се уноси број k ($1 \leq k \leq n$). Након тога се уноси ниска дужине n ($1 \leq n \leq 100$) која се састоји из карактера `.`, `1`, `2` и `3` која одређује која цифра се налази на којој позицији у шифри (тамо где је `.` није позната цифра).

Опис излаза

На стандардни излаз исписати све тражене шифре уређене лексикографски растуће, сваку у засебном реду.

Пример

Улаз	*Излаз*
`3`	`1132213`
`1.32.13`	`1132313`
	`1232113`
	`1332113`

Решење

Реукурзивно генерисање

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

void uljez(string& s, int i, int k, int broj_jedinica) {
  if (broj_jedinica > k || broj_jedinica + (s.length() - i) < k)
    return;
  if (i == s.length())
    cout << s << endl;
  else {
    if (s[i] == '.') {
      for (int x = '1'; x <= '3'; x++) {
        s[i] = x;
        uljez(s, i+1, k, broj_jedinica + (x == '1' ? 1 : 0));
      }
      s[i] = '.';
    } else
      uljez(s, i+1, k, broj_jedinica + (s[i] == '1' ? 1 : 0));
  }
}

void uljez(string& s, int k) {
  uljez(s, 0, k, 0);
}

int main() {
  int k;
  cin >> k;
  string s;
  cin >> s;
  uljez(s, k);
}
```

Задатак: Домине

Дато је n домина поређаних у низ и сваке суседне две домине су на међусобном растојању 1. Домине могу имати различите висине. Могуће је уклонити неке домине из низа (осим прве и последње) и затим оборити прву домину. Прва домина ће оборити прву следећу домину уколико јој је она на удаљености мањој или једнакој од висине прве. Тај поступак се наставља докле год је могуће оборити наредну домину. Написати

програм који одређује на колико начина је могуће издвојити подниз домина тако да се обарањем прве домине обара и последња. Временска сложеност треба да буде $O(n^2)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 1000$). Затим се уноси n елемената a_i ($1 \leq a_i \leq 1000$) који представљају висине домина.

Опис излаза

На стандардни излаз исписати број тражених поднизова. Како тај број може бити јако велик, све операције вршити по модулу $10^9 + 7$.

Пример

Улаз	*Излаз*
`5` `1 3 1 2 1`	`3`

Објашњење

Поднизови `1 3 . . 1`, `1 3 . 2 1` и `1 3 1 2 1` су такви да се обарањем прве домине обара и последња, док нпр. подниз `1 3 1 . 1` није такав. То је зато што се обарањем прве домине обара друга, обарањем друге трећа, али обарањем треће домине (чија је висина 1) није могуће оборити последњу домину (која јој је на удаљености 2).

Решење

Динамичко програмирање навише

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

const int MOD = 1'000'000'007;

using namespace std;

int main() {
  int n;
  cin >> n;

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

  vector<int> dp(n, 0);
  dp[0] = 1;
  for(int i = 1; i < n; i++)
    for(int j = i-1; j >= 0; j--)
      if(j + a[j] >= i)
        dp[i] = (dp[i] + dp[j]) % MOD;

  cout << dp[n - 1] << endl;
  return 0;
}
```

Глава 20

Фебруар 2023.

Задатак: Цене

Магационер Маре добио је задатак да уради специјалан попис робе. У магацину је n артикала, сваки артикал има своју цену c_i која је природан број између 1 и N. Маре је добио k упита облика `a b`, за сваки упит потребно је да одреди број артикала чија је цена између a и b ($a \leq c_i \leq b$). Помозите Марету да уради попис.

Временска сложеност: $O(n + N + k)$

Просторна сложеност: $O(n + N + k)$

Опис улаза

Са стандардног улаза уноси се број артикала n ($1 \leq n \leq 10^6$), затим n целих бројева c_i ($1 \leq c_i \leq 10^6$) који представљају цене артикала. Након тога се уноси број упита k ($1 \leq k \leq n$), затим k упита облика `a b` ($0 \leq a, b \leq 10^6$).

Опис излаза

На стандардни излаз за сваки упит исписати тражени број .

Пример

Улаз	*Излаз*
7	4
1 3 1 3 2 4 1	6
5	7
2 4	3
1 3	3
0 5	
1 1	
3 4	

Решење

Опис главног решења

У овом блоку се описује главно решење задатка.

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

using namespace std;

const int N = 1e6;

int main() {
  vector<int> brojProizvoda(N);
  int n;
```

```
  cin >> n;
  for (int i = 0; i < n; i++) {
    int cena;
    cin >> cena;
    brojProizvoda[cena]++;
  }
  vector<int> ps(N+1, 0);
  for (int i = 0; i < N; i++)
    ps[i+1] = ps[i] + brojProizvoda[i];

  int k;
  cin >> k;
  for (int i = 0; i < k; i++) {
    int a, b;
    cin >> a >> b;
    cout << ps[b+1] - ps[a] << endl;
  }

  return 0;
}
```

Задатак: Сеча стабала

Дуж x-осе постављено је n стабала. Стабла могу бити различите ширине и висине. Стабла се могу хоризонтално преклапати тј. више стабала може истовремено заузимати исту позицију на x-оси. Потребно је направити пресек у једној тачки на x-оси тако да укупна количина дрвета са десне стране буде што већа, али не већа од задатог броја M. Написати програм који одређује где је потребно направити пресек, као и количину стабала са десне стране након пресека. Временска сложеност треба да буде $O(n \log v_{max})$ где је v_{max} максимална x координата на којој има стабала, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уносе бројеви M ($1 \leq M \leq 10^{18}$) и n ($1 \leq n \leq 50000$). Након тога се у n линија уносе по три броја l_i, r_i ($0 \leq l_i < r_i \leq 10^9$) и v_i ($1 \leq d_i \leq 10$) који редом представљају x координату почетка, x координату краја и висину i-тог стабла.

Опис излаза

На стандардни излаз исписати два цела броја одвојена размаком: x координату траженог сечења и количину дрвета са десне стране сечења.

Пример

Улаз	*Излаз*
6 3	2 4
0 2 1	
1 3 2	
1 4 1	

Објашњење

Објашњење: Сечењем у тачки 2 са десне стране не остаје ништа од првог стабла, остаје комад дужине 1 од другог стабла и остаје комад дужине 2 трећег стабла. Како су дебљине стабала редом 1, 2 и 1, укупна количина стабала је $0 \cdot 1 + 1 \cdot 2 + 2 \cdot 1 = 4$.

Решење

Опис главног решења

У овом блоку се описује главно решење задатка.

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

```cpp
#include <algorithm>

using namespace std;

struct stablo {
  long long x_poc, x_kraj, visina;
};

long long kolicina(long long x, const vector<stablo>& stabla) {
  long long rezultat = 0;
  for (const stablo& s : stabla) {
    // sirina stabla iza tacke secenja
    long long duzina = max(s.x_kraj - max(x, s.x_poc), 0ll);
    rezultat += duzina * s.visina;
  }
  return rezultat;
}

int main() {
  long long t;
  cin >> t;

  int n;
  cin >> n;
  vector<stablo> stabla(n);
  for (int i = 0; i < n; i++)
    cin >> stabla[i].x_poc >> stabla[i].x_kraj >> stabla[i].visina;

  // maksimalna x koordinata na kojoj se nalazi neko stablo
  long long x_max = 0;
  for (int i = 0; i < n; i++)
    x_max = max(x_max, stabla[i].x_kraj);

  // binarnom pretragom pronalazimo optimum
  long long levo = 0, desno = x_max;
  while (levo <= desno) {
    long long sredina = levo + (desno - levo) / 2;
    if (kolicina(sredina, stabla) <= t)
      desno = sredina - 1;
    else
      levo = sredina + 1;
  }
  cout << levo << " " << kolicina(levo, stabla) << endl;
}
```

Задатак: Збир растућег

Написати програм који за дати низ природних бројева одређује максималан збир који је могуће добити сабирањем елемената неког његовог строго растућег подниза. Временска сложеност треба да буде $O(n^2)$, а просторна $O(n)$.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 1000$), након чега се уноси n бројева a_i ($1 \leq a_i \leq 10^6$) који представљају елементе низа.

Опис излаза

На стандардни излаз исписати један природан број који представља тражену вредност.

Пример

Улаз	*Излаз*
`6` `4 2 3 7 5 6`	`16`

Објашњење

Подниз `2 3 5 6` има највећи збир 16 међу свим растућим поднизовима.

Решење

Динамичко програмирње навише

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

using namespace std;

int main() {
  int n;
  cin >> n;

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

  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = a[i];
    for (int j = i-1; j >= 0; j--)
      if (a[j] < a[i])
        dp[i] = max(dp[i], dp[j] + a[i]);
  }

  cout << *max_element(begin(dp), end(dp)) << endl;
  return 0;
}
```