Probleme rezolvate cu heap-uri

Mirel Coşulschi
mirelc@central.ucv.ro

Mihai Gabroveanu
mihaiug@central.ucv.ro

April, 2023

1 Probleme

1. Un *max-heap* este un arbore binar complet cu următoarea proprietate suplimentară: valoarea din orice nod este mai mare sau egală cu valorile din orice nod descendent.

   Similar se defineşte noţiunea de *min-heap*: valoarea din orice nod este mai mică sau egală cu valorile descendenţilor.

   Într-un *max-heap* rădăcina are valoare maximă, iar într-un *min-heap* rădăcina are valoare minimă. Nu se precizează nicio relaţie între valorile din fiii unui nod.

   Arborele binar complet din figura 1 este un *max-heap*.

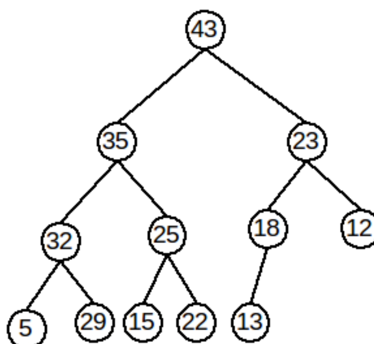

Fig. 1: Exemplu de arbore binar complet care este un *max-heap*.

   Pentru ca un arbore binar complet să fie *max-heap* (similar pentru *min-heap*), fiecare nod din arbore trebuie:

   - să fie mai mare sau egal cu descendenţii săi (dacă există);
   - să fie mai mic sau egal cu tatăl său (dacă există).

   Să presupunem că un arbore binar complet $H[]$ are proprietatea de max-heap, cu excepţia unui nod k. Cum îl corectăm, astfel încât să devină *max-heap*?! Distingem două cazuri:

   - dacă nodul k este mai mare decât tatăl său ($H[k] > H[k/2]$) îl vom muta în sus în arbore, până când acesta devine *max-heap*. Această operaţie se numeşte promovare a unui nod în heap;

- dacă nodul k este mai mic decât cel puţin unul dintre fii săi ($H[k] < H[2*k]$ şi / sau $H[k] < H[2*k+1]$) îl vom muta în jos în mod convenabil, până când arborele devine *max-heap*. Această operaţie se numeşte cernere a unui nod în arbore.

Se consideră o colecţie de numere naturale, iniţial vidă. Asupra ei se fac două tipuri de operaţii:

- `1 x` - valoarea x se adaugă în colecţie;
- `2` - cea mai mare valoare din colecţie se afişează, apoi se elimină din colecţie.

Dându-se un şir de m operaţii, să se afişeze în ordine rezultatele operaţiilor de tip 2.

Date de intrare

Fişierul de intrare `heap.in` conţine pe prima linie numărul m, $1 \leq m \leq 250.000$, iar pe următoarele m linii câte o operaţie. Pentru operaţiile de tip 1, $1 \leq x \leq 10^9$.

Date de ieşire

Fişierul de ieşire `heap.out` va conţine rezultatele operaţiilor de tip 2, câte unul pe o linie, în ordinea în care au fost efectuate.

Exemplu

heap.in	heap.out
12 1 18 1 12 1 3 2 1 3 1 15 2 2 1 8 2 1 19 2	18 15 12 8 19

(Heap, `https://www.pbinfo.ro/probleme/1855/heap`)

Rezolvare:

Listing 1: `heap.c`

```
#include <stdio.h>

#define NMAX 250000

typedef struct heap {
          int n;            // numarul de elemente din heap
          int a[NMAX + 1]; // valorile elementelor aflate in heap
        } HEAP;

HEAP h;  // un max-heap
```

```
/*
*  Initializeaza un max-heap vid.
*/
void init(HEAP* h) {
  h->n = 0;
}

/*
*  Functia verifica daca un max-heap h este vid. Intoarce valoarea
* 1 daca max-heap-ul h nu are niciun element sau 0 daca max-heap-ul
* contine cel putin un element.
*/
int isempty(HEAP* h) {
  return (h->n == 0);
}

/*
*   Functia insereaza un element de valoare v in max-heap-ul h. Se
* incrementeaza numarul de elemente din heap, se adauga un element de
* valoare v pe ultima pozitie si apoi se reorganizeaza max-heap-ul.
*  Elementul de pe ultima pozitie va fi promovat in max-heap atata timp cat
* nu se verifica proprietatea de max-heap, putand ajunge pana in radacina.
*/
void insert(HEAP* h, int v) {
  int i, j, aux;

  h->a[++h->n] = v;
  i = h->n;
  j = i >> 1;
  while (j > 0) {
    if (h->a[j] < h->a[i]) {
      aux = h->a[j]; h->a[j] = h->a[i]; h->a[i] = aux;
      i = j;
      j = i >> 1;
    } else {
        j = 0;
    }
  }
}

/*
*  Se reorganizeaza max-heap-ul h. Pornind de la elementul aflat in radacina,
* pe pozitia 1, se cauta o pozitie pentru acesta, coborandu-se in cadrul
* arborelui respectiv, astfel incat sa se pastreze proprietatea de max-heap.
*/
void push(HEAP* h) {
  int i, j, aux;

  i = 1;
  j = i << 1;
  while (j <= h->n) {
```

```
    if ((j < h->n) && (h->a[j] < h->a[j + 1])) {
      j++;
    }

    if (h->a[i] < h->a[j]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = h->n + 1;
    }
  }
}

/*
*   Se sterge elementul aflat in varful max-heap-ului: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de max-heap.
*   Functia returneaza cea mai mare valoare din max-heap (valoarea aflata
* in radacina inainte de stergere).
*/
int deletemax(HEAP* h) {
  int aux = h->a[1];

  h->a[1] = h->a[h->n];
  h->n--;
  push(h);

  return aux;
}

int main() {
  FILE *fin, *fout;
  int m, i, v, op;

  fin = fopen("heap.in", "r");
  fout = fopen("heap.out", "w");

  init(&h);                 // se initializeaza max-heap-ul

  fscanf(fin, "%d", &m);    // se citeste numarul de elemente
  for (i = 0; i < m; i++) {
    fscanf(fin, "%d", &op); // se citeste codul operatiei

    if (op == 1) {
      fscanf(fin, "%d", &v); // se citeste valoarea unui element

      insert(&h, v);        // se insereaza elementul v in max-heap
    } else {
        if (!isempty(&h)) {   // daca max-heap-ul nu este vid
```

```
            // se sterge elementul din varful max-heap-ului
            fprintf(fout, "%d\n", deletemax(&h));
        }
    }
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

2. Se dă un şir $a[1], a[2], \ldots, a[n]$ de numere naturale şi un număr natural k.

   Să se determine cele mai mari k numere din şir.

   Date de intrare

   Programul citeşte de la tastatură numerele `n, k, A, B, C, D`. Şirul de numere se va genera după formula:

$$a[i] = \begin{cases} A, & \text{pentru } i = 1 \\ (B * a[i-1] + C)\%D, & \text{pentru } i = 2 \ldots n \end{cases}$$

   Pentru datele de intrare avem următoarele restricţii: $1 \leq n \leq 5 \cdot 10^6$, $1 \leq k \leq \min(10^5, n)$, $1 \leq A, B, C, D \leq 10^9$.

   Date de ieşire

   Programul va afişa pe ecran, ordonate crescător, cele mai mari k numere din şir.

   Exemplu

Intrare
10 4 13 23 47 97

Iesire
55 56 74 96

   (Lastk, `https://www.pbinfo.ro/probleme/3011/lastk`)

   Rezolvare:
   Se va utiliza o structură de date de tip *min-heap* în care vor fi păstrate cele mai mari k valori întâlnite în cadrul şirului de valori calculate.

   Se generează mai întâi primele k valori şi se inserează în *min-heap*. Începând cu cea de-a $k+1$ valoare generată din şir, dacă este mai mică sau egală decât valoarea rădăcinii min-heap-ului, atunci se ignoră. Dacă are valoarea mai mare decât valoarea rădăcinii min-heap-ului, atunci se înlocuieşte valoarea rădăcinii min-heap-ului cu noua valoare şi se reorganizează structura pentru a păstra proprietatea de *min-heap*.

   După ce s-au generat cele n valori ale şirului se afişează conţinutul *min-heap-ului*: valorile aflate în acel moment sunt afişate în ordine crescătoare.

Listing 2: lastk.c

```
#include <stdio.h>

#define NMAX 100000

typedef struct heap {
          int lastidx;     // numarul de elemente din heap
          int a[NMAX + 1]; // valorile elementelor aflate in heap
        } HEAP;

HEAP h; // min-heap

/*
*  Promoveaza in heap ultimul nod pentru a se restabili proprietatea
* de min-heap. Elementul de pe ultima pozitie va fi promovat in heap
* atata timp cat nu se verifica proprietatea de min-heap, putand ajunge
* pana in radacina.
*/
void liftup(HEAP* h) {
  int i, ip2, aux;

  i = h->lastidx;
  while ((i > 1) && (h->a[i >> 1] > h->a[i])) {
    ip2 = i >> 1;

    aux = h->a[ip2]; h->a[ip2] = h->a[i]; h->a[i] = aux;

    i = ip2;
  }
}

/*
*  Se reorganizeaza structura de min-heap pentru nodul radacina. Pornind
* de la elementul aflat pe pozitia 1, se verifica conditia de min-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta, coborandu-se
* in cadrul arborelui.
*/
void pushdown(HEAP* h) {
  int i, j, aux;

  i = 1;
  j = i << 1;
  while (j <= h->lastidx) {
    if ((j + 1 <= h->lastidx) && (h->a[j] > h->a[j + 1])) {
      j++;
    }

    if (h->a[i] > h->a[j]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
    } else {
```

```
        i = h->lastidx + 1;
    }
    j = i << 1;
  }
}

/*
*   Functia insereaza un element de valoare v in min-heap-ul h.
*   Se incrementeaza numarul de elemente din heap, se adauga un element
*  de valoare v pe ultima pozitie si apoi se reorganizeaza min-heap-ul.
*/
void hinsert(HEAP* h, int v) {
  h->lastidx++;
  h->a[h->lastidx] = v;
  liftup(h);
}

/*
*   Se sterge elementul aflat in varful min-heap-ului: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de min-heap.
*   Functia returneaza cea mai mica valoare din min-heap (valoarea aflata
* in radacina inainte de stergere).
*/
void hdelete(HEAP* h) {
  h->a[1] = h->a[h->lastidx];
  h->lastidx--;
  pushdown(h);
}


int main() {
  int n, k, a, b, c, d, x, i;

  scanf("%d %d %d %d %d %d", &n, &k, &a, &b, &c, &d);

  x = a;
  h.a[++h.lastidx] = x;    // se adauga in min-heap primul element
  for (i = 1; i < n; i++) {
    x = (1LL * b * x + c) % d; // se calculeaza urmatorul element

    if (h.lastidx < k) { // daca in heap avem mai putin de k elemente
      hinsert(&h, x);    // inseram in min-heap un nou element
    } else {
        if (h.a[1] < x) { // daca cel mai mic element este mai mic decat x
          h.a[1] = x;     // inlocuim valoarea radacinii cu x
          pushdown(&h);   // reorganizam min-heap-ul
        }
    }
  }
```

```
  while (h.lastidx) {      // cat timp avem elemente in heap
    printf("%d ", h.a[1]); // afisam valoarea radacinii

    hdelete(&h);           // stergem radacina min-heap-ului
  }

  return 0;
}
```

3. În urma referendumului a rămas doar un şir de numere naturale $a_1, a_2, \ldots, a_n$.

   Să se determine cel mai mic număr ce apare exact o dată în şir.

   Date de intrare

   Programul citeşte de la tastatură numărul n, apoi şirul de n numere naturale, separate prin spaţii.

   Se garantează că există cel puţin un număr ce apare o singură dată, şi $10 \leq n \leq 10^6$, $0 \leq a_i \leq 2 \cdot 10^9$.

   Date de ieşire

   Programul va afişa pe ecran numărul m, reprezentând numărul minim ce apare exact o dată în şir.

   Exemplu

Intrare	Iesire
10 5 3 8 7 3 3 2 5 9 5	2

(H2, `https://www.pbinfo.ro/probleme/2628/h2`)

Rezolvare:
Se va utiliza o structură de date de tip *min-heap* în care vor fi păstrate valorile şirului de intrare.

Mai întâi valorile şirului citit se organizează ca *min-heap*. În continuare se va căuta cea mai mică valoare ce apare o singură dată. Se memorează separat valoarea rădăcinii (valoarea minimă) în variabila u şi se şterge nodul rădăcină.

Operaţia de ştergere se realizează astfel încât structura de date să îşi păstreze proprietatea de *min-heap*. Atâta timp cât valoarea rădăcinii este egală cu valoarea variabilei u se şterge nodul rădăcină. În momentul în care valoarea nodului rădăcină este diferită de valoarea păstrată în u, se verifică dacă a fost şters vreun nod rădăcină având valoarea egală cu u (variabila `equal`). Dacă nu a fost şters un astfel de nod (`equal == 0`), atunci valoarea lui u este valoarea căutată. Dacă `equal == 1`, atunci au fost şterse cel puţin două valori egale cu u, şi se trece la valoarea următoare.

Listing 3: h2.c

```
#include <stdio.h>

#define NMAX 1000000

int a[NMAX + 1]; // valorile elementelor aflate in min-heap
int n;           // numarul de elemente din heap

/*
*  Se reorganizeaza structura de min-heap pentru nodul start. Pornind de
* la elementul aflat pe pozitia start, se verifica conditia de min-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta,
* coborandu-se in cadrul subarborelui de radacina start.
*  start - indicele nodului reprezentand radacina unui subarbore
*  m - numarul de element din heap
*/
void push(int start, int m) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (a[j] > a[j + 1])) {
      j++;
    }

    if (a[i] > a[j]) {
      aux = a[i]; a[i] = a[j]; a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

/*
*   Se sterge elementul aflat in varful min-heap-ului: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de min-heap.
*   Functia returneaza cea mai mica valoare din min-heap (valoarea aflata
* in radacina inainte de stergere).
*/
int getmin() {
  int vmin = a[1];

  a[1] = a[n];
  n--;
  push(1, n);
```

```
  return vmin;
}

int main() {
  int i, u, v, equal;

  scanf("%d", &n);

  for (i = 1; i <= n; i++) {
    scanf("%d", &a[i]);
  }

  // construieste un min-heap din multimea de elemente 1..n.
  for (i = n / 2; i > 0; i--) {
    push(i, n);
  }

  u = getmin();        // se obtine valoarea minima din heap
  equal = 1;
  // cat timp mai sunt elemente in heap si ultimele doua
  // elemente extrase din min-heap sunt egale
  while (n && equal) {
    equal = 0;
    v = getmin();     // se obtine urmatoarea valoare minima din heap
    // cat timp mai sunt elemente in heap si ultimele doua
    // elemente extrase din min-heap sunt egale
    while (n && (u == v)) {
      v = getmin();   // se obtine urmatoarea valoare minima din heap
      equal = 1;
    }

    if (equal == 1) {
      u = v;          // se pastreaza in u ultima valoare extrasa din heap
    }
  }

  printf("%d", u);

  return 0;
}
```

4. Presupunem că avem n numere prime notate $a_1, a_2, \ldots, a_n$ sortate strict crescător. Formăm un şir strict crescător b ale cărui elemente sunt toţi multiplii acestor n numere prime astfel încât, multipli comuni apar o singură dată. Presupunem că numerotarea poziţiilor elementelor din şirul b începe tot cu 1.

   Scrieţi un program care citeşte din fişierul de intrare valoarea lui n şi apoi cele n elemente ale şirului a, determină elementul de pe poziţia m din şirul b şi afişează în fişierul de ieşire valoarea acestuia.

   Date de intrare

   Fişierul de intrare `numar6.in` conţine pe prima linie două numere naturale, separate

printr-un spaţiu, şi care reprezintă valoarea lui n, $n \leq 100$, respectiv valoarea lui m, $m \leq 15.000$.

Pe a doua linie avem n numere naturale prime, separate prin câte un spaţiu, care reprezintă valorile elementelor şirului a ($a_1 \leq 1000$, $a_n \leq 10^6$). Aceste numere sunt dispuse în ordine strict crescătoare.

Date de ieşire

Fişierul de ieşire `numar6.out` va conţine pe prima linie o singură valoare ce reprezintă termenul de pe poziţia m din şirul b.

Exemplu

numar6.in	numar6.out
3 11111 977 1009 1031	3726237

(Numar6, `https://www.pbinfo.ro/probleme/2174/numar6`)

Rezolvare:
Pentru rezolvarea problemei se va utiliza o structură de date de tip *min-heap* în care vor fi păstrate valorile şirului B, sub forma unor elemente de tip `PAIR`:

```
typedef struct pair {
          int idx;   // indicele unui element din tabloul A
          int mul;   // factor de multiplicare
        } PAIR;
```

Valoarea naturală a unui element v de tip `PAIR`, se determină astfel: $a[v.idx] * v.mul$, unde $a[v.idx]$ reprezintă un număr prim aflat pe poziţia $v.idx$ în şirul datelor de intrare.

Se inserează în min-heap toate numerele prime din şirul de intrare, fiecare având factorul de multiplicare 1. Apoi se obţine valoarea elementului aflat în rădăcina min-heap-ului. Dacă această valoare este diferită de valoarea elementului extras la pasul anterior, atunci se decrementează un contor ce păstrează numărul de numere distincte extrase din heap. Se incrementează cu 1 valoarea factorului de multiplicitate al elementului din rădăcină, se actualizează valoarea elementului din rădăcină şi se reorganizează min-heap-ul.

Listing 4: `numar6.c`

```
#include <stdio.h>

#define NMAX 100

typedef struct pair {
          int idx;   // indicele unui element din tabloul A
          int mul;   // factor de multiplicare
        } PAIR;

typedef struct heap {
          int n;             // numarul de elemente din heap
          PAIR a[NMAX + 1]; // valorile elementelor aflate in min-heap
        } HEAP;

HEAP h;           // un min-heap
int a[NMAX + 1]; // a[k] - al k-lea numar prim din datele de intrare
```

```
/*
*  Functia compara valorile a doua variabile de tip structura.
*  Functia returneaza -1 daca primul element are valoarea mai mica decat
* valoarea celui de-al doilea element.
*  Functia returneaza 1 daca primul element are valoarea mai mare decat
* valoarea celui de-al doilea element.
*  Functia returneaza -0 daca primul element are valoarea egala cu
* valoarea celui de-al doilea element.
*/
int cmp(const void* x, const void* y) {
  long long vx, vy;
  PAIR* px = (PAIR*)x;
  PAIR* py = (PAIR*)y;

  vx = 1LL * a[px->idx] * px->mul;
  vy = 1LL * a[py->idx] * py->mul;

  if (vx < vy) {
    return -1;
  } else {
      if (vx > vy) {
        return 1;
      } else {
          return 0;
      }
  }
}

/*
*  Initializeaza un min-heap vid.
*/
void init(HEAP* h) {
  h->n = 0;
}

/*
*   Functia verifica daca un min-heap h este vid. Intoarce valoarea
*  1 daca min-heap-ul h nu are niciun element sau 0 daca min-heap-ul
*  contine cel putin un element.
*/
int isempty(HEAP* h) {
  return (h->n == 0);
}

/*
*  Functia insereaza un element u in min-heap-ul h. Se incrementeaza numarul
* de elemente din heap, se adauga elementul u pe ultima pozitie si apoi se
* reorganizeaza min-heap-ul.
*/
void insert(HEAP* h, PAIR u) {
  int i, j;
```

```
  PAIR aux;

  j = ++h->n;
  h->a[j] = u;
  i = j / 2;
  while ((0 < i) && (cmp(&h->a[i], &h->a[j]) > 0)) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

    j = i;
    i = j / 2;
  }
}

/*
*  Se reorganizeaza structura de min-heap pentru nodul radacina. Pornind
* de la elementul aflat pe pozitia 1, se verifica conditia de min-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta,
* coborandu-se in cadrul arborelui.
*/
void push(HEAP* h) {
  int i, j;
  PAIR aux;

  i = 1;
  while (2 * i <= h->n) {
    j = 2 * i;
    if ((j < h->n) && (cmp(&h->a[j], &h->a[j + 1]) > 0)) {
      j++;
    }

    if (cmp(&h->a[i], &h->a[j]) > 0) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
    } else {
        i = h->n;
    }
  }
}

/*
*   Se sterge elementul aflat in varful min-heap-ului: pe prima pozitie
* este adus un element u si se reorganizeaza structura de date astfel
* incat sa se pastreze proprietatea de min-heap.
*/
void deleteMin(HEAP* h, PAIR u) {
  h->a[1] = u;

  push(h);
}

int main() {
```

```
  FILE *fin, *fout;
  int n, m, i;
  PAIR v, lastv;

  fin = fopen("numar6.in", "r");
  fout = fopen("numar6.out", "w");

  fscanf(fin, "%d %d", &n, &m);
  for (i = 1; i <= n; i++) {
    fscanf(fin, "%d", &a[i]);
  }

  init(&h);             // se initializeaza min-heap-ul h

  for (i = 1; i <= n; i++) {
    v.idx = i;          // i este indicele elementului a[i]
    v.mul = 1;          // valoarea de multiplicare 1

    insert(&h, v);      // se insereaza elementul v in min-heap-ul h
  }

  lastv.idx = 0;        // indicele primului element din tabloul A
  lastv.mul = 1;        // valoarea de multiplicare 1
  while (m) {           // cat timp nu am numarat m elemente distincte
    v = h.a[1];         // elementul aflat in radacina (minim)

    if (cmp(&v, &lastv) != 0) { // daca v si lastv sunt distincte
      m--;              // se decrementeaza valoarea contorului
    }
    lastv = v;          // salvam ultimul element extras din heap

    v.mul++;            // incrementam factorul de multiplicare
    deleteMin(&h, v); // inlocuim valoarea anterioara din radacina
                        // cu noua valoare a lui v si se reorganizeaza
                        // min-heap-ul
  }

  // afisam valoarea ultimului element extras din heap
  fprintf(fout, "%lld", 1LL * a[lastv.idx] * lastv.mul);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

5. Se dă un şir cu n numere întregi. Să se ordoneze descrescător valorile şirului cu ajutorul unui *min-heap*.

   Date de intrare

   Fişierul de intrare `heapsort.in` conţine pe prima linie numărul n, $1 \leq n \leq 100.000$, iar pe următoarea linie n numere naturale, separate prin exact un spaţiu, reprezentând valorile

elementelor şirului.

Date de ieşire

Fişierul de ieşire `heapsort.out` va conţine pe o linie n numere naturale, separate prin exact un spaţiu, reprezentând valorile elementelor şirului ordonate descrescător.

Exemplu

heapsort.in	heapsort.out
10 11 2 12 4 6 9 8 5 17 13	17 13 12 11 9 8 6 5 4 2

Rezolvare:
Pe baza datelor de intrare se construieşte un min-heap (un arbore binar special) ale cărui noduri sunt alocate static.

Listing 5: `heapsort.c`

```
#include <stdio.h>

#define NMAX 100000

int a[NMAX + 1]; // valorile elementelor aflate in heap
int n;           // numarul de elemente din heap

/*
*  Se reorganizeaza structura de min-heap pentru nodul start. Pornind de
* la elementul aflat pe pozitia start, se verifica conditia de min-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta,
* coborandu-se in cadrul subarborelui de radacina start.
*/
void push(int start, int m) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (a[j] > a[j + 1])) {
      j++;
    }

    if (a[i] > a[j]) {
      aux = a[i]; a[i] = a[j]; a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

int main() {
  FILE *fin, *fout;
  int n, i, aux;
```

```
  fin = fopen("heapsort.in", "r");
  fout = fopen("heapsort.out", "w");

  fscanf(fin, "%d", &n);
  for (i = 1; i <= n; i++) {
    fscanf(fin, "%d", &a[i]);
  }

  // construieste un min-heap din multimea de elemente 1..n.
  for (i = n / 2; i > 0; i--) {
    push(i, n);
  }

  for (i = n; i > 1; i--) {
    // cel mai mic element se salveaza pe pozitia i, iar elementul
    // aflat anterior pe pozitia i se salveaza in radacina heap-ului
    aux = a[1]; a[1] = a[i]; a[i] = aux;

    push(1, i - 1);   // reorganizam min-heap-ul pt elementele
                      // de indici 1...(i-1)
  }

  // se afiseaza valorile sirului ordonat descrescator
  for (i = 1; i <= n; i++) {
    fprintf(fout, "%d ", a[i]);
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

6. Problema `https://www.nerdarena.ro/problema/competitie`

   (Competitie, Shumen 2010 Juniori, `https://www.nerdarena.ro/problema/competitie`)

   Rezolvare:

Listing 6: `competitie.c`

```
#include <stdio.h>
#include <stdlib.h>

#define KMAX 10001
#define NMAX 1001

typedef struct competitor {
           int idx;           // indicele concurentului
           int lastlaptime;   // timpul de parcurgere al ultimei curse
           int laps;          // numarul de curse efectuate
           int currenttime;   // timpul trecut de la inceputul cursei
        } COMPETITOR;
```

```
typedef struct heap {
          COMPETITOR a[KMAX];
          int m;              // numarul de competitori din heap
        } HEAP;

int ms[KMAX];         // timpul initial necesar pt parcurgerea unei ture
int p[KMAX];          // perioada dupa care se reseteaza timpul de parcurgere
HEAP h;               // min-heap organizat dupa valoarea currenttime
                      // a fiecarui concurent

/*
*  Initializeaza un min-heap vid.
*/
void init(HEAP* h) {
  h->m = 0;
}

/*
*   Functia verifica daca un min-heap h este vid. Intoarce valoarea
*  1 daca min-heap-ul h nu are niciun element sau 0 daca min-heap-ul
*  contine cel putin un element.
*/
int empty(HEAP* h) {
  return (h->m == 0);
}

/*
*  Functia insereaza un element c in min-heap-ul h. Se incrementeaza numarul
* de elemente din heap, se adauga elementul c pe ultima pozitie si apoi se
* reorganizeaza min-heap-ul.
*/
void insert(HEAP* h, COMPETITOR c) {
  int i;
  COMPETITOR tmp;

  h->m++;
  h->a[h->m] = c;

  i = h->m;
  while ((1 < i) && (h->a[i].currenttime < h->a[i / 2].currenttime)) {
    tmp = h->a[i]; h->a[i] = h->a[i / 2]; h->a[i / 2] = tmp;

    i = i / 2;
  }
}

/*
*   Functia intoarce elementul aflat pe prima pozitie in min-heap
* (radacina heap-ului binar).
*/
COMPETITOR* minh(HEAP* h) {
```

```
  return &h->a[1];
}

/*
*  Se reorganizeaza structura de min-heap pentru nodul radacina. Pornind
* de la elementul aflat pe pozitia 1, se verifica conditia de min-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta,
* coborandu-se in cadrul arborelui.
*/
void push(HEAP* h) {
  int i, j;
  COMPETITOR tmp;

  i = 1;
  while (2 * i <= h->m) {
    if ((2 * i + 1 > h->m)
        || (h->a[2 * i].currenttime < h->a[2 * i + 1].currenttime)) {
       j = 2 * i;
    } else {
       j = 2 * i + 1;
    }

    if (h->a[i].currenttime > h->a[j].currenttime) {
      tmp = h->a[i]; h->a[i] = h->a[j]; h->a[j] = tmp;

      i = j;
    } else {
        i = h->m;
    }
  }
}

/*
*   Se sterge elementul aflat in varful min-heap-ului: pe prima pozitie
* este adus elementul de pe ultima pozitie, se decrementeaza numarul
* de elemente si se reorganizeaza structura de date astfel incat
* sa se pastreze proprietatea de min-heap.
*/
void deletemin(HEAP* h) {
  h->a[1] = h->a[h->m];
  h->m--;

  push(h);
}


int main() {
  FILE *fin, *fout;
  int k, n, i, oldfinishtime, finish, maxfinish;
  COMPETITOR* c;

  fin = fopen("competitie.in", "r");
```

```
fout = fopen("competitie.out", "w");

fscanf(fin, "%d %d\n", &k, &n);

// se initializeaza heap-ul vid
init(&h);

// se aloca spatiu pentru tabloul in care se vor memora informatii
// despre competitori
c = (COMPETITOR*)malloc(sizeof(COMPETITOR));
for (i = 1; i <= k; i++) {
  fscanf(fin, "%d %d", &ms[i], &p[i]);

  c->laps = 1;
  c->currenttime = ms[i];
  c->idx = i;
  c->lastlaptime = ms[i];

  insert(&h, *c);
}

oldfinishtime = 0;
maxfinish = 0;
while (!empty(&h)) {
  // iau elementul aflat in varful heap-ului
  c = minh(&h);

 // daca timpul total la care termina o tura concurentul curent este egal
 // cu timpul anterior al altui concurent
  if (c->currenttime == oldfinishtime) {
    // se incrementeaza nr de concurenti ce termina o tura simultan
    finish++;
  } else { // altfel, se actualizeaza timpul curent
     oldfinishtime = c->currenttime;
     finish = 1;
  }

  if (finish > maxfinish) {
    maxfinish = finish;
  }

  // daca concurentul curent nu a efectuat n ture
  if (c->laps < n) {
    // se incrementeaza timpul de parcurgere a unei ture
    c->lastlaptime++;

    // daca ne aflam dupa o perioada p[i]
    if (c->laps % p[c->idx] == 0) {
      // atunci timpul de parcurgere al unei ture este resetat
      // la valoarea initiala
      c->lastlaptime = ms[c->idx];
    }
```

```
      // actualizez timpul trecut de la inceputul cursei, cand va termina
      // urmatoarea tura
      c->currenttime += c->lastlaptime;
      // incrementez numarul de ture efectuate
      c->laps++;

      // nodul radacina il cobor in heap pe pozitia corespunzatoare
      // valorii actualizate
      push(&h);
    } else {
      // altfel, nodul radacina este sters
      deletemin(&h);
    }
  }

  fprintf(fout, "%d\n", maxfinish);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

7. Problema https://open.kattis.com/problems/annoyedcoworkers

   (Annoyed Coworkers, 2019 Virginia Tech High School Programming Contest, https://open.kattis.com/problems/annoyedcoworkers)

   Rezolvare:

Listing 7: annoyedcoworkers.c

```
#include <stdio.h>

#define NMAX 100000

typedef struct person {
          long long a;  // gradul curent de frustrare
          int d;        // nivelul de crestere al frustrarii
        } PERSON;

typedef struct heap {
          int n;           // numarul de elemente din heap
          int a[NMAX + 1]; // valorile elementelor aflate in heap
          int (*cmp)(int, int);
        } HEAP;

HEAP pq;        // un min-heap
PERSON c[NMAX]; // lista persoanelor angajate

/*
*  Functie de comparare pentru min-heap.
*/
```

```
int greater(int i, int j) {
  return (c[i].a + c[i].d > c[j].a + c[j].d);
}

long long max(long long a, long long b) {
  return ((a > b) ? a : b);
}

/*
*  Initializeaza un heap vid.
*/
void init(HEAP* h, int (*fcmp)(int, int)) {
  h->n = 0;
  h->cmp = fcmp;
}

/*
*  Functia verifica daca un heap-ul h este vid. Intoarce valoarea
* 1 daca heap-ul h nu are niciun element sau 0 daca heap-ul
* contine cel putin un element.
*/
int isempty(HEAP* h) {
  return (h->n == 0);
}

/*
*  Se reorganizeaza heap-ul h. Pornind de la elementul aflat pe pozitia 1,
* pozitia radacinii, se cauta o pozitie pentru acesta coborandu-se in cadrul
* arborelui respectiv astfel incat sa se pastreze proprietatea de min-heap.
*/
void push(HEAP* h, int start) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= h->n) {
    if ((j < h->n) && (h->cmp(h->a[j], h->a[j + 1]))) {
      j++;
    }

    if (h->cmp(h->a[i], h->a[j])) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = h->n + 1;
    }
  }
}
```

```
int main() {
  int h, n, i, j;
  long long vmax;

  scanf("%d %d", &h, &n);

  // initializam heap-ul
  init(&pq, greater);

  // citim datele de intrare
  for (i = 0; i < n; i++) {
    scanf("%lld %d", &c[i].a, &c[i].d);

    pq.a[i] = i;
  }
  pq.n = n;

  // organizam min-heap-ul
  for (i = n / 2; i > 0; i--) {
    push(&pq, i);
  }

  // cat timp nu am terminat
  while (h--) {
    // alegem persoana cu gradul cel mai mic de frustrare
    j = pq.a[1];

    // actualizam gradul de frustrare al persoanei curente
    c[j].a += c[j].d;
    // reorganizam heap-ul in functie de frustrarea viitoare:
    // in radacina heap-ului se va afla persoana care ar ajunge la cel
    // mai mic grad de frustrare dintre toate persoanele, daca ar fi
    // aleasa
    push(&pq, 1);
  }

  // determinam gradul de frustrare curent maxim
  vmax = c[0].a;
  for (i = 1; i < n; i++) {
    vmax = max(vmax, c[i].a);
  }

  printf("%lld\n", vmax);

  return 0;
}
```

8. Problema `https://open.kattis.com/problems/canvas`

   (Canvas Painting, Southwestern Europe Regional Contest (SWERC) 2015, `https://open.kattis.com/problems/canvas`)

   Rezolvare:

Listing 8: canvas.c

```
#include <stdio.h>

#define NMAX 100000

typedef struct heap {
          int n;              // numarul de elemente din heap
          long long a[NMAX];  // valorile nodurilor aflate in heap
        } HEAP;

HEAP h; // un min-heap

/*
*  Initializeaza heap-ul vid.
*/
void init(HEAP* h) {
  h->n = 0;
}

/*
*  Se reorganizeaza structura de heap pentru nodul start. Pornind de la
* elementul aflat pe pozitia start, se verifica conditia de min-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta,
* coborandu-se in cadrul subarborelui de radacina start.
*/
void push(HEAP* h, int start, int m) {
  int i, j;
  long long aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (h->a[j] > h->a[j + 1])) {
      j++;
    }

    if (h->a[i] > h->a[j]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

/*
*  Construieste un heap din multimea de elemente 1..n.
*/
void buildheap(HEAP* h) {
  int i;
```

```
  for (i = h->n / 2; i > 0; i--) {
    push(h, i, h->n);
  }
}

/*
*   Extrage din heap nodul aflat pe pozitia 1. Se sterge elementul
* aflat in varful min-heap-ului: pe prima pozitie  este adus ultimul
* element, se micsoreaza numarul de elemente cu 1 si se reorganizeaza
* structura de date a.i. sa se pastreze proprietatea de min-heap.
*/
long long deletemin(HEAP* h) {
  long long k = h->a[1];

  h->a[1] = h->a[h->n];
  h->n--;

  push(h, 1, h->n);

  return k;
}

/*
*  Promoveaza in heap nodul i pentru a se restabili proprietatea
* de min-heap.
*/
void insert(HEAP* h, long long x) {
  int i, j;
  long long aux;

  h->a[++h->n] = x;
  i = h->n;
  j = i >> 1;
  while ((0 < j) && (h->a[j] > h->a[i])) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

    i = j;
    j = i >> 1;
  }
}


int main() {
  int t, n, i;
  long long v1, v2, total;

  scanf("%d", &t);
  while (t--) {
    scanf("%d", &n);

    // se initializeaza min-heap-ul
```

```
    init(&h);
    // se citesc dimensiunile celor n panze
    for (i = 0; i < n; i++) {
      scanf("%lld", &h.a[++h.n]);
    }

    // organizam structura de min-heap
    buildheap(&h);

    total = 0;
    for (i = 0; i < n - 1; i++) {
      // se extrag cele mai mici doua multimi de panze,
      // panzele dintr-o multime avand toate aceeasi culoare
      v1 = deletemin(&h);
      v2 = deletemin(&h);

      // se coloreaza la fel toate panzele din cele 2 multimi
      insert(&h, v1 + v2);

      total += (v1 + v2);
    }

    printf("%lld\n", total);
  }

  return 0;
}
```

9. Problema `https://open.kattis.com/problems/caching`

   (Introspective Caching, Nordic Collegiate Programming Contest (NCPC) 2008, `https://open.kattis.com/problems/caching`)

   Rezolvare:

Listing 9: `caching.c`

```
#include <stdio.h>
#include <string.h>

#define NMAX 100000
#define TMAX 100000
#define MMAX 10001
#define NIL NMAX

typedef struct heap {
          int n;        // numarul de elemente din heap
          int a[MMAX];  // valorile nodurilor aflate in heap
        } HEAP;

int p[TMAX];          // p[i] - obiectul i apare la momentul de timp i
int nextTime[TMAX];   // momentul de timp al urmatoarei aparitii a obiectului
                      // curent
int firstTime[NMAX]; // momentul de timp al primei aparitii a unui obiect
```

```
int lastTime[NMAX];  // momentul de timp al ultimei aparitii a unui obiect
char incache[NMAX];  // daca un obiect se afla in cache sau nu

HEAP h;            // un max-heap
int pos[NMAX];     // pos[k] - pozitia in cache a obiectului k

/*
*  Initializeaza heap-ul vid.
*/
void init(HEAP* h) {
  h->n = 0;
}

/*
*  Se reorganizeaza structura de heap pentru nodul start. Pornind de la
* elementul aflat pe pozitia start, se verifica conditia de max-heap,
* si in cazul neindeplinirii, se cauta o pozitie pentru acesta,
* coborandu-se in cadrul subarborelui de radacina start.
*/
void push(HEAP* h, int start, int m) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (lastTime[h->a[j]] < lastTime[h->a[j + 1]])) {
      j++;
    }

    if (lastTime[h->a[i]] < lastTime[h->a[j]]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      pos[h->a[i]] = i; pos[h->a[j]] = j;

      i = j;
      j = i << 1;
    } else {
       j = m + 1;
    }
  }
}

/*
*  Promoveaza in heap nodul i pentru a se restabili proprietatea
* de max-heap.
*/
void liftup(HEAP* h, int i) {
  int j, aux;

  j = i >> 1;
  while ((0 < j) && (lastTime[h->a[j]] < lastTime[h->a[i]])) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;
```

```
    pos[h->a[i]] = i; pos[h->a[j]] = j;

    i = j;
    j = i >> 1;
  }
}


int main() {
  int c, n, m, i, cachemiss, objectid;

  scanf("%d %d %d", &c, &n, &m);

  // nu este niciun obiect in cache
  memset(incache, 0, sizeof(incache));

  // se initializeaza timpul de inceput si de sfarsit
  // la care poate fi identificat un obiect in datele de intrare
  for (i = 0; i < n; i++) {
    firstTime[i] = lastTime[i] = NIL;
  }

  // momentul de timp unde se poate identifica
  for(i = 0; i < m; i++) {
     nextTime[i] = NIL;
  }

  for (i = 0; i < m; i++) {
    scanf("%d", &p[i]);

    // se adauga momentul de timp i la lista simplu inlantuita
    // reprezentand timpii cand se citeste obiectul p[i]
    if (firstTime[p[i]] == NIL) {
      firstTime[p[i]] = lastTime[p[i]] = i;
    } else {
        nextTime[lastTime[p[i]]] = i;
        lastTime[p[i]] = i;
    }
  }

  // se initializeaza primul moment de timp pentru fiecare obiect i
  for (i = 0; i < n; i++) {
    lastTime[i] = firstTime[i];
  }

  // numarul de ratari
  cachemiss = 0;
  for (i = 0; i < m; i++) {
    // obiectul curent
    objectid = p[i];
    // momentul de timp cand va aparea urmatoarea data obiectul
```

```
    lastTime[objectid] = nextTime[lastTime[objectid]];

    // daca obiectul curent nu se afla in cache
    if (!incache[objectid]) {
      cachemiss++;

      // daca cache-ul nu este plin
      if (h.n < c) {
        // obiectul curent se adauga in cache
        incache[objectid] = 1;

        h.a[++h.n] = objectid;
        pos[objectid] = h.n;

        // se reorganizeaza max-heap dupa timpul urmatoarei
        // aparitii a obiectului
        liftup(&h, h.n);
      } else {
          // se sterge din cache obiectul cel mai indepartat
          // ca moment de aparitie
          incache[h.a[1]] = 0;

          // se adauga in cache obiectul curent
          incache[objectid] = 1;
          h.a[1] = objectid;
          pos[objectid] = 1;

          // se reorganizeaza max-heap-ul
          push(&h, 1, h.n);
      }
    } else {
        // se actualizeaza heap-ul in functie de timpul urmatoarei
        // aparitii a obiectului
        liftup(&h, pos[objectid]);
    }
  }

  printf("%d\n", cachemiss);

  return 0;
}
```

10. Problema `https://open.kattis.com/problems/stockprices`

    (Stock Prices, Northwestern Europe Regional Contest (NWERC) 2010, `https://open.kattis.com/problems/stockprices`)

    Rezolvare:

Listing 10: `stockprices.c`

```
#include <stdio.h>
#include <string.h>
```

```
#define NMAX 1000
#define WLEN 10

typedef struct heap {
          int n;                 // numarul de elemente din heap
          int a[NMAX + 1];       // valorile elementelor aflate in heap
          int (*cmp)(int, int); // pointer catre o functie de comparare
        } HEAP;

HEAP bidHeap;  // un max-heap
HEAP askHeap;  // un min-heap

int vsell[NMAX + 1];
int vbuy[NMAX + 1];

char word[WLEN + 1];


/*
*  Functia intoarce valoarea cea mai mica dintre valorile
* argumentelor functiei.
*/
int min(int a, int b) {
  return ((a < b) ? a : b);
}

/*
*  Functie de comparare: functia intoarce valoarea
*    1 daca valoarea primului argument este mai mica
*      decat valoarea celui de-al doilea argument
*    0 daca valoarea primului argument este mai mare sau egala
*      decat valoarea celui de-al doilea argument
*/
int less(int a, int b) {
  return (a < b);
}

/*
*  Functie de comparare: functia intoarce valoarea
*    1 daca valoarea primului argument este mai mare
*      decat valoarea celui de-al doilea argument
*    0 daca valoarea primului argument este mai mica sau egala
*      decat valoarea celui de-al doilea argument
*/
int greater(int a, int b) {
  return (a > b);
}

/*
*  Initializeaza un max/min-heap vid.
*  @param h - pointer catre o structura de tip HEAP
*  @param fcmp - pointer catre o functie de comparare;
```

```
*               functia intoarce valoarea 1 sau valoarea 0
*/
void init(HEAP* h, int (*fcmp)(int, int)) {
  h->n = 0;
  h->cmp = fcmp;
}

/*
*  Functia verifica daca un heap-ul h este vid. Intoarce valoarea
* 1 daca heap-ul h nu are niciun element sau 0 daca heap-ul contine
* cel putin un element.
*/
int isempty(HEAP* h) {
  return (h->n == 0);
}

/*
*   Functia insereaza un element de valoare v in heap-ul h. Se
* incrementeaza numarul de elemente din heap, se adauga un element de
* valoare v pe ultima pozitie si apoi se reorganizeaza heap-ul.
*  Elementul de pe ultima pozitie va fi promovat in heap atata timp cat
* nu se verifica proprietatea de heap, putand ajunge pana in radacina.
*/
void insert(HEAP* h, int v) {
  int i, j, aux;

  h->a[++h->n] = v;
  i = h->n;
  j = i >> 1;
  while (j > 0) {
    if (h->cmp(h->a[j],h->a[i])) {
      aux = h->a[j]; h->a[j] = h->a[i]; h->a[i] = aux;
      i = j;
      j = i >> 1;
    } else {
        j = 0;
    }
  }
}

/*
*  Se reorganizeaza heap-ul h. Pornind de la elementul aflat pe pozitia 1,
* pozitia radacinii, se cauta o pozitie pentru acesta coborandu-se in cadrul
* arborelui respectiv astfel incat sa se pastreze proprietatea de
* max/min-heap.
*/
void push(HEAP* h) {
  int i, j, aux;

  i = 1;
  j = i << 1;
  while (j <= h->n) {
```

```
    if ((j < h->n) && (h->cmp(h->a[j], h->a[j + 1]))) {
      j++;
    }

    if (h->cmp(h->a[i], h->a[j])) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = h->n + 1;
    }
  }
}

/*
*   Se sterge elementul aflat in varful max/min-heap-ului: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de max/min-heap.
*   Functia returneaza cea mai mare valoare din max-heap (valoarea aflata
* in radacina inainte de stergere) / cea mai mica valoare din min-heap.
*/
int deleteRoot(HEAP* h) {
  int aux = h->a[1];

  h->a[1] = h->a[h->n];
  h->n--;
  push(h);

  return aux;
}


int main() {
  int t, n, i;
  int volume, price, askPrice, bidPrice, stockPrice;

  scanf("%d", &t);

  while(t--) {
    scanf("%d", &n);

    memset(vbuy, 0, sizeof(vbuy));
    memset(vsell, 0, sizeof(vsell));

    init(&bidHeap, less);
    init(&askHeap, greater);
    stockPrice = 0;

    for (i = 0; i < n; i++) {
      scanf("%s", word);
```

```
if (word[0] == 'b') {
  scanf("%d %s %s %d", &volume, word, word, &price);

  vbuy[price] += volume;
  if (vbuy[price] == volume) {
    insert(&bidHeap, price);
  }
} else {
    scanf("%d %s %s %d", &volume, word, word, &price);

    vsell[price] += volume;
    if (vsell[price] == volume) {
      insert(&askHeap, price);
    }
}

while (!isempty(&askHeap) && !isempty(&bidHeap)
       && (bidHeap.a[1] >= askHeap.a[1])) {
  bidPrice = bidHeap.a[1];
  askPrice = askHeap.a[1];
  stockPrice = askPrice;

  volume = min(vsell[askPrice], vbuy[bidPrice]);

  vsell[askPrice] -= volume;
  if (vsell[askPrice] == 0) {
    deleteRoot(&askHeap);
  }

  vbuy[bidPrice] -= volume;
  if (vbuy[bidPrice] == 0) {
    deleteRoot(&bidHeap);
  }
}

if (!isempty(&askHeap)) {
  printf("%d ", askHeap.a[1]);
} else {
    printf("- ");
}

if (!isempty(&bidHeap)) {
  printf("%d ", bidHeap.a[1]);
} else {
    printf("- ");
}

if (stockPrice > 0) {
  printf("%d\n", stockPrice);
} else {
    printf("-\n");
```

```
        }
      }
    }

    return 0;
}
```

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introducere în Algoritmi*, Computer Libris Agora, Cluj-Napoca, 1999.

[2] M. Coşulschi, M. Gabroveanu, *Practica programării în C*, Editura Universitaria, Craiova, 2014.