Задатак: Сортирање бројева

Напиши програм који уређује (сортира) низ бројева неопадајуће (сваки наредни мора да буде већи или једнак од претходног).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5 \cdot 10^4$) а затим и n природних бројева мањих од $2n$, сваки у посебном реду.

Излаз: На стандардни излаз исписати учитане бројеве у сортираном редоследу.

Пример

Улаз	*Излаз*
5	1
3	1
1	3
6	6
8	8
1	

Решење

Сортирање вектора библиотечком функцијом

У језику C++ сортирање је најбоље вршити функцијом `sort`, која прима два итератора - први који указује на почетак дела низа тј. вектора који се сортира, а други указује иза последњег елемента у делу низа који се сортира. Када се сортира цео вектор `a` ти итератори се могу добити помоћу `a.begin()` и `a.end()`.

Сложеност овог приступа је $O(n \log n)$.

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

// sortiranje
sort(a.begin(), a.end());

// ispis rezultata
for (int i = 0; i < n; i++)
  cout << a[i] << endl;
```

Сортирање низа библиотечком функцијом

Када се сортира низ `a` са `n` елемената почетни итератор је `a`, а завршни се може добити као `next(a, n)` или евентуално `a+n`.

Сложеност овог приступа је $O(n \log n)$.

```
// ucitavanje brojeva
int n;
cin >> n;
int a[MAX];
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiranje
sort(a, next(a, n));

// ispis rezultata
for (int i = 0; i < n; i++)
  cout << a[i] << endl;
```

Мехурићи (BubbleSort)

Алгоритам BubbleSort се заснива на идеји да се у једном пролазу кроз низ мењају суседни елементи кад год су у погрешном редоследу. Такви пролази кроз низ се понављају све док се у неком пролазу не утврди да није било погрешно распоређених елемената тј. да је низ већ сортиран. Алгоритам се сигурно зауставља, јер се у првом пролазу осигурава да ће највећи елемент доћи на крај низа, у другом да ће други по величини доћи на крај и тако даље.

У најгорем случају (када је низ у старту сортиран у погрешном редоследу), алгоритам ће вршити n пролаза и укупно $O(n^2)$ поређења и размена, па је сложеност квадратна.

```
void bubble_sort(vector<int>& a) {
  // da li se niz promenio tokom tekuce iteracije
  bool promenjen;
  do {
    // pretpostavljamo da nije
    promenjen = false;
    // poredimo uzastopne elemente niza
    for (int i = 0; i < a.size() - 1; i++)
      // ako su naopako
      if (a[i+1] < a[i]) {
        // razmenjujemo ih
        swap(a[i], a[i+1]);
        // i belezimo da je bilo promena
        promenjen = true;
      }
    // postupak ponavljamo sve dok je bilo promena - ako nije, znaci
    // da su svaka dva uzastopna elementa u dobrom odnosu, tj. da je
    // niz sortiran
  } while (promenjen);
}
```

Једна могућа оптимизација је та да се запамти последња позиција на којој је промена вршена. Пошто се унапред зна да је део низа иза те позиције сортиран (јер није било промена, па су сви међусобно суседни елементи били исправно распоређени), у наредном пролазу се обрађује само део низа пре те последње промене.

```
void bubble_sort(vector<int>& a) {
  // elementi na pozicijama [k, n) su sortirani i potrebno je
  // sortirati jos samo deo niza na pozicijama [0, k]
  int k = a.size() - 1;
  do {
    // pretpostavljamo da su svi elementi na pozicijama [0, k]
    // sortirani
    int kk = 0;

    // poredimo uzastopne elemente dela niza [0, k]
    for (int i = 0; i < k; i++)
      // ako su naopako
      if (a[i] > a[i + 1]) {
        // razmenjujemo ih
        swap(a[i], a[i+1]);
        // ne znamo odnos elemenata na pozicijama i i i-1, pa ne
        // mozemo da tvrdimo da je deo [0, i] sortiran.
        // pretpostavljamo da je bar deo [i, k] sortiran
        kk = i;
      }
    // znamo da je deo na pozicijama [kk, k] sortiran, a znali smo i
    // da je deo na pozicijama [k, n), sortiran, pa je sortiran i deo
    // niza [kk, n). Zato je potrebno sortirati jos samo deo [0, kk] i
    // gornju granicu k postavljamo na kk
    k = kk;
```

```
    // ako je niz [0, k] jednoclan ili prazan, on je sortiran, pa se
    // petlja moze prekinuti
  } while (k > 0);
}
```

Сортирање селекцијом (SelectionSort)

Алгоритам сортирања селекцијом (*SelectionSort*) низ сортира тако што се у првом кораку најмањи елемент доводи на прво место, у наредном кораку се најмањи од преосталих елемената доводи на друго место и тако редом док се низ не сортира.

Најједноставнији начин да се алгоритам реализује је да се у спољној петљи разматрају редом позиције i у низу од прве до претпоследње, да се у унутрашњој петљи разматрају све позиције j од $i+1$ па до последње и да се у телу унутрашње петље размењују елементи на позицији i и j, ако су тренутно у наопаком редоследу.

У најгорем случају и број поређења и број размена је $O(n^2)$.

```
void selection_sort(vector<int>& a) {
  for (int i = 0; i < a.size(); i++) {
    for (int j = i + 1; j < a.size(); j++)
      if (a[i] > a[j])
        swap(a[i], a[j]);
  }
}
```

Бољи начин имплементације је да се за сваку позицију i прво пронађе позиција најмањег елемента у делу низа од позиције i до краја, и да се онда елемент на позицији i размени са пронађеним минимумом (тиме се број размена значајно смањује). Позицију најмањег елемента проналазимо на исти начин као у задатку Редни број максимума.

Нагласимо да је чак и са овом бољом имплементацијом овај алгоритам прилично неефикасан и није га пожељно примењивати осим у случају релативно кратких низова (мада се и они сортирају једноставније библиотечком функцијом).

Алгоритам врши увек $O(n)$ размена и $O(n^2)$ поређења, па је сложеност квадратна.

```
void selection_sort(vector<int>& a) {
  // na svaku poziciju i dovodimo najmanji element iz dela niza na
  // pozicijama [i, n)
  for (int i = 0; i < a.size(); i++) {
    // pozicija najmanjeg elementa u delu [i, n)
    int pmin = i;
    for (int j = i + 1; j < a.size(); j++)
      // ako je element na poziciji j manji od elementa na poziciji
      // pmin, nasli smo novi minimum, pa azuriramo vrednost pmin
      if (a[pmin] > a[j])
        pmin = j;
    // razmenjujemo element na poziciji i sa minimumom
    swap(a[i], a[pmin]);
  }
}
```

Сортирањем уметањем (InsertionSort)

Алгоритам сортирања уметањем (*InsertionSort*) заснива се на томе да се сваки наредни елемент у низу умеће на своје место у сортираном префиксу низа испред њега.

Најједоставнија имплементација је таква да се у спољној петљи разматрају све позиције од друге, па до краја низа и да се уметање врши тако што се сваки елемент размењује са елементима испред себе, све док се испред њега не појави елемент који није мањи од њега или док елемент не стигне на почетак низа.

У најгорем случају је и број поређења и број размена квадратни тј. $O(n^2)$.

```
void insertion_sort(vector<int>& a) {
  // element sa pozicije i umecemo u vec sortirani prefiks na
  // pozicijama [0, i)
  for (int i = 1; i < a.size(); i++) {
    // dok je element na poziciji j manji od njemu prethodnog
    // razmenjujemo ih
    for (int j = i; j > 0 && a[j] < a[j-1]; j--)
      swap(a[j], a[j-1]);
  }
}
```

У бољој имплементацији избегавају се размене током уметања. Елемент који се умеће се памти у привременој променљивој, затим се сви елементи мањи од њега померају за једно место удесно и на крају се запамћени елемент уписује на своје место.

У овом алгоритму се не врше размене, него појединачне доделе. Размена захтева три доделе, па се овом оптимизацијом фаза довођења елемента на своје место може убрзати око три пута. Међутим, у најгорем случају је и број поређења и број додела квадратни тј. $O(n^2)$.

```
void insertion_sort(vector<int>& a) {
  // element sa pozicije i umecemo u vec sortirani prefiks na
  // pozicijama [0, i)
  for (int i = 1; i < a.size(); i++) {
    // pamtimo element na poziciji i
    int tmp = a[i];
    // sve elemente koji su veci od njega pomeramo za
    // jednu poziciju udesno
    int j;
    for (j = i - 1; j >= 0 && a[j] > tmp; j--)
      a[j + 1] =  a[j];
    // stavljamo element sa pozicije i na njegovo mesto
    a[j + 1] = tmp;
  }
}
```

Шелов алгоритам сортирања

Шелов алгоритам сортирања (енгл. Shell sort) представља поправку алгоритма сортирања уметањем где се скраћују дугачке путање малих елемената од краја ка почетку низа. Низ се прво (имплицитно) организује у матрицу димензије $2 \times \frac{n}{2}$ и свака колона се сортира сортирањем уметањем, чиме се постиже да се након враћања матрице у низ сви релативно мали елементи налазе у првој половини низа. Након тога се организује у матрицу димензије $4 \times \frac{n}{4}$ и поступак се понавља.

Прикажимо рад алгоритма на сортирању низа 4, 8, 6, 1, 2, 5, 3, 7.

Прва фаза:

```
4 8 6 1 2 5 3 7  ->  4 8 6 1  ->  2 5 3 1 ->  2 5 3 1 4 8 6 7
                     2 5 3 7      4 8 6 7
```

Друга фаза:

```
2 5 3 1 4 8 6 7  ->  2 5  ->  2 1  ->  2 1 3 5 4 7 6 8
                     3 1      3 5
                     4 8      4 7
                     6 7      6 8
```

Трећа фаза

```
2 1 3 5 4 7 6 8 ->  2  ->  1  ->  1 2 3 4 5 6 7 8
                    1      2
                    3      3
                    5      4
                    4      5
```

```
                  7      6
                  6      7
                  8      8
```

```
void shell_sort(vector<int>& a) {
  // razmaci su redom: n/2, n/4, ..., 1
  for (int razmak = a.size() / 2; razmak > 0;  razmak /= 2)
    // primenjujemo insertion sort algoritam na sve podnizove
    // elemenata ciji su elementi na rastojanju razmak
    for (int i = razmak; i < a.size(); i++) {
      // razmatramo razmaknuti podniz koji se zavrsava na poziciji i
      // svi njegovi elementi osim elementa na poziciji su vec
      // sortirani, pa element na poziciji i umecemo u sortirani
      // prefiks tog razmaknutog podniza
      int tmp = a[i];
      int j;
      for (j = i - razmak; j >= 0 && a[j] > tmp; j -= razmak)
        a[j + razmak] = a[j];
      a[j + razmak] = tmp;
    }
}
```

Сортирање обједињавањем (MergeSort)

Алгоритам сортирања обједињавањем дели низ на два дела чије се дужине разликују највише за 1 (уколико је дужина низа паран број, онда су ова два дела једнаких дужина), рекурзивно сортира сваки од њих и затим обједињује сортиране половине. За обједињавање је неопходно користити додатни, помоћни низ, а на крају се обједињени низ копира у полазни низ. Излаз из рекурзије је случај једночланог низа (случај празног низа не може да наступи осим ако је полазни низ празан).

Кључна операција у овом алгоритму је операција обједињавања сортираних низова, техником два показивача. На пример, обједињавањем сортираних низова `a` и `b` добија се сортирани низ `c`. Два већ сортирана низа могу се објединити у трећи сортирани низ само једним проласком кроз низове (тј. у линеарном времену $O(m+n)$ где су m и n димензије полазних низова).

```
a:  1 3 4 7 9 11                  b:  2 5 8 9 10 12 14
          c: 1 2 3 4 5 7 8 9 9 10 11 12 14
```

Функција сортирања обједињавањем сортира део низа $a[l, d]$, уз коришћење низа tmp као помоћног. Променљива n чува број елемената који се сортирају у оквиру овог рекурзивног позива, а променљива s чува средишњи индекс у низу између l и d. Рекурзивно се сортира $n_1 = \left\lfloor \frac{n}{2} \right\rfloor$ елемената између позиција l и $s-1$ и $n_2 = n - \left\lfloor \frac{n}{2} \right\rfloor$ елемената између позиција s і d. Након тога, сортирани поднизови обједињују се у помоћни низ. Пошто се више не обједињују цели низови, већ делови једног низа, функцију обједињавања морамо мало прилагодити.

Помоћни низ може се пре почетка сортирања алоцирати и користити кроз рекурзивне позиве.

Добијена функција сортирања има гарантовану сложеност најгорег случаја $O(n \log n)$, што значи да је много бржа од функција заснованих на сортирању селекцијом или сортирању уметањем чија је сложеност $O(n^2)$.

```
// ucesljava deo niza a iz intervala pozicija [i, m] i deo niza b iz
// intervala pozicija [j, n] koji su vec sortirani tako da se dobije
// sortiran rezultat koji se smesta u niz c, krenuvsi od pozicije k
void merge(vector<int>& a, int i, int m,
           vector<int>& b, int j, int n,
           vector<int>& c, int k) {
  while (i <= m && j <= n)
    c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
  while (i <= m)
    c[k++] = a[i++];
  while (j <= n)
    c[k++] = b[j++];
```

```
}

// sortira deo niza a iz intervala pozicija [l, d] koristeci
// niz tmp kao pomocni
void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
  // ako je segment [l, d] jednoclan ili prazan, niz je vec sortiran
  if (l < d) {
    // sredina segmenta [l, d]
    int s = l + (d - l) / 2;
    // sortiramo segment [l, s]
    merge_sort(a, l, s, tmp);
    // sortiramo segment [s+1, d]
    merge_sort(a, s+1, d, tmp);

    // ucesljavamo segmente [l, s] i [s+1, d] smestajuci rezultat u
    // niz tmp
    merge(a, l, s, a, s+1, d, tmp, l);
    // vracamo rezultat iz niza tmp nazad u niz a
    for (int i = l; i <= d; i++)
      a[i] = tmp[i];

    // moze i pomocu biblioteckih funkcija
    /*
    merge(next(a.begin(), l), next(a.begin(), s+1),
          next(a.begin(), s+1), next(a.begin(), d+1),
          next(tmp.begin(), l));
    copy(next(tmp.begin(), l), next(tmp.begin(), d+1), next(a.begin(), l));
    */
  }
}

// sortira niz a
void merge_sort(vector<int>& a) {
  // alociramo pomocni niz
  vector<int> tmp(a.size());
  // pozivamo funkciju sortiranja
  merge_sort(a, 0, a.size() - 1, tmp);
}
```

Брзо сортирање (QuickSort)

У сваком кораку алгоритма сортирања један елемент (обично називан *пивот*) се доводи на своје место (пожељно близу средине низа). Да би након тога, проблем могао бити сведен на сортирање два мања подниза, потребно је приликом довођења пивота на своје место груписати све елементе мање или једнаке од њега лево од њега, а све елементе веће од њега десно од њега (ако се низ сортира неопадајуће). То прегруписавање елемената низа, *корак партиционисања* кључни је корак алгоритма брзог сотирања.

Брзо сортирање се може имплементирати на следећи начин. Позив `qsort_(a, l, d)` sortira deo niza $a[l, d]$. Партиционисање се врши техником два показивача.

Након партиционисање рекурзивно се сортирају лева и десна половина низа. Излаз из рекурзије представља случај када је низ (тј. његов део $a[l, d]$) празан или једночлан (такав низ је већ сортиран).

Сложеност најгорег случаја овог алгоритма може бити квадратна тј. $O(n^2)$, ако се стално дешава да пивот дели низ на две неравномерне целине. Ипак, може се доказати да је просечна сложеност овог алгоритма $O(n \log n)$ и у пракси он показује веома добре резултате (за разлику од сортирања обједињавањем не троши се време на померање елемената између помоћног и главног низа).

```
// soritra segment pozicija [l, d] u nizu a
void quick_sort(vector<int>& a, int l, int d) {
  // ako segment [l, d] jedan ili nula elementa on je vec sortiran
```

```
  if (l < d) {
    // za pivot uzimamo proizvoljan element segmenta
    swap(a[l], a[l + rand() % (d - l + 1)]);
    // particionisemo niz tako da se u njemu prvo javljaju elementi
    // manji ili jednaki pivotu, a zatim veci od pivota
    // tokom rada vazi [l, k] su manji ili jednaki pivotu
    // (k, i) su veci od pivota, [i, d] su jos neispitani
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] <= a[l])
        swap(a[i], a[++k]);
    // razmenjujemo pivot sa poslednjim manjim ili jednakim elementom
    swap(a[l], a[k]);
    // rekurzivno sortiramo deo niza levo i desno od pivota
    quick_sort(a, l, k - 1);
    quick_sort(a, k + 1, d);
  }
}

// sortira niz a
void quick_sort(vector<int>& a) {
  // poziv pomocne funkcije koja u nizu a sortira segment pozicija [0, n-1]
  quick_sort(a, 0, a.size() - 1);
}
```

Нерекурзивно брзо сортирање

Једна веома важна употреба стека је за реализацију рекурзије. Током извршавања рекурзивних функција, на стек се смештају вредности локалних промењљивих и аргумената сваког активног позива функције. Рекурзију увек можемо уклонити и уместо системског стека можемо ручно одржавати стек са тим подацима. Прикажимо ову технику уклањања рекурзије на примеру нерекурзивне имплементације алгоритма брзог сортирања. Нагласимо да је код алгоритма QuickSort дубина рекурзије мала (она логаритамски зависи од броја елемената низа), па се њеном елиминацијом не добија ништа значајно.

На стеку ћемо чувати аргументе рекурзивних позива функције сортирања. На почетку је то пар индекса $(0, n1)$. Главна петља се извршава све док се стек не испразни и у њој се обрађује пар индекса који се скида са врха стека. Уместо рекукрзивних позива њихове ћемо аргументе постављати на врх стека и чекати да они буду обрађени у некој од наредних итерација петље. Приметимо да се аргументи другог рекурзивног позива обрађују тек када се у потпуности реши потпроблем који одговара првом рекурзивном позиву, што одговара понашању функције када је заиста имплементирана рекурзивно.

Рецимо и да је ова техника општа и да се рекурзија увек може елиминисати на овај начин. За разлику од тога, елиминисање специфичних облика рекурзије (попут, на пример, репне) није увек применљиво, али када јесте, доводи до боље меморијске (па и временске) ефикасности јер се не користи стек.

```
void quick_sort(ctor<int>& a) {
  // stek na kome čuvamo argumente rekurzivnih poziva
  stack<pair<int, int>> sortirati;
  // sortiranje kreće od obrade celog niza tj. pozicija (0, n-1)
  sortirati.emplace(0, n - 1);
  while (!sortirati.empty()) {
    // skidamo par (l, d) sa vrha steka
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    // obrađujemo par (l, d) na isti način kao u rekurzivnoj implementaciji
    if (d - l < 1)
      continue;
    int k = l;
    for (int i = l+1; i <= d; i++)
```

```
      if (a[i] < a[l])
        swap(a[++k], a[i]);
    swap(a[k], a[l]);
    // umesto rekurzivnih poziva njihove argumente
    // postavljamo na stek
    sortirati.emplace(l, k-1);
    sortirati.emplace(k+1, d);
  }
}
```

Сортирање уз помоћ реда са приоритетом

Сортирање бројева се може извршити коришћењем реда са приоритетом. Користи се алгоритам *сортирања уз помоћ хипа* (енгл. *heap sort*) који је варијација алгоритма сортирања селекцијом (енгл. *selection sort*) у којем се, подсетимо се, у сваком кораку најмањи елемент доводи на почетак низа. Алгоритам хип сорт користи чињеницу да је одређивање и уклањање најмањег елемента из реда са приоритетом прилично ефикасна операција. Стога се сортирање може реализовати тако што се сви елементи уметну у ред са приоритетом, из кога се затим проналази и уклања један по један најмањи елемент.

И убацивање елемената у ред са приоритетом и избацивање елемената из реда са приоритетом обично је сложености $O(\log k)$, где је k број елемената у реду са приоритетом). Стога је укупна сложеност овог алгоритма сортирања $O(n \log n)$.

```
// ovo je način da se u C++-u definiše red sa prioritetom u kome su
// elementi poređani u opadajućem redosledu prioriteta (ovde, vrednosti)
priority_queue<int, vector<int>, greater<int>> Q;
// učitavamo sve elemente niza i ubacujemo ih u red
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int ai;
  cin >> ai;
  Q.push(ai);
}
// vadimo jedan po jedan element iz reda i ispisujemo ga
while (!Q.empty()) {
  cout << Q.top() << endl;
  Q.pop();
}
```

Ред са приоритетом може бити реализован и помоћу ручно имплементираног хипа.

```
// svi elementu u nizu a na pozicijama [0, i) zadovoljavaju uslov hipa
// pomera se element na poziciji i navise tako da nakon toga svi elementi
// na pozicijama [0, i] zadovoljavaju uslov hipa
void pomeri_gore(vector<int>& a, int i) {
  while (i > 0) {
    // roditelj cvora na poziciji i
    int roditelj = (i - 1) / 2;
    // ako element nije veci od svog roditelja, postupak je zavrsen
    if (a[i] <= a[roditelj])
      break;
    // inace razmenjujemo element sa svojim roditeljem
    swap(a[i], a[roditelj]);
    // i postupak nastavljamo od pozicije roditelja
    i = roditelj;
  }
}


// svi elementi u nizu a na pozicijama od 0 do n-1 zadovoljavaju uslov
```

```
// hipa osim eventualno elementa na poziciji i koji moze biti manji od
// svojih potomaka (ali je svakako manji ili jednak svom roditelju,
// ako roditelj postoji) - element se pomera nanize, tako da se na
// pozicijama 0 do n-1 dobije ispravan hip
void pomeri_dole(vector<int>& a, int i, int n) {
  while (true) {
    // pozicija sa kojom treba zameniti element na poziciji i
    // vrednost i govori da element ne treba menjati
    int menjam = i;
    // pozicija levog i desnog potomka cvora i
    int levi = 2 * i + 1;
    int desni = 2 * i + 2;
    // ako element na poziciji i ima levog potomka i ako je manji od njega
    // trebalo bi da se zameni sa levim
    if (levi < n && a[i] < a[levi])
      menjam = levi;
    // medjutim, ako postoji i desni potomak i ako je on jos manji
    // trebalo bi da se zameni sa njim
    if (desni < n && a[menjam] < a[desni])
      menjam = desni;

    // ako je menjam ostalo na vrednosti i znaci da nije manje ni od
    // jednog potomka i posao je zavrsen
    if (menjam == i)
      break;

    // u suprotnom menjamo element sa svojim potomkom
    swap(a[i], a[menjam]);
    // i nastavljamo postupak popravke od tog potomka
    i = menjam;
  }
}

// od niza pravi hip
void napravi_hip(vector<int>& a) {
  // za sve pozicije od 1 do a.size()-1
  for (int i = 1; i < a.size(); i++)
    // na pozicijama [0, i) je vec napravljen hip
    // element na poziciji i jedini ne mora da zadovoljava uslov
    // pa ga pomeramo navise tako da dobijemo hip na pozicijama [0, i]
    pomeri_gore(a, i);
}

// od niza u kojem se nalazi hip pravi sortirani niz
void sortiraj_hip(vector<int>& a) {
  // za sve pozicije od a.size()-1 unazad do 1
  for (int k = a.size() - 1; k > 0; k--) {
    // najveci element u hipu razmenjujemo sa elementom na poziciji n
    swap(a[0], a[k]);
    // element na vrhu ne mora biti na svom mestu u hipu koji ima k
    // elemenata pa ga pomeramo na dole tako da tih prvih k elemenata
    // cine ispravan hip
    pomeri_dole(a, 0, k);
  }
}

void heap_sort(vector<int>& a) {
  // Od niza pravi hip - binarno drvo u cijem se svakom cvoru nalazi
```

```
  // element koji je veci ili jednak od svojih potomaka. Drvo je u niz
  // smesteno po nivoima. Na primer:
  //        13
  //    8        10           13 8 10 4 2 5
  // 4     2   5
  napravi_hip(a);
  // od niza u kojem se nalazi hip pravi sortirani niz
  sortiraj_hip(a);

  // moze i uz pomoc biblioteckih funkcija
  // make_heap(a.begin(), a.end());
  // sort_heap(a.begin(), a.end());
}
```

Сортирање уз помоћ мултискупа

Сортирање се може извршити коришћењем мултискупа. Користи се алгоритам *сортирања уметањем* (енгл. *insertion sort*), једино што уместо уметања елемената у низ врши уметање елемената у мултискуп. На крају се испишу сви елементи мултискупа редом (они се чувају у сортираном редоследу). Пошто се мултискуп имплементира коришћењем сортираног бинарног дрвета, овај се алгоритам назива и *сортирањем коришћењем дрвета* (енгл. *tree sort*).

У језику C++ можемо користити библиотечку колекцију `multiset`.

Пошто се уметање у мултиску извршава у сложености $O(\log k)$, где је k број елемената у мултискупу, а испис у времену $O(k)$, сложеност овог поступка сортирања је $O(n \log n)$. Заузеће меморије је $O(n)$, уз, додуше, мало већи константни фактор него када се елементи користе у низ. Додатно, елементи нису поређани један уз други у меморији, што мало успорава приступ.

```
int n;
cin >> n;
multiset<int> a;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  a.insert(x);
}
for (int x : a)
  cout << x << endl;
```

Види другачија решења овог задатка.