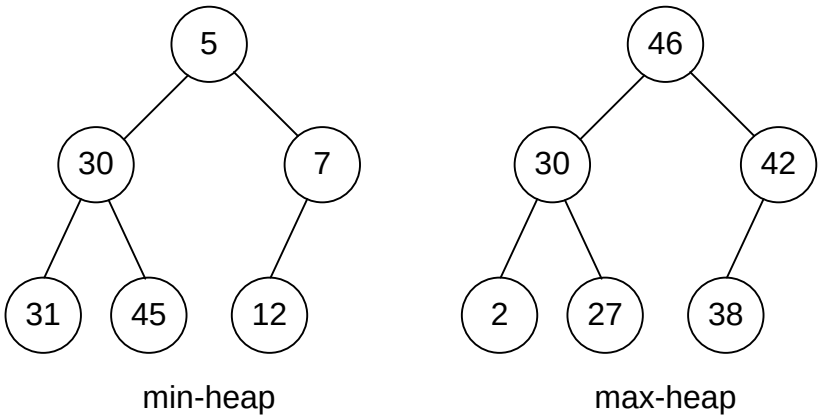
Heap

Min-heap is an almost complete binary tree where every node has a value *smaller* than the values of its children.

Max-heap is an almost complete binary tree where every node has a value *bigger* than the values of its children.



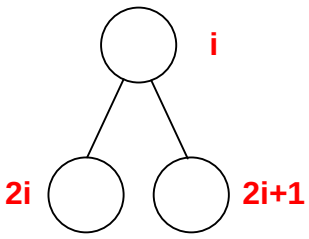
When inserting elements, the tree vertices are filled from left to right: if some vertex does not have a left child, then it does not have a right one either. If the tree has a vertex at level *h*, then level *h* – 1 is completely filled.

The heap is simulated with array m with certain ordering properties. Each node of the tree corresponds to an element of the array. The parent of the vertex with index *i* is the vertex $\lfloor i/2 \rfloor$ (vertex m[1] is the root). The left and right children of vertex *i* are vertices 2*i* and 2*i* + 1, respectively. The heap may not occupy the entire array, so we will store its size *size*. The heap consists of elements m[1], m[2], ..., m[*size*]. The movement along the tree is carried out by the following operations:

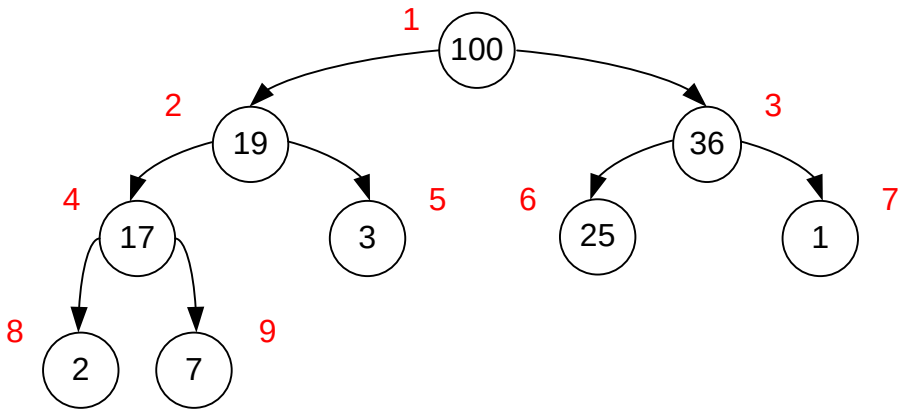
```
Parent(i)
  return ⌊i/2⌋;

Left(i)
  return 2*i;

Right(i)
  return 2*i+1;
```



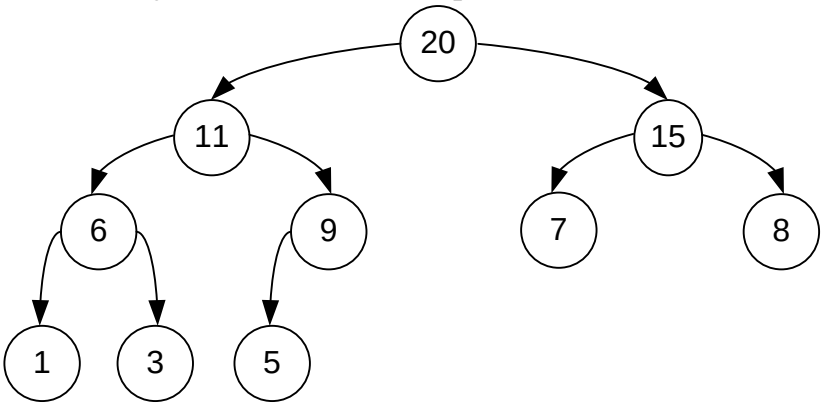
The main property of the heap is that the value of the child does not exceed the value of the parent: m[parent(*i*)] ≥ m[*i*] (in the case of *max-heap*). Thus, the largest element of the tree (subtree) is at the root of the tree (subtree).



i	1	2	3	4	5	6	7	8	9
m[i]	100	19	36	17	3	25	1	2	7

Size of the heap *size* = 9.
Left(2) = 4, Right(2) = 5. Left(3) = 6, Right(3) = 7.
Parent(9) = 4, Parent(4) = 2, Parent(2) = 1.

Exercise. Create an array m for the next heap:



E-OLYMP **3737. Is it a Heap?** The **Heap** data structure can be implement using an array.

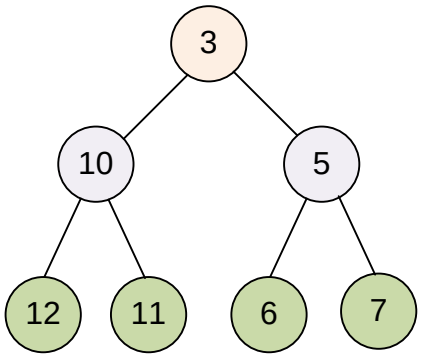
The array must maintain the main **Heap** property: for each i ($1 \le i \le n$) next conditions must hold:

- If $2i \le n$, then a[i] ≤ a[$2i$]
- If $2i + 1 \le n$, then a[i] ≤ a[$2i + 1$]

The array of integers is given. Determine whether it is a Heap.

► For each index i of the input array, the heap condition must be checked. It is enough to iterate over the value of i from 1 to n / 2.

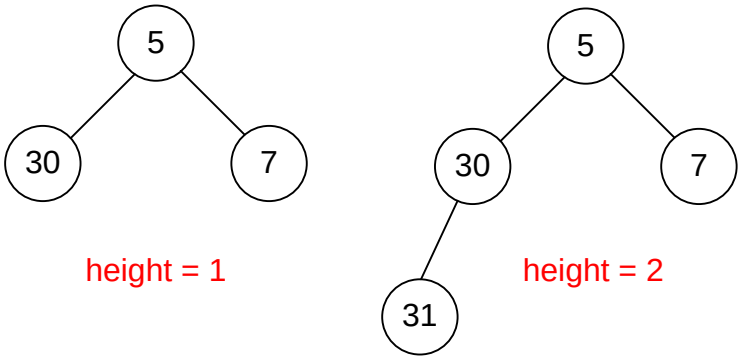
The heap given in the sample, has the form:



The **height** of the vertex of the tree is the height of the subtree rooted at this vertex – the number of ***edges*** in the *longest path* starting at this vertex down the tree to the leaf. The ***height of the tree*** is the same as the height of its root.

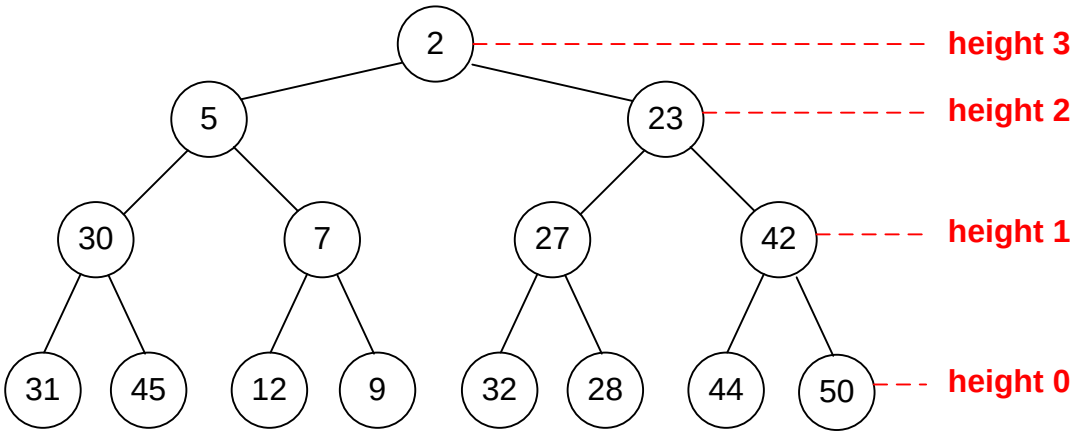
Statement. Heap with n elements has a height $\lfloor \log_2 n \rfloor$.

Example. The height of the heap with 3 elements is $\lfloor \log_2 3 \rfloor = 1$. If the heap contains 4 elements, its height is $\lfloor \log_2 4 \rfloor = 2$.

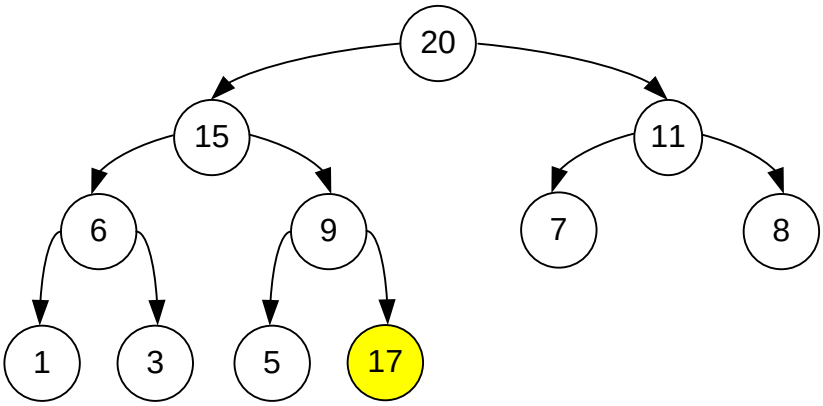


Statement. Number of vertices with height h in the heap with n elements do not exceed $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

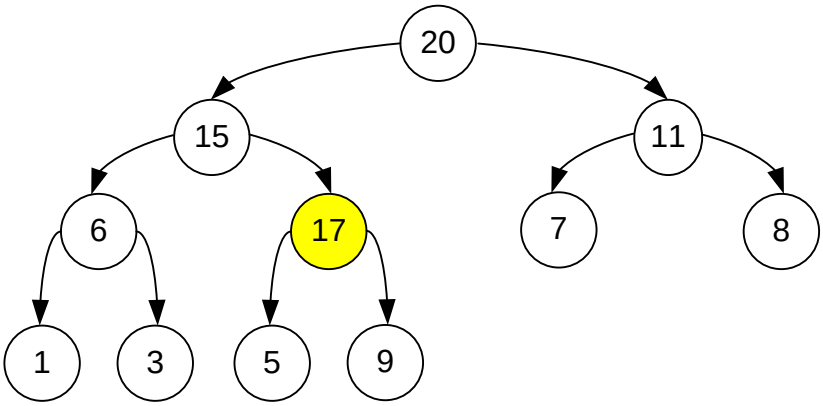
Example. Let heap contains $n = 15$ elements (complete binary tree with height 3). Then number of vertices with height 3 equals to $\lceil n/16 \rceil = 1$ (the only root of the tree). Number of vertices with height 2 equals to $\lceil n/8 \rceil = 2$ (two sons of the root). $\lceil n/4 \rceil = 4$ vertices hace the height 1. Number of leaves (vertices of height 0) equals to $\lceil n/2 \rceil = 8$.



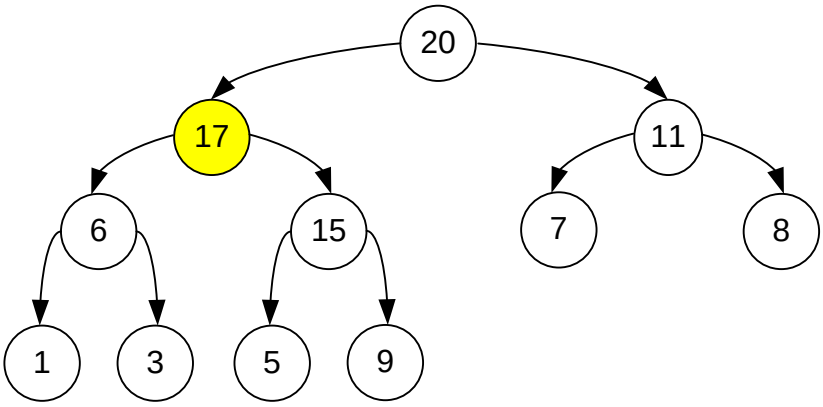
Insert an item to the heap. The new element is added to the last position in the array, that is, the position with the index *size* + 1 (and increase *size* by 1). This can violate the main property of the heap, since the new element can be larger than the parent. In this case, you should "*raise*" the new element one level (change with the parent node) until the main property of the heap is observed. For example, let's insert 17 into the heap.



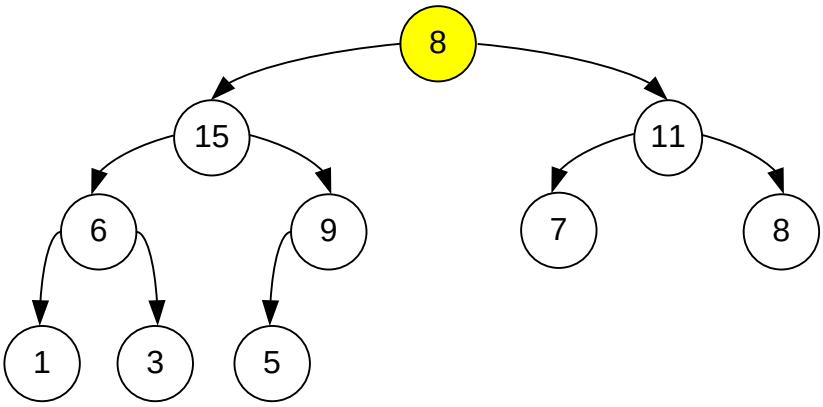
Swap the elements 17 and 9.



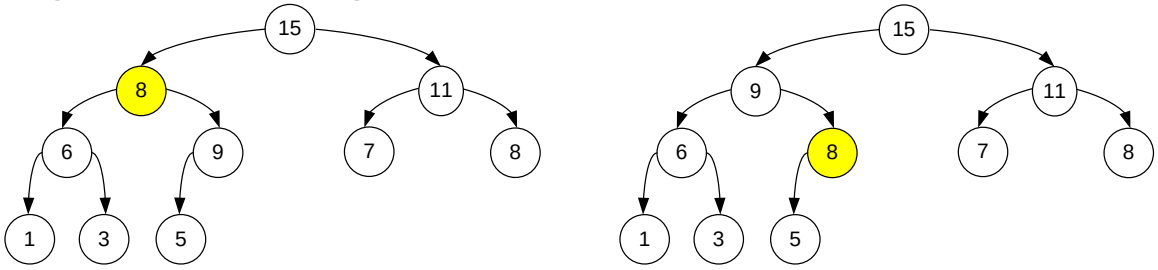
Swap the elements 15 and 17.



Heap ordering. During the operations with an already built binary heap, the main property of the heap can also be violated: a vertex can become smaller than its child.



The **heapify** method restores the main heap property for the tree rooted at the *i*-th node, if both subtrees satisfy it. To do this, it is necessary to "*drop*" the *i*-th vertex (swap with the largest of the descendants) until the main property is restored (the process ends when there is no descendant larger than its parent). It is easy to see that the complexity of this algorithm is also $O(\log_2 n)$.



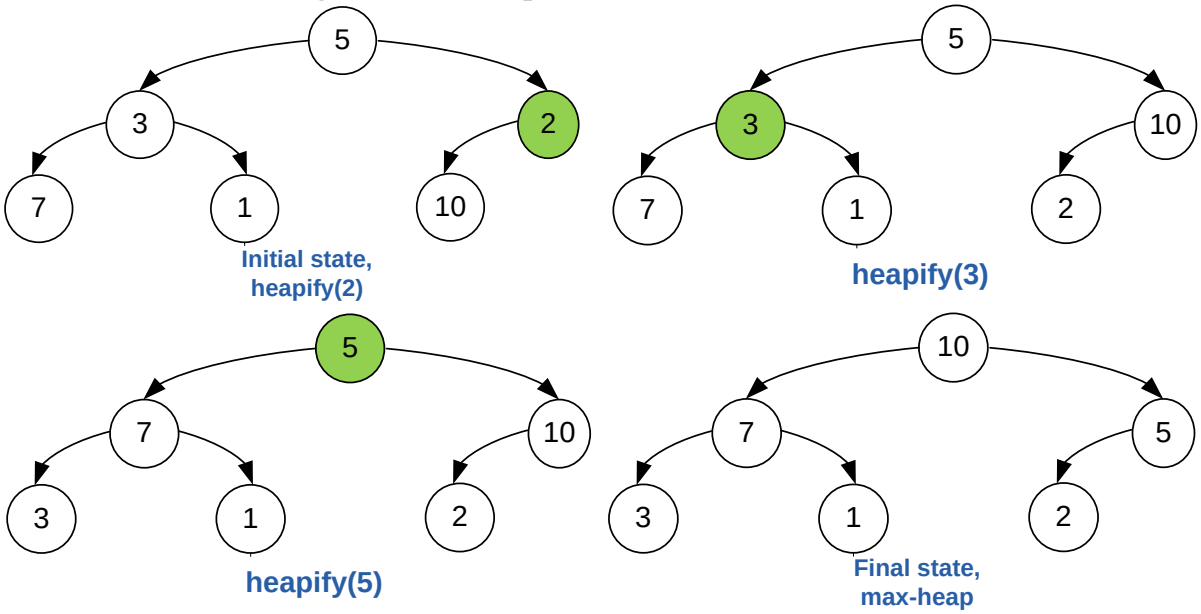
Let A be an array and *i* and index in this array. When **MAX-HEAPIFY** is called, it is assumed that the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but that A[*i*] may be smaller than its children, thus violating the max-heap property. The function of **MAX-HEAPIFY** is to let the value at A[*i*] "float down" in the maxheap so that the subtree rooted at index *i* becomes a **max-heap**.

MAX-HEAPIFY(A, *i*)
l ← LEFT(*i*)
r ← RIGHT(*i*)
if *l* ≤ heap-size[A] and A[*l*] > A[*i*] ***then*** *largest* ← *l*
else *largest* ← *i*
if *r* ≤ heap-size[A] and A[*r*] > A[*largest*] ***then*** *largest* ← *r*
if *largest* ≠ *i*
then **exchange** A[*i*] ↔ A[*largest*]
MAX-HEAPIFY(A, *largest*)

Build a Heap. Let an array m[1..*n*] be given, which must be turned into a heap by rearranging its elements. To do this, you can apply the *heapify* operation to all vertices, starting from the bottom. However, the vertices numbered $\lfloor n/2 \rfloor+1$, ..., *n* are leaves, and the subtrees with these vertices already satisfy the main property of the heap. For each of the remaining vertices, in descending order of indices, use the *heapify* procedure.

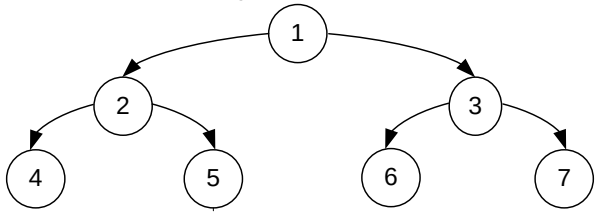
BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for *i* ← length[A] / 2 ***downto*** 1
do **MAX-HEAPIFY**(A, *i*)

Example. Let m = {5, 3, 2, 7, 1, 10}, *n* = 6. The structure of the tree that corresponts to initial state, is given below. Tree has three leaves (vertices 7, 1 and 10) that already satisfy the main property of the heap. We shall run the *heapify* procedure for vertices 2, 3 and 5 to get a max-heap.



Final state of array is m = {10, 7, 5, 3, 1, 2}.

Exercise. Let m = {1, 2, 3, 4, 5, 6, 7}, *n* = 7. Simulate the heapify process to get **max-heap**. Output the final state of array.



The running time of the procedure **Build a Heap** does not exceed O($n\log_2 n$). However, this approximation is not accurate. The point is that the running time of the *heapify* procedure depends on the height of the vertex for which it is called (and is proportional to this height).

Since the number of vertices of height *h* in a heap of *n* elements does not exceed $\lceil n/2^{h+1} \rceil$, and the height of the entire heap does not exceed $\lfloor \log_2 n \rfloor$, time complexity for building the heap does not exceed

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) \le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

To estimate the last sum, we will use the following formulas:

1. The sum of an infinite geometric progression: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$.

2. Let us differentiate the above formula and multiply it by x: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$.

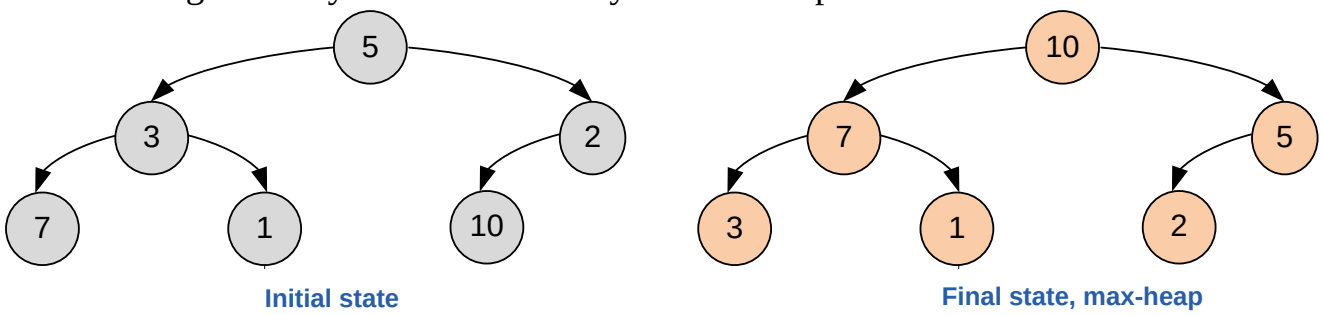
3. Substitute to the last formula $x = ½$: $\sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{1/2}{(1-1/2)^2} = 2$.

Therefore $O\left(n\sum_{h=0}^{\infty} \frac{h}{2^h}\right)$ = O(n), so the time to build the heap is proportional to the number of its elements.

E-OLYMP **10166. Max-heap** Given array of integers. Rearrange its elements so that to get a max-heap.

► Read the input data into array a. For all indexes of array from n / 2 downto 1 execute the procedure ***heapify***. Array turns into **max-heap**.

The original array and the final array that is a heap are shown below.



Declare the array, which later will be converted into a heap.

```
#define MAX 1001
int a[MAX];
```

Function ***left*** returns the index of the left sun.

```
int left(int i)
{
  return 2 * i;
}
```

Function ***right*** returns the index of the right sun.

```
int right(int i)
{
  return 2 * i + 1;
}
```

Function ***swap*** swaps the elements i and j.

```
void swap(int &i, int &j)
{
  int temp = i;  i = j; j = temp;
}
```

Function ***heapify*** restores the heap property for the tree with the root in the *i*-th vertex.

```
void heapify(int a[], int i, int n) // n = size of heap
{
  int largest = 0;
  int l = left(i);
  int r = right(i);
```

We are looking for the index of the maximum element among the current a[*i*] and its sons a[*l*] and a[*r*].

```
  if (l <= n && a[l] > a[i]) largest = l;
  else largest = i;
  if (r <= n && a[r] > a[largest]) largest = r;
```

If a[*i*] is not the maximum element, then we change it with the maximum one and then recursively restore the heap property in the left or right subtree.

```
  if (largest != i)
  {
    swap(a[i], a[largest]);
    heapify(a, largest, n);
  }
}
```

The main part of the program. Read the input array starting from index 1.

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

For all indexes from *n* / 2 to 1 execute the ***heapify*** procedure.

```
for (i = n / 2; i > 0; i--)
  heapify(a, i, n);
```

Print the resulting array, which is the max-heap.

```
for (i = 1; i <= n; i++)
  printf("%d ", a[i]);
printf("\n");
```

Heapsort algorithm

The heapsort algorithm starts by using **BUILD-MAX-HEAP** to build a max-heap on the input array A[1 . . *n*]. Since the maximum element of the array is stored at the root A[1], it can be put into its correct final position by exchanging it with A[*n*]. If we now “discard” node *n* from the heap (by decrementing heap-size[A]), we observe that A[1 . . (*n* − 1)] can easily be made into a **max-heap**. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is

needed to restore the max-heap property, however, is one call to **MAX-HEAPIFY**(A, 1), which leaves a max-heap in A[1 . . (n − 1)]. The heapsort algorithm then repeats this process for the maxheap of size n − 1 down to a heap of size 2.

HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i ← length[A] ***downto*** 2
   do exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] − 1
      MAX-HEAPIFY(A, 1)

E-OLYMP **2321. Sorting** Sort array of integers in nondecreasing order.
► Use **heapsort** to sort an array.

```
void BuildHeap(int a[], int n)
{
  for (int i = n / 2; i > 0; i--)
    heapify(a, i, n);
}

void HeapSort(int a[], int n)
{
  BuildHeap(a, n);
  for (int i = n; i >= 2; i--)
  {
    swap(a[1], a[i]);
    heapify(a, 1, i - 1);
  }
}
```

E-OLYMP **972. Sorting time** Sort the time according to specified criteria.
► Use **heapsort** to sort the time structures.

Declare structure **MyTime**.

```
struct MyTime
{
  int hour, min, sec;
  MyTime() {};
  MyTime(MyTime &a) : hour(a.hour), min(a.min), sec(a.sec) {};
};
```

Declare the comparator.

```
int f(MyTime a, MyTime b)
{
  if ((a.hour == b.hour) && (a.min == b.min)) return a.sec < b.sec;
  if (a.hour == b.hour) return a.min < b.min;
  return a.hour < b.hour;
}
```

Read the input data into array of **MyTime** sturctures.

```
#define MAX 1001
MyTime lst[MAX];
```

Call **Heapsort** to sort the data.

```
HeapSort(lst, n);
```

HEAPSORT with STL
Read the input data.

```
scanf("%d",&n);
v.resize(n);
for(i = 0; i < n; i++)
  scanf("%d",&v[i]);
```

Transform the array of numbers into the heap.

```
make_heap(v.begin(),v.end());
```

Delete the maximum element from the heap. Function ***pop_heap*** swaps the first and last elements, decrease the size of the heap by 1 and restores the heap property.

```
for(i = v.size(); i > 0; i--)
  pop_heap(v.begin(),v.begin() + i);
```

Print the sorted array.

```
for(i = 0; i < v.size(); i++)
  printf(" %d",v[i]);
printf("\n");
```

E-OLYMP **1953. The results of the olympiad** *n* Olympiad participants have unique numbers from 1 to *n*. As a result of solving problems at the Olympiad, each participant received a score (an integer from 0 to 600). It is known how many points everybody scored.

Print the list of participants in Olympiad in decreasing order of their accumulated points.

► Use **heapsort** to sort the *Member* (participant) structures. Each participant has his own *id* and *score*.

```
struct Member
{
  int id, score;
  Member(int id = 0, int score = 0) : id(id), score(score) {};
};
```

E-OLYMP **8236. Sort evens and odds** Sequence of integers is given. Sort the given sequence so that first the odd numbers are arranged in ascending order, and then the even numbers are arranged in descending order.

► Use **heapsort** to sort integers. Sort the numbers according to the following comparator f(***int*** *a*, ***int*** *b*):

```
int f(int a, int b)
{
```

If *a* and *b* have different parity, then even numbers must come after odd numbers.

```
  if (abs(a % 2) != abs(b % 2)) return abs(a % 2) > abs(b % 2);
```

If *a* and *b* are even, then sort them in in decreasing order.

```
  if (a % 2 == 0) return a > b;
```

If *a* and *b* are odd, then sort them in in increasing order.

```
  if (abs(a % 2) == 1) return a < b;
}
```

Note that the input numbers can be positive and negative.

E-OLYMP **8637. Sort the points** The coordinattes of *n* points are given on a plane. Print them in increasing order of sum of coordinates. In the case of equal sum of point coordinates sort the points in increasing order of abscissa.

► Use **heapsort** to sort the *Point* structures. Each point has *x* and *y* coordinate.

```
struct Point
{
  int x, y;
  Point(int x, int y) : x(x), y(y) {}
};
```

A **priority queue** is a container adaptor that provides constant time lookup of the ***largest*** (by default) element, at the expense of logarithmic insertion and extraction.

Working with a **priority_queue** is similar to managing a heap, with the benefit of not being able to accidentally invalidate the heap.

Example of **max – heap**:

```
#include <cstdio>
#include <queue>
using namespace std;

priority_queue<int> pq;

int main(void)
{
  pq.push(5); pq.push(3); pq.push(2);
  pq.push(7); pq.push(1); pq.push(10);
  printf("Size = %d, Max element = %d\n", pq.size(), pq.top());
  pq.pop();
  printf("Size = %d, Max element = %d\n", pq.size(), pq.top());
```

```
  return 0;
}
```

To change **max-heap** to **min-heap** just change the declaration:

```
priority_queue<int,vector<int>,greater<int> > pq;
```

By *default* Java priority queue is a **min-heap**.

```
import java.util.*;

public class Main
{
  public static void main(String[] args)
  {
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
    pq.add(5); pq.add(3); pq.add(2);
    pq.add(7); pq.add(1); pq.add(10);
    System.out.println("Size = " + pq.size() + " Min element = " +
pq.peek());
    pq.poll();
    System.out.println("Size = " + pq.size() + " Min element = " +
pq.peek());
  }
}
```

To change **min-heap** to **max-heap** use the declaration:

```
PriorityQueue<Integer> pq =
  new PriorityQueue<Integer>(Collections.reverseOrder());
```

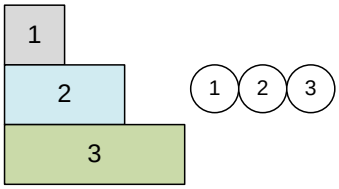
E-OLYMP **3004. Queue** *k* ticket offices are working at the train station, but the queue to them is just one. Initially, when all the ticket offices are free, the first *k* people from the queue go to the offices. The others are waiting their turn. As soon as someone is served and the corresponding office becomes free, the next person in the queue comes to this office. This continues until all the people will be served. You are given the time t_i to serve the *i*-th client in the queue. Simulate the process of selling tickets.

► Lets simulate the process of selling tickets using the multiset *s*. Bring the first *k* people to free cash desks and store their service time in the multiset. During further processing, the multiset will contain *k* elements. Each value in multiset reflects the time moment when the corresponding cash register will become free and the next person will be able to approach it. Obviously, each time a new person must come to the cash register for which this time is minimal. The time of the last served client will be the desired one.

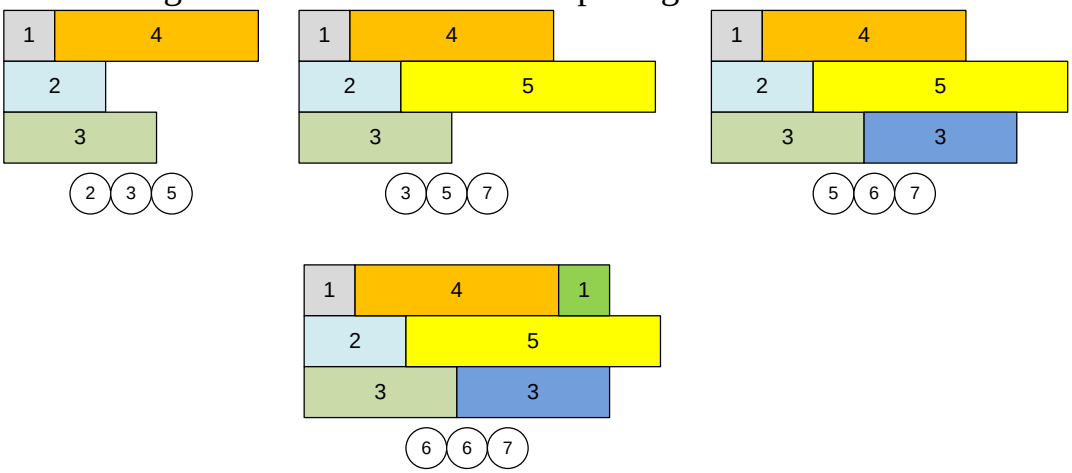
Let we have $n = 7$ people and $k = 3$ ticket offices. The time to serve the clients is

1 2 3 4 5 3 1

Put to the heap first $k = 3$ elements.



Then, at each iteration, we take the next element (the next person from the queue) and put it instead of the smallest one (we put the person to the ticket office that will be released earlier). We put to the queue the time at which this new person leaves the ticket office. Next to each figure the numbers in the heap are given.



Create a heap, which root contains the smallest element.

```
priority_queue<long long, vector<long long>, greater<long long> > pq;
```

Read the input data. Put the service time of the first *k* people into the queue *pq*.

```
scanf("%d %d",&n,&k);
for(i = 0; i < n; i++)
{
  scanf("%d",&ti);
  if (pq.size() != k) pq.push(ti);
  else
  {
```

Find the person who leaves the ticket office first and put the next person from the queue behind him.

```
    long long temp = pq.top(); pq.pop();
    pq.push(temp + ti);
  }
}
```

The largest number in queue *pq* equals to the time when the last client will be served. It will be the answer.

```
while(pq.size() > 1) pq.pop();
printf("%lld\n",pq.top());
```

E-OLYMP **3841. Dr Who's Banquet** Dr. Who is organizing a banquet, and will be inviting several guests. A guest is happy, if he can chat with a fixed number of other

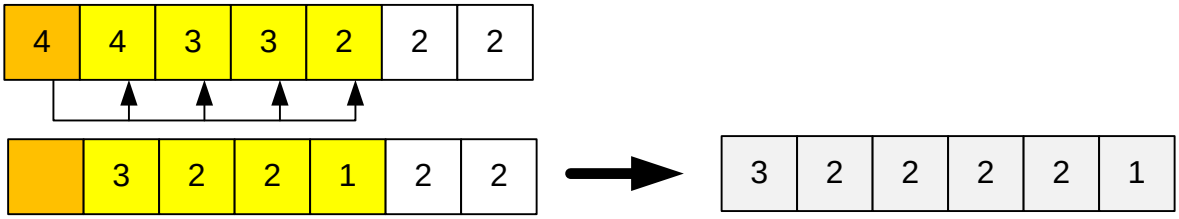
guests. We assume that guests cannot talk to themselves. Help Dr. Who make all his guests happy, if possible, by organizing chats between guests.

► Put the input numbers into the **max-priority** queue. Let partner *v* be at the top of the queue. Then assign for him the interlocutors who would like to get the largest number of partners for communication. We get a greedy algorithm of establishing the correspondences for communication. If all partners can be satisfied this algorithm, then we output **ok**. Otherwise, the output is **fail**.

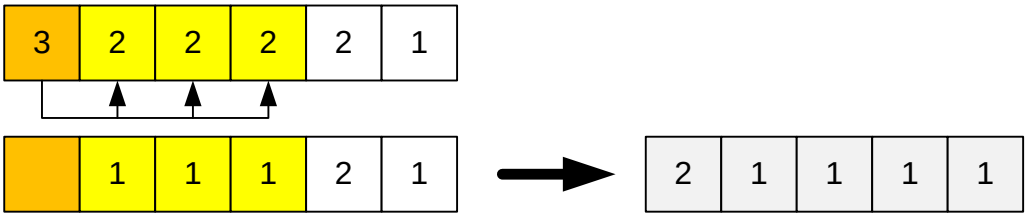
Consider the test case with input

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

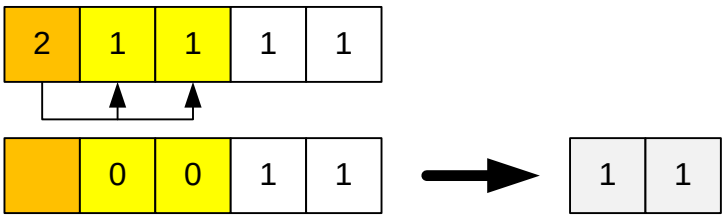
There are 7 guests in total. For communication with a person who wants to have 4 interlocutors, we will assign those who want to have 4, 3, 3 and 2 interlocutors, respectively.



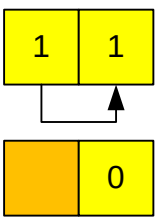
For communication with a person who wants to have 3 more interlocutors, we will assign those who want to have 2, 2 and 2 interlocutors, respectively.

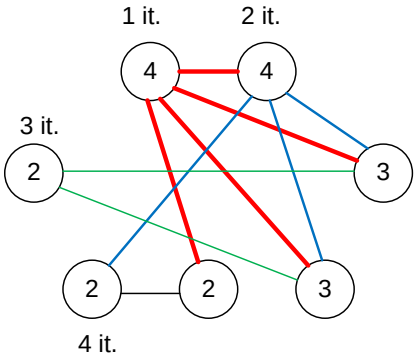


Third iteration:



Fourth (last) iteration:





1 it. = 1-st iteration