# Heap Ltd.

Evgeniy Gabrilovich gabr@cs.technion.ac.il
Alex Gontmakher gsasha@cs.technion.ac.il

## Abstract

Programmers frequently need to select a number of best elements from a sequence of values. This problem is not even nearly new, and several algorithms have been developed to address it, each having different time complexity characteristics. We analyze the various existing algorithms, and then present another one called *Limited Heap*. This algorithm provides arguably the best tradeoff between speed and memory utilization. The algorithm is frequently useful (in fact, both authors have independently used it in different projects), but is surprisingly little known to the wide public. The implementation of the proposed idea is very simple, and expressed in terms of conventional heap operations only takes several dozens lines of code.

## Introduction

When you perform a search on a Web site, you usually want to see the most relevant results first. To do this, the search engine assigns each search result a relevance score, and ultimately returns 10 or 20 highest-scoring results.

Selecting several best elements is obviously not limited to the Web. For instance, when monitoring system health, you might want to find the ten biggest files on a given disk. In the realm of genetic algorithms, during each phase you choose a certain number of the fittest organisms to form the next generation.

While finding a single best element in a sequence is straightforward, selecting $k$ best elements is a tricky business. There are several solutions, with different performance characteristics. The trade-off between time and space is not trivial, and the most practical algorithm must be selected very carefully.

What is the best that you can expect from a good solution? On the one hand, each element must be examined at least once, so the time complexity can be no lower than $O(N)$. On the other hand, the resulting $k$ best elements need to be stored, hence the memory complexity can be no better than $O(k)$. The problem is, however, that you cannot have both time and memory complexity low at the same time.

**Solution #1: Sorting**

A naïve way to select the $k$ largest elements is simply to store the entire sequence in memory, sort it in the decreasing order, and then return the first $k$ elements from the sorted sequence. Coding this technique is trivial since a sorting algorithm is a part of the standard library in virtually any language.

The time complexity, however, is mediocre – $O(N \cdot log(N))$. In addition, storing the entire sequence consumes $O(N)$ additional memory. When $N$ is very large (and, especially, when the elements are produced on-the-fly and need not be permanently stored otherwise), this technique may impose an unreasonable burden on memory usage.

**Solution #2: Heap sort**

An apparently better way to achieve our aim is to use a variant of heap sort. The original heap sort algorithm [1] collects all the elements in an array, rearranges the array as a heap (this can be accomplished in $O(N)$ in the worst case), and then extracts the largest element from the heap $N$ times (this amounts to $O(N \cdot log(N))$ since the heap property needs to be restored after

each extraction). In our case, we only need to perform $k$ extractions to obtain the $k$ largest elements, hence the overall time complexity is $O(N + k \cdot log(N))$.

Note that this solution still suffers from the same drawback as the previous one, as it requires $O(N)$ additional memory to operate the heap that initially contains the entire sequence.

**Solution #3: Limited heap**

When the number of elements in the sequence $(N)$ is huge, we'd rather not store them all merely for selecting the $k$ largest ones. Let us see how we can minimize the memory requirements, using only $O(k)$ additional memory to store the $k$ elements requested.

To achieve this aim, we use a *limited* heap, which cannot grow beyond $k$ elements. We "sift" the entire sequence through it, while at any given moment the heap stores the $k$ largest elements seen so far. New elements are only inserted if they are larger than the current smallest element in the heap, in which case they replace the latter, and the heap size never grows beyond $k$.

Ideally, we would like to sift all the sequence elements through the heap one by one, removing the worst element whenever the heap size becomes larger than $k$. However, while the heap provides easy access to its best element, removing the worst element is more complex and can require as much as $O(k)$ operations. To circumvent this problem, we observe that during the selection process we do not need access to the best element, but only to the worst. Thus, we *reverse the heap order*, so that the heap root always contains the current smallest element. In fact, we maintain a "min-heap", even though what we really want is to select the $k$ largest elements.

**Complexity analysis:** At the steady state, when the heap contains $k$ items, determining the value of the smallest one takes $O(1)$ (due to the "min-heap" ordering). Whenever applicable, replacing the smallest element takes $O(log(k))$, thus the overall worst-case time complexity is $O(N \cdot log(k))$. Observe that since the heap is *limited*, it only keeps as many elements as are eventually required, hence the additional memory complexity is only $O(k)$ – a substantial saving when $k << N$ !

Implementation of this algorithm is straightforward. Reverting the order of elements in the heap means you just need to override the comparison operation it uses. In C++, the comparison predicate is a template parameter to the heap operations, so the algorithm can be readily implemented using the heap manipulation functions from the C++ standard library [2].

Listing 1 presents a fragment of the C++ code of limited heap[1] (template class KMaxValues). The class inherits from std::vector, and thus capitalizes on its container functionality. To make the template more generic, we allow elements to carry a "payload" in addition to their values (eliminating the payload when it is not necessary should be straightforward).

The limited heap interface is provided through a constructor, a push_back function that sifts new elements through the structure, and (constant) iterators that allow access to individual elements. All the rest of original vector functions are declared private, so that their inadvertent use does not invalidate the heap property. Most of these functions, such as erase and pop_back, are actually inapplicable to the heap structure. The only notable exception is operator[], whose non-constant version may render the heap inconsistent; we block it altogether as it is impossible to selectively allow only the constant version.

Whenever the push_back function decides to add a new item to the heap, it first uses std::pop_heap to pop off the smallest element. The way pop_heap is implemented, it moves the

[1] The complete code can be obtained from the Dr. Dobb's Journal Web site at http://www.ddj.com/ftp/.

first (i.e., smallest) heap element to the last position (namely, vector[_n-1]), and then restores the heap property. Subsequently, the new value is injected into this last position; using vector::push_back would both unnecessarily grow the vector *and* include anew the element that has just been removed by pop_heap. Ultimately, function std::push_heap is invoked to include the newly added element into the heap.

**Caveat:** For various combinations of $k$ and $N$, solution #2 (heap sort) may be asymptotically more time-efficient than #3, but the modest memory requirements of the latter can hardly be beaten. (Mathematically inclined readers will find out by complexity comparison that the former solution is preferable when $\frac{k}{\log k - 1} < \frac{N}{\log N}$, or roughly when $k << N$). Note also that in both solutions elements are extracted from the heap in a *sorted* order. This might be a benefit (if sorted order is actually desired), or a drawback (if stable[2] operation is necessary). In the latter case, the payload fields may be utilized to track the original element ordering.

```
using namespace std;

template<class _Key, class _T> class KMaxValues : public vector<pair<_Key, _T> > {
  typedef vector<pair<_Key, _T> > Base;

  int _n; /* maximum size allowed */

  /* Block access to extraneous functions inherited from std::vector,
     lest their invocation might invalidate heap properties. */
  using Base::assign; using Base::erase; using Base::insert;
  using Base::pop_back; using Base::resize; using Base::swap;
  using Base::operator[];using Base::iterator;

  struct greater : public binary_function<value_type, value_type, bool> {
    bool operator()(const value_type& x, const value_type& y) const
      { return (x.first > y.first); }
   };
public:
  explicit KMaxValues(int maxSize = 1) : _n(maxSize)
    { reserve(maxSize); /* preallocate storage */ }

  void push_back(const value_type& x) {
    if (size() < _n) {
      Base::push_back(x);
      push_heap(begin(), end(), greater());
    } else { /* maximum size reached */
      if (x.first < begin()->first)
        return; /* no need to add the element at all */

      /* delete the smallest element, then add the new one */
      pop_heap(begin(), end(), greater());
      (*this)[_n-1] = x; /* inserts the new element into the last position */
      push_heap(begin(), end(), greater()); /* restore the heap property */
    }
  }
};
```

**Listing 1. Template-based implementation of limited heap.**

[2] Reminiscent of *stable sort*, stable operation in this context simply preserves the original relative order of equivalent elements.

**Other options**

At this point you probably start wondering "Is the limited heap algorithm optimal? Is it the best we can hope for? Can we really achieve the *O(N)* lower bound necessary for scanning the input sequence?" It turns out that there are algorithms that can do this (alas, have we said that there ain't such thing as a free lunch?)

There are so-called *selection algorithms* that find the *k*-th largest (single) element in a sequence[3]. Given such an element, the sequence can be trivially partitioned (in a single *O(N)* pass) so that all the *k* largest elements are grouped together. The problem is, however, that algorithms that work reasonably on average, might require as much as $O(N^2)$ time in the worst case, while algorithms with guaranteed *O(N)* selection time are extremely slow in practice [1, Chapter 10]. In either case, *O(N)* additional memory is required to store the entire sequence – a considerable drawback when working "on-the-fly".

## Summary

The big-O time complexity is very important in choosing an algorithm. However, in real life one should not blindly select the algorithm that advertises the best complexity. Other concerns, such as memory requirements, can often make a seemingly inferior algorithm preferable in practice.

## Sidebar (The Heap data structure)

Heap is a very simple but useful data structure, which allows you to enter a sequence of elements in an arbitrary order and then retrieve them one by one in a sorted order. There are naturally other data structures that can perform the same task, but heap is arguably the simplest, most efficient and easiest to implement.

Basically, heap is a binary tree with two special properties:

*Heap property:* the value of each node is smaller than those of its sons. This effectively implies that the root element is the smallest one, so it can be readily accessed in *O(1)*. On the other hand, finding the largest element is inefficient. It must reside in one of the leaves of the tree (since if it had a child, it would not be the largest one), but in order to find it you must painstakingly scan all the leaves.

*Tree property:* heap is an *almost complete* binary tree, that is, all the elements up to a certain depth are present in the tree, except maybe for some elements at the lowermost level (see Figure 1). The maximum depth of the heap tree is $log_2\ n$, where *n* is the total number of nodes. Consequently, you don't need to use a tree to actually implement a heap – an array (or a `vector`) will do. This way, for a node residing at index *i*, its two sons reside at indices *2*i+1* and *2*i+2*, while its parent resides at *(i-1)/2*.

[3] In computer science, a closely related value of the *k-th smallest* element is referred to as the *k-th order statistic* of a sequence.

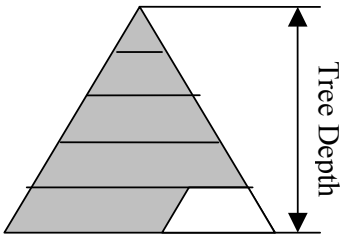


**Figure 1**: Tree property. Only the rightmost part of the lowest level can be empty.

When a heap is embedded in an array as explained above, we can define its "last" element as the very last element of the host array. Note that this element can be safely removed from the heap without disturbing any of its properties. It is this observation that allows to efficiently extract the smallest element from the heap – you simply remove the last element from the heap, inject it in place of the smallest element (i.e., at the root), and "bubble" it down, restoring the heap property for every node it encounters en route (Figure 2 illustrates this process). The time complexity of this operation is bounded by the tree depth and equals *O(log n)*.

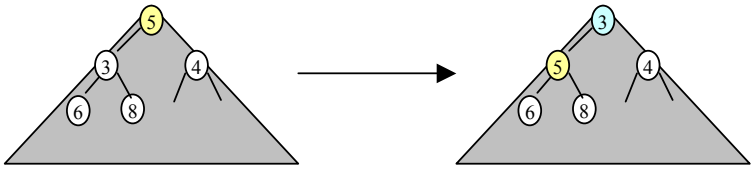


Figure 2: Restoring the heap property. Node 5 must be moved since it is larger than both of its sons. It switches places with its smallest son, in this case the left one. After the swap, node 5 is smaller than both of its sons (6 and 8), and the heap property is completely restored.

To insert a new element into the heap, you just insert it after the last array element, and "bubble" it up similarly to restore the heap property. The complexity of this operation is also *O(log n)*.

Finally, an unsorted array can be converted into a heap ("heapified") by performing the "bubble" operations in a particular sequence. Interestingly, while the worst-case complexity of any given "bubble" operation is *O(log n)*, it can be shown [1] that all the operations required for heapification together sum up to only O(n)!

And the best news is that you don't even need to implement any of the heap manipulation functions, as they constitute an integral part of the C++ standard library [2]. The three heap operations described above are realized by the STL functions `std::pop_heap`, `std::push_heap` and `std::make_heap`, defined in the header file `<algorithm>`.

## References

[1] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *"Introduction to Algorithms"*, MIT Press, 1990.

[2] *"Information Technology – Programming Languages – C++"*, International Standard ISO/IEC 14882-1998(E).

## About the authors

**Evgeniy Gabrilovich** is a Ph.D. student in Computer Science at the Technion – Israel Institute of Technology. He is a member of the ACM and the IEEE. His interests involve computational linguistics, information retrieval, and machine learning. He can be contacted at gabr@cs.technion.ac.il.

**Alex Gontmakher** is a Ph.D. student in Computer Science at the Technion – Israel Institute of Technology. His interests include parallel algorithms and constructed languages. He can be reached at gsasha@cs.technion.ac.il.