Informatics in Education, 2002, Vol. 1, 73–92



# Finding the Median under IOI Conditions

Gyula HORVÁTH
*University of Szeged, Department of Informatics*
*H-6701 Szeged, P.O. Box 652, Hungary*
*e-mail: horvath@inf.u-szeged.hu*

Tom VERHOEFF
*Eindhoven University of Technology, Faculty of Mathematics and Computing Science*
*Den Dolech 2, NL–5612 AZ, Eindhoven, The Netherlands*
*e-mail: t.verhoeff@tue.nl*

Received: September 2002

**Abstract.** We explain the ingredients of the International Olympiad in Informatics (IOI), which is a challenging competition for high-school students focusing on algorithmic problem solving. We treat in detail the MEDIAN task, which the authors created for IOI 2000: Given an odd number of objects, all of distinct strength, develop an efficient algorithm to determine the object of median strength, *using as* only *operation a function that returns the median of* three *objects*. This problem is easy to formulate and understand. It is related to well-studied standard computing problems, but further analysis of this problem leads to interesting algorithms and variations of the heap data structure. We finish by pointing out some open problems related to this task and we invite you to contribute nice competition tasks for future IOIs.

**Key words:** computing competitions, algorithms, median finding.

## 1. Introduction

The International Olympiad in Informatics (IOI) is an annual competition in computing science (informatics) for talented high-school students from all over the world (IOI, 2002). Fourteen successful IOIs have been organized since the first IOI hosted by Bulgaria in 1989.

In this article, we hope to accomplish three things:

- draw attention to the IOI and, especially, to one of its nice competition tasks;
- make clear that it is quite a rewarding challenge to develop good IOI competition tasks, and invite researchers to contribute in this area;
- pose some unsolved problems for further research.

We begin by explaining the ingredients of the IOI in Section 2. In Section 3, we present the MEDIAN task that we created for IOI 2000 and we discuss some of the difficulties surrounding the development of IOI competition tasks. The main part of the article concerns various approaches to solving it. Section 6 concludes the paper with some open problems and an invitation to contribute nice computing tasks to the IOI.

## 2. International Olympiad in Informatics

The IOI is modeled after the International Mathematical Olympiad (IMO), which started in 1959 and is the oldest of several international science olympiads (International Science Olympiads, 2002). The three main goals of the IOI are:

- to discover, encourage, bring together, challenge, and give recognition to young people who are exceptionally talented in the field of informatics;
- to foster friendly international relationships among computer scientists and informatics educators;
- to bring the discipline of informatics to the attention of young people.

The IOI has two competition days and a social-cultural program. It usually takes place during a week in the summer and is hosted by one of the participating countries. Nearly 80 countries were represented at IOI 2002. Each participating country sends a delegation of four students accompanied by two leaders. These students are typically selected in national olympiads in informatics.

The IOI competition offers six computing tasks in two sessions of five hours each. Although the problems are algorithmic in nature, it is also required that contestants implement their algorithms in one of the allowed programming languages[1]. For that purpose each contestant is provided with a computer and program development tools.

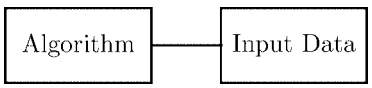


Fig. 1. Batch type of task: Input data directly available.

In traditional IOI competition tasks, all input data is directly available to the algorithm, in one batch (see Fig. 1). Task PALIN from IOI 2000 (China) created by Sergey Melnik from Latvia is a nice example:

> Design an efficient algorithm that reads a sequence of characters and outputs the minimum number of characters to be inserted into the input sequence to make it a palindrome.

For example, for input `'Ab3bc'` the output should be 2, because by inserting two characters a palindrome can be made (e.g., `'Acb3bcA'`), but no palindrome can be made by inserting fewer than two characters. Note, however, that it is not required to produce a witness.

At IOI 1995, the second author introduced another kind of task (Verhoeff, 1995), which involves a dialogue between the algorithm and its environment (see Fig. 2). For that reason, it is also called a reactive task. Games, such as *Master Mind*, fall in this category. The input data, such as the secret code in Master Mind, is indirectly available through the environment, which offers a limited set of operations.

[1] Nowadays: Pascal, C, and C++.


Fig. 2. Reactive type of task: Input data indirectly available through the environment.

## 3. Task MEDIAN from IOI 2000

The authors created the following reactive task for IOI 2000 (China):

> Given is an odd number of objects, all of distinct strength. The only way to compare strengths is through the function *Med3*$(a, b, c)$ that returns the object of median (middle) strength among three distinct objects:
>
> $$\min\{\, a, b, c \,\} < \textit{Med3}(a, b, c) < \max\{\, a, b, c \,\}.$$
>
> Design an efficient algorithm to determine the object of median strength among all given objects, using only function *Med3*.

This is only a summary of the actual task description, which covers almost two pages. Additional details are needed because

- contestants must deliver an *implementation* of their algorithms in one of the allowed programming languages,
- the contestants' scores are determined through *execution* of the submitted programs, and
- contestants must be able to base their design decisions on scientific reasoning, rather than guessing and bluffing.

Thus, engineering constraints must be precisely specified. These constraints concern:

- bounds on input values;
- bounds on computational resources, such as time and memory;
- interface to the reactive environment.

These details, in turn, depend on the intended level of difficulty, the characteristics of the competition computers, the program development tools, and the software system used in determining the scores. In this article, we ignore some of these details.

At the IOI, it is customary to allow for a range of scores, rather than binary scoring (pass/fail) as is done in the ACM International Scholastic Programming Contest (ACM ICPC, 2002). In case of task MEDIAN, this was accomplished as follows. As a precondition, the number $N$ of objects is bounded by

$$5 \leqslant N \leqslant 1499, \tag{1}$$

and, for each run, the number $C$ of calls to *Med3* is bounded by

$$C \leqslant 7777, \tag{2}$$

as a postcondition independent of $N$. For scoring, the program is executed on ten cases, including both small and large values of $N$. Each correctly solved case yields 10 points. The fixed bound on the number of calls to *Med3* implies that small values of $N$ are relatively easy, whereas large values are relatively hard. We present the scoring cases in Appendix A.

The (abstract) interface to the environment is given by three operations:

- *GetN*, which returns the number $N$ of objects and must be called once at the beginning;
- *Med3*$(a, b, c)$, which returns the object of median strength among the three *distinct* objects $a, b, c$;
- *Answer*$(x)$, which reports the answer $x$ and must be called once at the end.

At IOI 2000, these operations were made available through a library, which contestants had to link with their program.

The straightforward part of the library functionality is that *GetN* reads $N$ and the object strengths from an input file, that *Med3* responds to queries, and that *Answer* writes the answer and the number of calls on *Med3* to an output file. However, there is more to it than just that.

Such a library must be **robust**, that is, protected against accidental misuse, by checking all preconditions on its use. It must also be **secure**, that is, protected against intentional abuse, by making it extremely difficult to cheat. Furthermore, the contestants must have a way to **experiment** with the library during the competition, that is, they should be able to do test runs, by supplying their own choice of input data and inspecting the results. Our library distinguished an experimental and a scoring mode through a digital signature in the input data. If the input data has a valid signature, then the library knows that the data is scrambled and it will not even store a descrambled version of the data to maintain security. If there is no signature, then the input data is interpreted plainly. Finally, during experimentation and in case of disputes about scoring, it is convenient if the library leaves a **record** of the complete dialogue. That is, it not only counts how many times *Med3* is called, but also reports the actual parameters of each call.

## 4. Analysis

Median finding has been studied extensively in the literature (Blum *et al.*, 1972; Knuth, 1998; Mehlhorn, 1984). However, we are not aware of publications dealing with the variant presented here.

Let us first analyze task MEDIAN. Because the number of objects is odd and all their strengths are distinct, the object of median (middle) strength is **uniquely determined**.

Let $s$ be the **strength mapping** from object labels $L$ (1 through $N$) to object strengths (integers). That is, $s(i)$ is the strength of object $i$. We define the weaker-than relation $<_s$ on $L$ by

$$i <_s j \Leftrightarrow s(i) < s(j). \tag{3}$$

For set $V$, with $V \subseteq L$, and an odd number of objects, let $\text{med}_s\, V$ be the object in $V$ with **median strength**, that is, the object $m \in V$ with

$$\#\{\, i \in V \mid i <_s m \,\} = \#\{\, i \in V \mid m <_s i \,\}. \tag{4}$$

We are asked to determine $\text{med}_s\, L$, without inspecting $s$ directly, but only by using the operations $\text{med}_s\, V$ for sets $V \subseteq L$ of size three ($\#V = 3$):

$$\textit{Med3}(a, b, c) = \text{med}_s\{\, a, b, c \,\}. \tag{5}$$

With the given operation *Med3* it is impossible to determine which is the stronger or weaker of two objects. It is, however, unnecessary to identify the extremal objects. Observe that object $\textit{Med3}(a, b, c)$ is known *not* to be extremal (weakest or strongest).

We introduce some further notation. For $V$ a nonempty set, let $\min_s V$ be the **weakest** object in $V$, and $\max_s V$ the **strongest** object in $V$. Consider set $V = \{\, a, b, c \,\}$ of three distinct objects. We then have

$$\min_s V <_s \text{med}_s\, V <_s \max_s V,$$

and also

$$V = \{\, \min_s V, \text{med}_s\, V, \max_s V \,\}.$$

More generally, for any set $V$ with an odd number of objects $\geq 3$, and any subset $W$ of $V$ with 3 objects, we have the following relations:

$$\min_s V <_s \text{med}_s\, W <_s \max_s V,$$

and also

$$\text{med}_s\, V = \text{med}_s(V - \{\, \min_s V, \max_s V \,\}). \tag{6}$$

That is, the median is invariant under elimination of the extremes. Note that $\text{med}_s\{\, a \,\} = a$.

### 4.1. *Onion Peeling*

A very simple algorithm to determine the object of median strength is based on repeated elimination of two extremal objects as in (6) We call it **onion peeling elimination** (OPE). The two extremes can be found be repeated elimination of non-extremal objects, based on the earlier observation that the median $\textit{Med3}(a, b, c)$ for distinct $a, b, c \in V$ is *not* an extremal object of $V$:

**const** $L = \{\, 1, \ldots, \mathit{GetN} \,\}$
**var** $V, W$: **set of** int

$V := L$
invariant: $\mathit{odd}(\#V)$, and $\mathrm{med}_s\, V = \mathrm{med}_s\, L$
variant function: $\#V$
**while** $\#V \neq 1$ **do begin** assert $\#V \geqslant 3$
  $W := V$
  invariant : $\{\, \min_s V, \max_s V \,\} \subseteq W \subseteq V$
  variant function : $\#W$
  **while** $\#W \neq 2$ **do begin** assert $\#W \geqslant 3$
    choose $a, b, c \in W$, all distinct
    $W := W - \{\, \mathit{Med3}(a, b, c) \,\}$
  **end**
  assert: $W = \{\, \min_s V, \max_s V \,\}$
  $V := V - W$
**end**
assert: $V = \{\, \mathrm{med}_s\, L \,\}$
*Answer* (the object in $V$)

The inner loop iterates $\#V-2$ times, since each iteration eliminates one object from $W$, which starts out equal to $V$ and terminates with $\#W = 2$. The outer loop iterates $(N-1)/2$ times, since each iteration eliminates two objects from $V$, which starts out with $N$ objects and terminates with $\#V = 1$. Therefore, the total number of calls to *Med3* equals a sum with $(N-1)/2$ terms:

$$(N-2) + (N-4) + \ldots + 3 + 1 = \left(\frac{N-1}{2}\right)^2 .$$

For $N \leqslant 177$, this results in no more than $7744$ calls, and for $N \geqslant 179$ it requires at least $7921$ calls. Apparently, we need to improve on this for values of $N$ larger than $177$.

It is also clear that this elimination approach "throws away useful information", because the inner loop starts "from scratch" on each iteration. It is not so obvious how to reuse results of a previous iteration and how to choose $a, b, c$ carefully to improve reuse.

One could analyze the situation for some small values of $N$, such as 5, 7, and 9. But this easily gets you into all kinds of irrelevant nitty-gritty arguments that are hard to generalize.

For $N = 5$, the elimination approach above requires 4 calls to *Med3* for determining the object of median strength. It can, however, always be found in no more than 3 calls, but this is a little tricky:

For the first call we may assume without loss of generality

$$\textit{Med3}(a, b, c) = b,$$

which implies the order $abc$ (the strength of $b$ lies between $a$ and $c$). The second call is critical. Calling *Med3*$(b, c, d)$ is no good, because when *Med3*$(b, c, d) = b$ it possibly requires four calls to find the median of five (exercise for the reader). The second call needs to be *Med3*$(a, c, d)$. The results $a$ and $c$ are equivalent, since they yield a total order on the four objects, namely $dabc$ and $abcd$ respectively. The result $d$ leads to a true partial order, with unordered pair $b, d$ wedged between $a$ and $c$.

- In the case of the total order, the third call compares the two center objects to the fifth: *Med3*$(a, b, e)$ for $dabc$, and *Med3*$(b, c, e)$ for $abcd$. The result of this call is the median of all five objects.
- In the case of the true partial order, the third call also compares the two center objects to the fifth: *Med3*$(b, d, e)$. The result of this call is the median of all five objects, though you may have to give it a second thought to convince yourself.

This concludes the special case $N = 5$.

### 4.2. *Lower Bound*

By the way, a lower bound on the number of calls for obtaining the median of $N$ objects is $(N-1)/2$. Each object needs to be involved in at least one call to *Med3*. Furthermore, the triples for which *Med3* is called need to be "connected", when considering the graph over all triples with an edge between two triples if they have a common object. This lower bound can be achieved (with luck) by "guessing" the median $m$ and making $(N-1)/2$ calls of the form *Med3*$(a_{2i}, m, a_{2i+1})$ where all $a_i$ ($0 \leqslant i < N-1$) are distinct and such that each call has $m$ as result.

### 4.3. *Total Ordering*

Another approach is based on a "bold" idea: sort the objects on strength, either increasing or decreasing. The median can then be found in the middle. Sorting is overkill to obtain just the median, but let us see how far it gets us.

When there are three objects, a single call to *Med3* will enable us to sort them modulo up/down, that is, put them in order without knowing whether it is increasing or decreasing. From now on, we will use the term "**ordering**" to mean "sorting on strength *modulo* up/down".

How to order more than three objects? What about the traditional sorting methods based on pairwise comparisons?

- **Selection sort** does not look promising, because it requires determining an isolated extreme object.

- **Insertion sort** might work, provided that we can insert an object into an already ordered list. Standard insertion sort has quadratic worst-case and average-case behavior, but insertion can also be done by binary search in the case of conventional sorting. This approach is worked out below.
- **Merge sort** has the extra complication that when merging two ordered lists, it is not known what their relative direction of order is: one list may have been sorted up, the other down. From what ends to start merging?
- **Quicksort** requires partitioning of the objects, ordering the resulting parts, and catenating the ordered parts. As with merge sort, extra care is needed when combining two independently ordered parts. Standard quicksort has quadratic worst-case behavior and $N \log N$ average-case behavior.
- **Heap sort** usually involves twice the number of comparisons of merge sort, and heaps are based on a particular direction, making the process of combining heaps more complicated (as occurs in the first phase of standard heap sort).

Here are some further observations on ordering. If $a$ and $b$ are known to lie on the *same side* of $c$, that is *Med3*$(a, b, c) \neq c$, then the call *Med3*$(a, b, c)$ effectively orders $a$ and $b$ (with respect to $c$). Thus, after finding an extreme object (e.g., via elimination as explained above), one can use *Med3* as a **binary comparison**.

Note that in such use, each call yields no more than *one bit* of information (two equally likely outcomes). In general, a call *Med3*$(a, b, c)$ can yield *three* results, providing up to $\log_2 3 = 1.58\ldots$ bits of information. For $N \geq 2$, there are $N!/2$ ways to order $N$ objects (the reverse order cannot be distinguished). Consequently, a lower bound on the number of calls for ordering $N$ objects ($N \geq 3$) is

$$\frac{\log_2(N!/2)}{\log_2 3} = \log_3 \frac{N!}{2}.$$

For $N = 5$, this yields a lower bound of $3.7\ldots$ and, hence, at least four calls are needed to *order* five objects. We do not know whether this lower bound can be achieved. Note that the method explained earlier to determine the median of five objects does not always yield a total order.

### 4.4. *Insertion Ordering*

Let us investigate insertion sort further. Given is an ordered list of, say, $k$ objects, and an object $x$ not yet in the list. Question is to determine the location where $x$ needs to be inserted into the list to make it an ordered list of $k+1$ objects.

The only interesting calls are of the form *Med3*$(a, x, b)$ where $a$ and $b$ occur in the list. Depending on the result of the call, one knows in which of three parts $x$ belongs:

- to the left of $a$,
- between $a$ and $b$, or
- to the right of $b$.

There are several ways to choose $a$ and $b$:

- If $a$ and $b$ are chosen at the right end of the list, and moved one step to the left whenever $x$ appears left of $a$, then we obtain a **linear search**.
- If $a$ and $b$ are chosen next to each other near the middle of the list, then we obtain a **binary search**.
- If $a$ and $b$ are chosen at about one third and two third of the list length, then we have, what can be called, a **ternary search**.

The ternary version has the best worst-case performance (measured in terms of the number of calls to *Med3*): $\mathcal{O}(N \log N)$ (details below), which is also its average-case behavior. The linear and binary versions have better best-case performance: linear (if every time *Med3*$(a, x, b) = x$, then $x$ is pinpointed between the neighbors $a$ and $b$ in a single call). The worst-case performance of the linear method is quadratic, and that of the binary method $\mathcal{O}(N \log N)$, but with a larger constant than for the ternary version. Some statistics are presented in Table 1.

Table 1
Number of calls for insertion ordering on $N = 1499$ objects

| **Insertion Method** | **List Variant** | **Worst case** | **Average case** |
|---|---|---|---|
| Linear | Full | 561749 | 282532 |
| | Half | 421499 | 169655 |
| | Zoom | 281623 | 141676 |
| Binary | Full | 12953 | 11680 |
| | Half | 12477 | 11492 |
| | Zoom | 11481 | 10471 |
| Ternary | Full | 9399 | 8977 |
| | Half | 9399 | 8522 |
| | Zoom | 8319 | 8041 |

### 4.5. *Improved Insertion Approaches: Half List and Zoom List*

There are two noteworthy improvements on the insertion approach[2]. First of all, consider the situation where the list built up by insertions includes $(N+1)/2$ of the $N$ objects. After inserting another object, it is clear that the resulting rightmost element cannot be the median, because there are now $(N+1)/2$ objects to its left. Recall that the median will have exactly $(N-1)/2$ objects on either side. Thus, the rightmost element can be eliminated, yielding again a list with $(N+1)/2$ objects. When all remaining objects have been handled similarly, the rightmost object in the final list is the median: it has $(N-1)/2$ objects to the left, and a same number would have ended up to its right would they have been retained in the list. By restricting the maximum list length, the total number of calls

[2] In fact, these apply more generally to other ordering approaches as well.

to *Med3* is possibly reduced. We call this the **half-list** insertion approach, as opposed to the **full-list** approach presented earlier.

The same reasoning can be carried even further. Consider again the situation where the insertion list has reached length $(N+1)/2$. After inserting another object, is is even clear that *neither* of the two extreme objects can be the median, because both have too many objects to one side. It is also clear that these extremes "straddle" the actual median. Thus, when dropping both extremes from the list, the median of the remaining objects (in the list and those still to be handled) equals the median of the original set. For each additional object inserted, two extremes can be eliminated, until all objects are handled and the list finally consists of just one object, the median. The number of calls to *Med3* is possibly even further reduced. We call this the **zoom-list** approach, because it zooms out and then zooms in on the median.

It is straightforward to determine for given $N$ what the exact worst-case numbers of calls to *Med3* are for the various insertion approaches. These are shown in Table 1, together with experimental average-case numbers for $N = 1499$. It is obvious that a better method is still called for.

### 4.6. *Expected-Time Linear Algorithm*

The median can also be selected by recursively partitioning the set, as in QuickSort, and discarding the subset that is known *not* to contain the median. This algorithm is also named FIND. We only have considered algorithms that partition into three parts, based on choosing *two* **pivot** objects rather than one. We call this **Ternary Partitioning Find** (TPF). In general, these methods are quadratic in worst case, but linear in average and best case.

There are various ways to choose the pivots:

**Straddled:** One at each end of the list (TPFS)

**First:** Both at the same end, say the first two objects (TPFF)

**Proportional:** At one third and two thirds in the list (TPFP)

**Random:** At randomly selected positions in the list (TPFR).

For TPFS and TPFF, the sorted input is bad, but for TPFP and TPFR it is (very) good. TPFR has no specific worst-case inputs. Worst-case input for TPFP depends on details of rounding when choosing the proportional pivots. Experimental data suggests that the average-case number of queries is about $2N$. In particular, for $N = 1499$, it is about $3300 \pm 1600$ measured on 100 random cases. None of these algorithms ever exceeded the bound of 7777 calls to *Med3* on the random cases.

### 4.7. *Worst-Case Linear Algorithm*

For comparison-based median finding, there is also a famous *worst-case linear* algorithm (Blum *et al.*, 1972). The basic idea is the same as partitioning, explained in the preceding

subsection. The key ingredient is a method for choosing the pivot in such a way that the two parts are not too extreme in size. This can be accomplished by partitioning the set into maximal subsets of size at most 5, determining their medians, and then the median of these median-of-five objects is determined recursively and taken as pivot.

Refining this approach to task MEDIAN is rather complicated, especially if you want to do ternary partitioning. The median-of-five can be found by just 3 calls to *Med3* (cf. Section 4.1), but the linear constant for the complete algorithm is prohibitively large (for $N = 1499$, even the best case takes over 10000 calls to *Med3*). Competitors were not required to discover a worst-case linear solution for a $100\%$ score.

## 5. Heap-Based Algorithms

The half-list insertion approach of Section 4.5 can be generalized to a half-*heap* algorithm. A set of objects is called a "heap" if it supports these three operations:

- *BuildHeap*$(V)$: returns a "heap" for the set $V$ of objects
- *ReplaceMin*$(H, x)$: returns $H$ if $x <_s \min_s H$, and returns "$H$ in which $\min_s H$ is replaced by $x$" if $x >_s \min_s H$; the size of the heap remains the same
- *GetMin*$(H)$: returns $\min_s H$

Using these operations, the generic **half-heap** algorithm is:

**var** $H$: **heap of** int
  $N, m, x$: int

$N :=$ *GetN*
$m := (N + 1)$ **div** 2
$H :=$ *BuildHeap* ( $\{ 1, \ldots, m \}$ )
assert: $\#\{ 1 \leqslant i \leqslant m : \min_s H <_s i \} = m - 1$
**for** $x := m + 1$ **to** $N$ **do**
  $H :=$ *ReplaceMin*$(H, x)$
  invariant: $\#\{ 1 \leqslant i \leqslant x : \min_s H <_s i \} = m - 1$
**end**
assert: $\#\{ 1 \leqslant i \leqslant N : \min_s H <_s i \} = m - 1 = \#\{ 1 \leqslant i \leqslant N : \min_s H >_s i \}$
*Answer* ( *GetMin*$(H)$ )

### 5.1. *Ordered List*

An obvious implementation of the heap is an ordered list. Using ternary search (cf. 4.4), the *ReplaceMin* operation takes at most $\lceil \log_3(m + 1) \rceil$ calls. The *BuildHeap* operation requires in worst case

$$C(m) \;=\; 3 + \sum_{i=4}^{m} \lceil \log_3(i + 1) \rceil \tag{7}$$

calls of *Med3*. For $N \geqslant 1357$ the worst-case number of calls to *Med3* exceeds the bound 7777.

We are looking for a data structure that supports the operation *ReplaceMin* as efficient as the ordered list, but that provides a better implementation for *BuildHeap*. The basic idea is that only a partial order is enough instead of a total ordering of the objects.

### 5.2. *Binary Tree Heap*

Implementing the "heap" by a conventional binary tree is not promising. The *BuildHeap* operation can be implemented in worst-case linear. But the number of comparisons for one replacement into such a heap is about twice the height of the tree in the worst case. We have not considered this approach further.

### 5.3. *Two-Dimensional Ordered List*

A two-dimensional list is a data structure as depicted in Fig. 3. Nodes are arranged in rows and columns, each column forms a (vertical) list and the vertical lists are connected by their head nodes to form a horizontal list. We say that a two-dimensional list is ordered with respect to a given $\leqslant$ linear ordering relation if for any two nodes $x$ and $y$ with an arrow from $x$ to $y$ then $\leqslant$ holds between the content of $x$ and $y$. Therefore, the head node of the horizontal list (which is the head node of the first vertical list) contains the smallest of all data items in the two-dimensional ordered list.

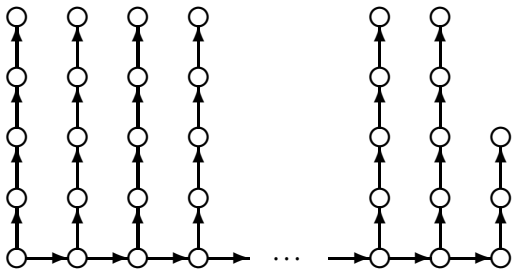

Fig. 3. Example of a 2d-list.

We can implement the required operations with two-dimensional ordered lists in a straightforward way. First, choose an upper bound $b$ on the length of the vertical lists.

*BuildHeap*$(V) =$

**var** $L$ : **2d-list of** int
Subdivide $V$ into disjoint subsets $V_1, \cdots, V_k$ with $\#V_i \leqslant b$
**for** $i := 1$ **to** $k$ **do begin**
  Create a vertical list $L_i$ from the set $V_i$
  *Sort*$(L_i)$
**end**
Create a horizontal list $L$ from the vertical lists $L_i, i = 1, \cdots, k$
*Sort*$(L)$ according to the head elements of the vertical sublists of L
*BuildHeap* $:= L$

*ReplaceMin*$(L, x) =$

Detach the head vertical list $L_1$ of $L$
Insert $x$ into $L_1$
Delete the head node of $L_1$
Insert $L_1$ into $L$ according to its head element
$ReplaceMin := L$

Denote by $B(m)$ the worst-case number of calls to *Med3* for *BuildHeap*$(V)$ for a set with $\#V = m$. We calculate an upper bound on $B(m)$.

Assume that all vertical sublists are of length $b$, except one. Then the length of the horizontal list is $l = \lceil \frac{m}{b} \rceil$. If sorting of the lists is done by ternary insertion sort, we obtain the following upper bound.

$$\begin{aligned} B(m) &\leqslant lC(b) + C(l) \\ &= \left\lceil \frac{m}{b} \right\rceil \Big( 3 + \sum_{i=4}^{b} \lceil \log_3(i+1) \rceil \Big) + 3 + \sum_{i=4}^{\lceil \frac{m}{b} \rceil} \lceil \log_3(i+1) \rceil. \end{aligned}$$

Now we analyze the performance of *ReplaceMin*. Assume that the two-dimensional list has been created by *Buildheap* for a set $\#V = m$ with vertical list length $b$. Insertion into both vertical and horizontal lists can be performed by ternary search. With this assumption we obtain an upper bound on the worst-case number of calls to *Med3*, denoted by $R(m)$:

$$R(m) \leqslant \lceil \log_3(b+1) \rceil + \left\lceil \log_3 \left\lceil \frac{m}{b} \right\rceil \right\rceil.$$

One can see that $b$ should be chosen to be $3^k - 1$ for some $k$. For the possible input sizes ($5 \leqslant N \leqslant 1499$), the best choice for $b$ is $8$. In this case, for the median algorithm an upper bound on the worst-case number of calls to *Med3* for $N = 1499$ is 6816.

*Multiple-Dimensional Ordered List*

We can further improve the BuildHeap phase by generalizing the notion of the two-dimensional ordered list to arbitrary dimensions. A $q$-dimensional list ($q$**d-list** shortly) is defined recursively as follows.

- the empty list $\langle\rangle$ is $q$d-list for any $q$.
- a 0d-list is an object itself.
- $q$d-list $L$ is a list $\langle L_1, \cdots, L_k \rangle$ of $(q-1)$d-lists.

The dimension of a $q$d-list $L$ is denoted by *dim*$(L)$. For a $q$d-list $L = \langle L_1, \cdots, L_k \rangle$, $L_i$ is called the $i$th sublist (of dimension $q-1$) of $L$. We define operations on nonempty $q$d-lists. Let $L = \langle L_1, \cdots, L_k \rangle$ and $dim(L) = q$

- *Head*$(L)$ is the $(q-1)$d-list $L_1$.
- *Tail*$(L)$ is the $q$d-list $\langle L_2, \cdots, L_k \rangle$.

- *First*$(L)$ is the object $L$ if *dim*$(L) = 0$ else *First* ( *Head*$(L)$ )

The data elements contained in a $q$d-list $L$ are those that can be obtained as *First*$(R)$ for some lower dimensional sublist of $L$.

We say that a $q$d-list $\langle L_1, \cdots, L_k \rangle$ is ordered with respect to an underlying $\leqslant$ linear ordering relation if

- *dim*$(L) = 0$, or
- *dim*$(L) > 0$ and *First*$(L_1) \leqslant$ *First*$(L_2) \leqslant \cdots \leqslant$ *First*$(L_k)$ and each sublist $L_i$ is an ordered $(q-1)$d-list.

Because of the transitivity of the relation $\leqslant$ the smallest element contained in an ordered $q$d-list $L$ is *First*$(L)$.

We can also define $q$d-lists as a subset of the space $\mathbf{N}^q$. In order to do this we introduce a binary relation $\rightarrow$ on $\mathbf{N}^q$.

Let $x = (x_1, \cdots, x_q)$, $y = (y_1, \cdots, y_q) \in \mathbf{N}^q$. $x \rightarrow y$ holds if and only if there is an index $1 \leqslant k \leqslant q$ such that $x_i = y_i = 1$ for all $i < k$ and $y_k = x_k + 1$ and $x_i = y_i$ for all $i > k$. A subset of grid points $L \subseteq \mathbf{N}^q$ is called a $q$d-list, if for any $x, y \in \mathbf{N}^q$ if $y \in L$ and $x \rightarrow y$ implies $x \in L$.

Then the $i$th sublist of $L$ is $\{ (x_1, \cdots, x_{q-1}) : (x_1, \cdots, x_{q-1}, i) \in L \}$.

In order to devise an efficient algorithm for BuildHeap using a $q$d-list, we set an upper bound $b_1$ on the length of 1d-sublists and an upper bound $b$ on the length of higher dimensional sublists.

*BuildHeap*$(V) =$

**var** $L$: **Heap of** int
**if** $\#V \leqslant b_1$ **then begin**
  Create a 1d-list $L$ from the elements of $V$
  *Sort*$(L)$
  *BuildHeap* $:= L$
**end else begin**
  Subdivide $V$ into nearly equal-sized, disjoint subsets $V_1, \cdots, V_b$
  $L :=$ ⟨*BuildHeap*$(V_1), \cdots,$ *BuildHeap*$(V_b)$⟩
  *Sort*$(L)$ according to the First elements of the sublists of L
  *BuildHeap* $:= L$
**end**

*ReplaceMin*$(L, x) =$

**var** $F$: **Heap of** integer;
**if** *dim*$(L) = 1$ **then begin**
  *Insert*$(L, x)$
  *ReplaceMin* $:=$ *Tail*$(L)$
**end else begin**
  $F :=$ *ReplaceMin* ( *Head*$(L), x$ )
  *ReplaceMin* $:=$ *Insert* ( *Tail*$(L), F$ )
**end**

The dimension of the list $L$ created by *BuildHeap*$(V)$ for a set with $\#V = m$ is the least integer q with $b_1 b^{q-1} \geqslant m$. Therefore we have that $q = \left\lceil \log_b \frac{m}{b_1} \right\rceil + 1$.

We assume that sorting an ordered list is done by ternary insertion sort and insertion into a sorted list is done by ternary insertion. Denote by $B(m)$ the worst-case number of calls to *Med3* for *BuildHeap*$(V)$ for a set with $\#V = m$. We calculate an upper bound on $B(m)$.

Let $c_1$ and $c$ is the number of calls of Med3 needed in worst case to sort $b_1$ and $b$ elements, respectively.

$$\begin{aligned}
B(m) &= c_1\left\lceil\frac{m}{b_1}\right\rceil + c\left(\left\lceil\frac{m}{b_1 b}\right\rceil + \cdots + \left\lceil\frac{m}{b_1 b^{q-1}}\right\rceil\right)\\
&\leqslant c_1\left(\frac{m}{b_1}+1\right) + c\left(\frac{m}{b_1 b} + \cdots + \frac{m}{b_1 b^{q-1}} + q - 1\right)\\
&\leqslant \frac{m}{b_1}\left(c_1 + c\left(\sum_{i=0}^{q-1}\frac{1}{b^i} - 1\right)\right) + c_1 + c(q-1)\\
&= \frac{m}{b_1}\left(c_1 + c\left(\frac{(\frac{1}{b})^q - 1}{\frac{1}{b}-1} - 1\right)\right) + c_1 + c(q-1)\\
&= \frac{m}{b_1}\left(c_1 + c\left(\left(1 - \frac{1}{b^q}\right)\frac{b}{b-1} - 1\right)\right) + c_1 + c(q-1)\\
&\leqslant \frac{m}{b_1}\left(c_1 + c\left(\frac{b}{b-1} - 1\right)\right) + c_1 + c(q-1)\\
&= m\left(\frac{c_1}{b_1} + \frac{c}{b_1}\frac{1}{b-1}\right) + c_1 + c(q-1).
\end{aligned}$$

We conclude that the worst-case performance of *BuildHeap* (measured in terms of the number of calls to *Med3*) is $\mathcal{O}(N)$.

Now we analyze the performance of *ReplaceMin*. Assume that the heap has been created by *Buildheap* for a set $\#V = m$ with list length bounds $b_1$ and $b$. Denote the worst-case number of calls to *Med3* by $R(m)$.

$$R(m) = \lceil\log_3(b_1+1)\rceil + (q-1)\lceil\log_3 b\rceil.$$

It is natural to chose $b = 3^k$ and $b_1 = b - 1$ for some $k$. In this case

$$\begin{aligned}
R(m) &= \log_3(b_1+1) + (q-1)\log_3 b\\
&= k + (q-1)k\\
&= kq\\
&= k\Big(\Big\lceil\log_b\frac{m}{b_1}\Big\rceil + 1\Big)\\
&= k\Big(\Big\lceil\frac{\log_3 m}{k} - \frac{\log_3(3^k-1)}{k}\Big\rceil + 1\Big).
\end{aligned}$$

For $k = 1$ we obtain that $R(m) = \lceil\log_3 m\rceil + 1$. This is the same as for totally ordered list. If $b_1 = 2$ and $b = 3$ then $c_1 = c = 2$. In this case, the upper bound $A(N)$ on the

worst-case number of calls to *Med3* of the algorithm for $N$ objects is

$$\begin{aligned} A(N) &\leqslant B(N) + \frac{N}{2} R\Big(\frac{N}{2}\Big) \\ &= \frac{N}{2}\Big(\frac{c_1}{b_1} + \frac{c}{b_1}\frac{1}{b-1}\Big) + c_1 + c(q-1) + \frac{N}{2} R\Big(\frac{N}{2}\Big) \\ &= \frac{N}{2}\Big(1+\frac{1}{2}\Big) + 2 + 2\Big\lceil \log_3 \frac{N}{4}\Big\rceil + \frac{N}{2}\Big(\Big\lceil \log_3 \frac{N}{2}\Big\rceil + 1\Big) \\ &= \frac{N}{2}\Big(\frac{5}{2} + \Big\lceil \log_3 \frac{N}{2}\Big\rceil\Big) + 2\Big\lceil \log_3 \frac{N}{4}\Big\rceil + 2. \end{aligned}$$

For $N = 1499$, this upper bound is $6388$. We note that the upper bound for the *BuildHeap* phase is 1138.

*Implementation of Ordered $q$d-Lists*

We implemented ordered $q$d-list of list size $b$ using a tree of degree $b$. The leaf nodes of the trees contain the 1-dimensional sublists. Each internal node that represents a $k$d-sublist $L = \langle L_1, \cdots, L_b \rangle$ for $k > 1$ contains the sequence

$$(\mathit{First}(L_1), p_1, \cdots, \mathit{First}(L_b), p_b),$$

where $p_i$ is the pointer pointing to the subtree representing sublist $L_i$ ($i = 1, \cdots, b$). Storing the values $\mathit{First}(L_i)$ for the sublists is useful because these values are needed to carry out an insertion into the list $L$.

## 6. Conclusion

We have presented the competition task MEDIAN of IOI 2000 and several solutions. The task concerns finding the median using as only operation the median-of-three *Med3*. Some interesting datastructures and algorithms turn up, which can also be applied to comparison-based median finding. There are still some open problems:

- What is the true minimum worst-case number of calls to *Med3* for finding the median among $N$ objects ($N$ odd)?
- What is a good worst-case linear-time algorithm (with small constant)?
- Can you design an efficient *adversarial environment* that answers *Med3* queries consistently but in a (close to) "worst" way?

There are also interesting variants of this task:

- Find the median if the only available operation *sorts* three elements, that is, identifies minimum, median, and maximum?
- Given a sequence of calls to *Med3*, including parameters and results, decide whether it gives enough information to determine the median, and if it does, indeed determine the median.

A less obvious variant of task MEDIAN was developed by the first author for CEOI 2001 under the name CHAIN OF ATOMS (Horváth, 2001):

> Design an algorithm to reconstruct the linear order of $N$ objects, which can only be inspected by a function that returns the absolute difference in ranks. Furthermore, each object can be inspected at most three times.

It is not only challenging to solve such competition problems, but also to design them. We hope that CS researchers will consider to contribute nice computing tasks to the IOI.

## A Tuning the Task Parameters for Scoring

An IOI task author can still tune various parameters, such as the upper bounds on the number $N$ of objects and the number $C$ of calls to *Med3* for task MEDIAN. These bounds are best designed together with the test cases used for scoring. The choice for upper bounds $N = 1499$ and $C = 7777$ were driven by the ability to distinguish the performance of various algorithms. At IOI 2000, the programs submitted for task MEDIAN were scored by running them on 10 test cases. These cases have been designed to detect performance differences as exhibited by the 16 algorithms in Table 2. The 10 test cases belong to 4 categories:

**M** Manually designed
**R** Randomly generated (uniform)
**N** Nearly sorted (in the identity strength mapping, the left-most, middle-most, and right-most 11 elements have been rotated left over 5 positions)
**A** Alternating outside-to-inside (1 3 5 ... 6 4 2)

The cases of type **N** and **A** were introduced specifically to penalize algorithms TPFS and TPFF. Table 3 shows how many calls each algorithm made for each test case solved

Table 2
Reference algorithms for design of test cases

| | |
|---|---|
| **OPE** | Onion Peeling Elimination |
| **LISF** | Linear Insertion Sort using Full list |
| **LISH** | Linear Insertion Sort using Half list |
| **LISZ** | Linear Insertion Sort using Zoom list |
| **BISF** | Binary Insertion Sort using Full list |
| **BISH** | Binary Insertion Sort using Half list |
| **BISZ** | Binary Insertion Sort using Zoom list |
| **TISF** | Ternary Insertion Sort using Full list |
| **TISH** | Ternary Insertion Sort using Half list |
| **TISZ** | Ternary Insertion Sort using Zoom list |
| **TPFS** | Ternary Partitioning Find using Straddled pivots |
| **TPFF** | Ternary Partitioning Find using First pivots |
| **TPFP** | Ternary Partitioning Find using Proportional pivots |
| **TPFR** | Ternary Partitioning Find using Random pivots |
| **2LHH** | 2d-List implementation of Half-Heap algorithm |
| **QLHH** | $q$d-List implementation of Half-Heap algorithm |

Table 3
How each algorithm performs on the *passed* test cases

| **Case #** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | 5 | 177 | 577 | 975 | 1087 | 1267 | 1357 | 1415 | 1415 | 1499 | |
| **Cat** | M | R | N | R | R | R | R | R | A | R | **Score** |
| **Alg** | | | | | | | | | | | |
| OPE | 4 | 7744 | | | | | | | | | 20 |
| LISF | 4 | 4062 | 619 | | | | | | | | 30 |
| LISH | 3 | 2590 | 598 | | | | | | | | 30 |
| LISZ | 3 | 2160 | 598 | | | | | | | | 30 |
| BISF | 4 | 861 | 4175 | 7051 | | | | | | | 40 |
| BISH | 5 | 843 | 4108 | 6803 | | | | | | | 40 |
| BISZ | 4 | 730 | 3621 | 6269 | 7078 | | | | | | 50 |
| TISF | 3 | 712 | 2918 | 5415 | 6143 | 7376 | | | | | 60 |
| TISH | 3 | 669 | 2707 | 5349 | 6011 | 7103 | 7642 | | | | 70 |
| TISZ | 3 | 609 | 2537 | 4889 | 5540 | 6641 | 7191 | 7511 | 7572 | | 90 |
| TPFS | 3 | 517 | | 1525 | 2842 | 3257 | 3531 | 2231 | | 3218 | 80 |
| TPFF | 4 | 395 | | 2205 | 2378 | 3635 | 3601 | 2663 | | 2493 | 80 |
| TPFP | 5 | 331 | 848 | 3512 | 1705 | 2291 | 3093 | 2860 | 2863 | 2985 | 100 |
| TPFR | 4 | 372 | 1778 | 2201 | 2507 | 2981 | 3377 | 3987 | 3279 | 3540 | 100 |
| 2LHH | 4 | 491 | 1954 | 3242 | 3605 | 4258 | 4578 | 4824 | 4149 | 5147 | 100 |
| QLHH | 4 | 508 | 2218 | 3184 | 3902 | 4517 | 4862 | 5074 | 4389 | 5354 | 100 |

within the bound of 7777 calls. The rightmost column shows the score. For $N = 1499$, algorithms 2DHH and QDHH are the only ones that stay within the bound of 7777 calls *under worst-case conditions*. The library used for scoring, however, was not able to create such worst-case conditions dynamically. It only worked with fixed test cases.

Note that when designing a task, it is not enough to produce just a table like Table 3. The performance of various algorithms on various values of $N$ must be studied carefully (worst-case, average-case, best-case, variance, ...). We have not included all those results in this article.

## References

ACM ICPC, *ACM International Collegiate Programming Contest*. `http://www.acm.org/contest/` (accessed September 2002).

Blum, M., R.W. Floyd, V.R. Pratt, R.L. Rivest and R.E. Tarjan (1972). "Time Bounds for Selection", *JCSS* **7**, 448–461.

Horváth, G. (2001). *Task* CHAIN *for CEOI 2001*.
`http://ceoi.inf.elte.hu/tasks/chain.html` (accessed September 2002).
*International Science Olympiads*, Verhoeff, T. (Ed).
`http://www.scienceolympiads.org/` (accessed September 2002).
IOI, *International Olympiad in Informatics*.
`http://www.IOInformatics.org/` (accessed September 2002).
Knuth, D.E. (1998). *The Art of Computer Programming: Sorting and Searching*, Vol. 3, 2nd Ed., Addison–Wesley.
Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. Springer.
Verhoeff, T. (1995). The Lost Group Chart and Related Problems. *Simplex Sigillum Veri, A Liber Amicorum for Prof. Dr. F.E.J. Kruseman Aretz*. Faculty of Mathematics and Computing Science, Eindhoven University of Technology. 308–313. `http://www.win.tue.nl/ wstomv/publications/kruseman.pdf` (accessed September 2002).

**G. Horváth** is an associate professor in computing science at the University of Szeged, Hungary. He obtained his PhD (Functor State Machines, 1981) from the University of Szeged. His specialty is algorithms and data structures. He chaired the Scientific Committee of IOI 1996 in Hungary and contributed tasks to various competitions. From 1999 to 2001 he was a member of the IOI Scientific Committee.

**T. Verhoeff** is an assistant professor in computing science at Eindhoven University of Technology (TUE), the The Netherlands. He obtained his PhD (A Theory of Delay-Insensitive Systems, 1994) from TUE. His current reseach area is software construction. He chaired the Scientific Committee of IOI 1995 in The Netherlands and contributed tasks to various competitions. In 1999, he was Finals Directors of the ACM ICPC World Finals held in Eindhoven, The Netherlands. He chairs the IOI Scientific Committee since 1999.

# Viduriniosios vertės radimas pagal tarptautinės informatikų olimpiados sąlygas

Gyula HORVÁTH, Tom VERHOEFF,

Straipsnyje aptariami tarptautinių informatikos olimpiadų uždavinių sprendimo moksliniai principai. Šios olimpiados vyksta kasmet, jose dalyvauja daugelio pasaulio šalių bendrojo lavinimo mokyklos mokiniai. Olimpiadoje sprendžiami algoritminio pobūdžio uždaviniai, joms rašomos programos. Testuojama iš anksto parinkus daug įvairių kontrolinių duomenų. Straipsnio autoriai išsamiai nagrinėja vieną uždavinį äVidutinė vertė", kurį jie buvo sukūrę ir pateikę 2000 metų olimpiadai, vykusiai Kinijoje. Uždavinys paprastas: duotas nelyginis skaičius objektų, kurių visos vertės skirtingos; reikia sukurti efektyvų algoritmą viduriniosios vertės objektui rasti, vartojant tik vieną operaciją û funkciją, kuri randa viduriniosios vertės objektą iš trijų duotų. Šis uždavinys lengvai formuluojamas ir suprantamas. Tačiau jo algoritmas gana sudėtingas ir įdomus. Nagrinėjamas uždavinio sprendimas atsižvelgiant į olimpiados reikalavimus. Aptariamos problemos, susijusios su šiuo uždaviniu. Taip pat pateikiamos idėjos panašaus pobūdžio uždaviniams spręsti.