The Acceleration of the Determination of the Median of Nested Subarrays Using Two Binary Pyramids

Alexander Shportko[1'[0000-0002-4013-3057]'], Veronika Shportko[2'[0000-0002-9460-0781]']

[1]Department of Information Systems and Computing Methods, Academician Stepan Demianchuk International University of Economics and Humanities, 4, Acad. S. Demianchuk Str, Rivne, Ukraine
[2]Software Department, Lviv Polytechnic National University, 12, Bandera Str, Lviv, Ukraine
ITShportko@ukr.net, veronikashportko@gmail.com

Abstract. A new method for determining the median of the array and subarrays using two binary pyramids is described. The duration of determining the medians for different types of arrays and continuous subarrays of by both the traditional algorithms and the proposed method is analyzed. The C# program snippets for the implementation of the algorithms for determining medians by the investigated methods are presented. It is shown that to determine the medians of different arrays and unrelated subarrays, it is advisable to use the Hoare's partition instead of the known sorting methods. To identify the median of sequence of nested continuous subarrays, the method of two pyramids should be used. To find the median of neighboring subarrays of the same length, it is better to use the binary search in their sorted analogues. According to the results of experiments, the usage of the proposed method of two binary pyramids allows to accelerate the determination of the median of embedded continuous subarrays, generated randomly, in more than 10 times.

Keywords: array median, subarray medians, binary search and inclusion, method of two binary pyramids.

1 Introduction

As it is known, in statistics to analyze economic indicators the median of the array is used more frequently than arithmetic mean of all elements of the array, their minimum or maximum values. Let us remind you that in statistics, the median is a value that is in the middle of a series of values in ascending or descending order. The median divides a sequence of values into two equal parts [1], so to determine the median of an array, you must first order its elements, and then, if the number of these elements is odd, you need to select the value of the central element, and if even - then calculate the arithmetic mean of the two central elements. The median of the array characterizes economic activity more objectively. For example, if in a firm there are 9 employees who receive a salary of 5000 UAH per month and one employee who earns 25000 UAH for a month. The average employee of this company earns not 15000 UAH and not 7000 UAH, but still 5000 UAH per month. Therefore, it is not

Copyright © 2020 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2604/paper72.pdf

surprising that in recent times, tasks to accelerate the calculation of the median of array occur more frequently. So, let's explore ways of accelerating the definition of medians for unbound arrays, for nested arrays, and for continuous fixed-length arrays that start from adjacent elements. Fragments of the programs for demonstrating the logic of the algorithms of the proposed methods are in C # programming language [2], since today it is one of the most popular programming languages.

2 The traditional way to determine the median of the array by Hoare's partition

Let us find the median of the array $X = \langle x_0, x_1, \ldots, x_{N-1} \rangle$, where $N \geq 1000$ – is the number of its elements. We will index the array elements from zero, as it is in all C programming languages. Obviously, in the process of determining the median, we will need to use additional memory to sort the copies of array elements or store dynamic structures (pyramids, binary trees) so for not to distort the original array. It is also clear that a standard method of sorting of a programming language can be used (for example, in C# there is the Array.Sort() method) or one of the fast sorting methods [3] can also be used to find the median of the array, and then select the elements that will then be in the middle. But this calculation will be time-consuming because during this process all the elements of the array will be analyzed, but we only need to know what values will be after sorting in the middle. Therefore, in practice, the C. A. R. Hoare's partition is used to determine the medians of the array [4]. This method works according to the "Divide and conquer" principle: between the elements of the array the pivot element is selected and all elements that are not larger than it move in the array to the left of this element, while smaller elements are moved to the right. After permutations if the pivot element is in the middle of the array (for odd-length arrays) or in one of the central positions (for even-length arrays), then the median of the array is calculated with this pivot element. Otherwise, if the support element is on the right side from the center, then the partitioning of the left side is continued, otherwise, the elements placed to the right of the pivot element are analyzed. The function for implementing this method can be the following:

```
static double MedianaHoare(double[] h, int len) {...
if (len == 0) return 0;
if (len == 1) return t[0];
if (len == 2) return (t[0] + t[1]) / 2;
if (len%2==0) //the number of elements is odd
 {indexMaxMed = len/2; indexMinMed = indexMaxMed - 1;}
else
 indexMaxMed=indexMinMed = len/2;
//the main partition cycle
i = 0; j = len - 1; //first we break the whole array
while (true)
{//selecting the new pivot element
```

```
indexPivot = i + R.Next(j - i);
pivot = t[indexPivot];
t[indexPivot] = t[i];
t[i] = pivot; //the pivot element is written
//at the beginning of the fragment
i1 = i + 1; j1 = j; //the borders of the unordered part
while (i1 < j1)
{while (i1 < j1 && t[i1] < pivot)
  i1++; //looking for not smaller element on the left
 while (t[j1] > pivot)
  j1--; //looking for not larger element on the right
 if (i1 < j1)
  {//rearranging the elements that violate ordering
   prom = t[i1]; t[i1++] = t[j1]; t[j1--] = prom; }}
if (t[j1] > pivot) j1--; //move to the not smaller item
//returning the pivot element to the ordered part
t[i] = t[indexPivot = j1]; t[indexPivot] = pivot;
if (indexPivot==indexMaxMed)
//a pivot element is near the center
 {if (indexMinMed!=indexMaxMed)
  //the pivot element is to the right of the center
  {//finding the maximum on the left side
   max = t[i]; index = i;
   for (k = i+1; k <= indexMinMed; k++)
    if (t[k] > max) {max = t[k]; index=k; }
   //we place the maximum to the left of the center
   if (index != indexMinMed)
    {t[index] = t[indexMinMed]; t[indexMinMed] = max; }}
   break; }
if (indexPivot==indexMinMed)
 {if (indexMinMed!=indexMaxMed)
  //the pivot element is to the left of the center
  {//finding the minimum on the right side
   min = t[indexMaxMed]; index = indexMaxMed;
   for (k = indexMaxMed+1; k <= j; k++)
    if (t[k] < min) {min = t[k];  index=k; }
   //we put the minimum to the right of the center
   if (index!=indexMaxMed)
    {t[index]=t[indexMaxMed]; t[indexMaxMed] = min; }}
   break; }
//the support element is not near the center -
//we move on to the next iteration
if (indexPivot>indexMaxMed) j=indexPivot-1;
else i=indexPivot+1; }
if (indexMinMed==indexMaxMed) //for odd length arrays
```

```
 return t[indexMinMed]; //we return the central element
//else we return the mean of the two central elements
return (t[indexMinMed]+t[indexMaxMed])/2; }
```

Then it is enough to output the median

```
Console.WriteLine(MedianaHoare(x, N));
```

The computational complexity (in terms of the number of comparisons) of the implementations of this method depends on the positions of the supporting elements: if the value of this element at each iteration is contained approximately within the analyzed fragment of the array, then the average computational complexity will be $2N$ [5], since for the next iteration it is necessary to compare each element of the fragment with the pivot element. But if at each iteration the support element is minimum/maximum value of its fragment, then in the right/left part of it after transformation all other elements will move and the length of the fragment for the next iteration will be reduced by only one element. In this case, the iteration depth will be N-1 and therefore the overall computational complexity of the algorithm will be $O(N^2)$ [5]. The choice of such an pivot element is not unlikely, especially when its values are taken from the first or last element of the fragment, and the array itself is pre-sorted. Even the author of the QuickSort method and the considered Hoare's partition have emphasized the importance of choosing the correct pivot element [4]. There he suggested two options for forming the value of a reference element: either to set it to the median of a subset of elements (for example, from the first, last and middle element of a fragment), or to select it randomly among the elements of a fragment. And if for the first variant of formation it is still possible to pick up elements of an array so that the computational complexity of sorting is $O(N^2)$, then it is almost impossible for the second option. That is why in practice, the pivot element is often chosen randomly, as it is in the above implementation. Today, some other methods of finding the median of the array that have a theoretical computational complexity of $O(N)$ [5; 6] are also developed, but in practice their implementation is slower than the Hoare's partition [5].

3 Determining the median of arrays by Hoare's partitions

We show below that Hoare's partition is not always the most efficient solution for determining the median of subarrays. Consider, for example, the usage of this partition to determine the median of continuous nested subarrays. Let such arrays not to begin with a zero element: $Y_i = \langle x_0, x_1, \ldots, x_{i-1}, x_i \rangle$, $i = \overline{0, N-1}$. For example, for the array X=<44, 55, 12, 42, 94, 18, 6, 67> [7] such subarrays will be Y_0=<44>, Y_1=<44, 55>, Y_2=<44, 55, 12>, ..., Y_7=X. In fact, the next subarray is derived from the previous subarray by supplementing it with another element. As it was noted above, the Hoare's partition arranges the elements of fragments relative to the pivot ele-

ments, forming an "almost ordered" subarray. Therefore, to find the median of another subarray by this method, it is advisable to use not the subarray itself, but the result of Hoare's partition for the previous subarray by supplementing it with the last element:

```
for (i=0; i<N; i++)
{array[i]=x[i]; //supplement with a new element
 Console.WriteLine(MedianaHoare(array, i)); }
```

This option of finding the median for nested subarrays Y_i has a significant drawback - the Hoare's partition is applied to every subarray every time. Since the average computational complexity of finding the median by Hoare's iterative partitions is twice the size of the subarray, the complexity of calculating the median of all such nested subarrays will be $2+4+6+\ldots+2N = N^2 + N$ and, as it will be shown below, this is not the best option.

Let us now explore the possibilities of using the Hoare partition to determine the median of adjacent continuous subarrays $Z_i = \langle x_i, x_{i+1}, \ldots, x_{i+subN-1} \rangle$, $i = \overline{0, subX-1}$, where *subN* is fixed length of the arrays, *subX* – their number ($subN \le N, subX \le N - subN + 1$), that is what we *consider* neighboring contiguous subarrays of the same length starting with adjacent elements of the input array *X*. For example, for array *X*=<44, 55, 12, 42, 94, 18, 6, 67> adjacent three-element arrays will be Z_0=<44, 55, 12>, Z_1=<55, 12, 42>, Z_2=<12, 42, 94>, ..., Z_5=<18, 6, 67>. In fact, the next neighboring subarray is derived from the previous subarray by removing the first element and supplementing it with the next element. Therefore, having an "almost ordered" subarray after processing the previous subarray by Hoare's partition and replacing the element to be removed with a new element and again to use iterative Hoare's partitions are enough to determine the median of the next neighboring subarray. But the point is that after the Hoare's partition, the elements of the array are not sorted, and that is why the search of the element to remove x_{i-1} should be consistent, not binary. The only thing you can do to speed up this search is to compare it x_{i-1} with the median of the previous subarray, and if this value is less than the previous median, the search for the element to be removed should be conducted from left to right and otherwise from left to right. As experiments have shown, for arrays generated randomly, choosing the direction of such linear search accelerates the median definition by an average of 8%, because then the computational complexity of search of the element to be extracted will not exceed *subN*/2. A fragment of a program to determine the median of adjacent subarrays with a given length can be the following:

```
for (j = 0; j < subN; j++) //form a zero subarray
 array[j] = x[j];
for (i = 0; i < subX; i++) //loop on arrays
{if (i > 0) //replace the following arrays x_{i-1} into x_{i+subN-1}
  //it is more profitable to look from left to right
```

```
  if (x[i - 1]<medMethod)
   array[indexSearchLeftToRight(array, x[i - 1])] =
    x[i + subN - 1];
  else
   array[indexSearchRightToLeft(array, x[i - 1])] =
    x[i + subN - 1];
 Console.WriteLine(MedianaHoare(subMas, subN)); }
```

The option of determining the median for adjacent subarrays Z_i has the same disadvantage as for nested subarrays. Hoare's partition applies to each subarray. The average computational complexity of determining these medians is $2.5 \times subN \times subX$.

4 Determining the median of arrays using binary search

It is common knowledge that the binary search of the element in subarray is faster than linear search [3], but to perform the binary search it is necessary for the subarray to be sorted. Let's first consider the mechanism of applying the binary search to determine the median of **nested** arrays. Let Y_0 contains only one element, which is its median. This subset is already sorted. If the initial array Y_0 contained more elements, then its copy should be sorted by one of the known sorting algorithms and then the median should be determined. But for a faster (relatively Hoare's partition) search for the median of the following nested arrays Y_i , $i = \overline{1, N-1}$ we apply **the binary inclusion [7] of a new element** x_i in the sorted subarray from the elements of the previous subarray. To perform each binary inclusion, we should firstly find the insert index *indexInsert* of the new element in the pre-sorted subarray after small elements, then move to the right all the elements from the *indexInsert* index to the end of the subarray, and then insert x_i into position *indexInsert*:

```
//binary search function within a given
//position range to include an item
static int BinarySearchIndexToIncludeElement(double[] t,
           int left, int right, double element)
{while (left<right) //while as there is a search interval
 {int j = (left + right) / 2;//middle index
  if (t[j] <= element) left = j + 1; //fold to the left
  else right = j; } //we reject the case
 return left; }

//the procedure for printing the median of a subarray
//with the specified length
static void WriteMediana(double[] t, int len)
{if (len % 2==0)
  Console.WriteLine((t[len/2]+t[len/2-1])/2);
 else Console.WriteLine(t[len/2]); }
```

```
...
//determining the median of nested arrays
array[0]=x[0]; Console.WriteLine(x[0]);
for (i = 1; i < N; i++) //index of the element to include
{indexInsert =
  BinarySearchIndexToIncludeElement(array, 0, i, x[i]);
 if (indexInsert < i)
  //you must move the items to the right to enable
  for (k = i - 1; k >= indexInsert; k--)
   array[k + 1] = array[k];
 array[indexInsert] = x[i]; }
 WriteMediana(array, i); }
```

The given fragment of the program implements the sorting of array by binary inclusions [7], but additionally after each inclusion displays the median of the received subarray. The computational complexity of such inclusions on comparison operations is $N(\log N - \log e \pm 0.5)$ [7], that is much less than the complexity of the Hoare's partitioning method. The weak point of the binary inclusion is the need to move for each subarray of the group of elements from the position of inclusion to its end.

Let us show how to apply the binary search to determine the median of adjacent arrays Z_i. As it was noted above, Z_i is obtained from Z_{i-1} remove the first item x_{i-1} and adding a new element $x_{i+subN-1}$ ($i = \overline{1, subX - 1}$). To speed up the determination of the median by binary search, these adjustments must be made not on unordered arrays. Z_{i-1}, but on their sorted copies $\tilde{Z}_{i-1}$. Of course, you could first do a binary search and remove the element x_{i-1}, and then make the binary inclusion of the new element $x_{i+subN-1}$, but then you would have to move the elements twice to the end (or top) of the sorted subarray. Therefore, to determine the median of adjacent arrays Z_i we apply the following algorithm:

1. Sort Z_0 by one of the known algorithms of sorting and find the median of the resulting subarray;
2. For all subsequent neighboring subarrays ($i = \overline{1, subX - 1}$) repeat steps 3-6;
3. By binary search find in the sorted subarray the index of the element to be removed x_{i-1} and write it into a variable *indexDel*;
4. Find the index of insertion of the new element in a sorted subarray by binary search $x_{i+subN-1}$ after not smaller elements and write it into a variable *indexInsert*;
5. If *indexInsert* > *indexDel*, then move the sub-elements from the position *indexDel*+1 to the position *indexInsert*-1 one item to the left and paste $x_{i+subN-1}$ into the sorted subarray into the position *indexInsert*-1;
6. Otherwise move the sub-elements from the position *indexDel*-1 to the position *indexInsert* one item to the right and paste $x_{i+subN-1}$ into the sorted subarray into position *indexInsert*.

Moving elements of the sorted subarray only from the position of deletion to the position of inclusion, but not twice from each of these positions to the end of the subarray, accelerates the determination of the median of neighboring arrays by more than twice. The program to determine these medians may be the following:

```
//function of binary element search in the array
static int IndexBinarySearchElement
             (double[] t, double element)
{int left = 0, right = x.Length-1, j = right;
 while (left < right)
  {j = (left + right) / 2;
   if (t[j] < element) left = j + 1;
   else if (t[j] > element) right = j - 1;
        else break; }
 if (left == right) return left;
 return j; }
...
//determining the median of the original subarray
for (j = 0; j < subN; j++)
 array[j] = x[j];
Array.Sort(array); WriteMediana(array, subN);
//determining the median of the next adjacent arrays
for (i = 1; i < subX; i++)
 {indexDel = IndexBinarySearchElement(array, x[i-1]);
  indexInsert = BinarySearchIndexToIncludeElement(
                 array, 0, subN-1, x[i+subN-1]);
  if (indexInsert > indexDel)
   {for (k = indexDel + 1; k < indexInsert; k++)
     array[k - 1] = array[k]; //move to the place removed
    array[indexInsert-1]=x[i+subN-1]; }
  else
   {for (k = indexDel; k > indexInsert; k--)
    arrayx[k] = array[k-1];
    array[indexInsert]=x[i+subN-1]; }
  WriteMediana(array, subN); }
```

The computational complexity of such definitions of the median by comparison operations is $2 \times subX \times \log subN$, which is much less difficult to find using Hoare's partitions. The disadvantage of this method and the previous algorithm is the need to move for each subset of the group of elements from the removal position to the inclusion one. The following algorithms are essentially aimed at reducing the number of moving elements, which can accelerate the determination of the median.

Generally speaking, the average number of comparisons to perform two binary searches (positions for extraction and insertion) in the sorted subarray while replacing elements is $2 \times \log subN$. Defining the median of the same subarray by Hoare's parti-

tion requires $2 \times \log subN$ comparisons. Therefore, if the neighboring arrays are less than by $subN / \log subN$ elements, then to determine their median, it is advisable not to use the Hoare's partition, but to sort the initial subarray and sequentially perform binary searches in it.

5 Finding the median of arrays and subarrays using two pyramids

From the sequence of elements of the input array X we construct two binary pyramids [8] $A = \langle a_0, a_1, \ldots, a_{\lfloor N/2 \rfloor - 1} \rangle$ and $B = \langle b_0, b_1, \ldots, b_{\lfloor N/2 \rfloor - 1} \rangle$ with the same size, so that the pyramid A is non-ascending

$$a_m \geq a_{2m+1}, a_m \geq a_{2m+2}, \quad (1)$$

and pyramid B is non-descending

$$b_m \leq b_{2m+1}, b_m \leq b_{2m+2} \quad (2)$$

and $a_i \leq b_j \; (i, j = \overline{0, \lfloor N/2 \rfloor - 1})$, so that the elements of pyramid B must be smaller than the elements of pyramid A. The numbers of elements in these pyramids must be the same after each step of sequentially processing of the elements of array X. An example of such pyramids is given in Fig. 1. From the principles of construction of these pyramids, it follows that if N is even, then the median of the array X will be equal to the arithmetic mean of the elements of the vertices of the pyramids a_0 and b_0, and if it is odd, the median will be a mean of a_0, b_0 and x_{N-1}. We will calculate the medians of nested and adjacent arrays on the same principle.

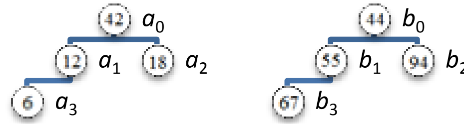

Fig. 1. The non-ascending pyramid A=<42, 12, 18, 6> and the non-descending pyramid B=<44, 55, 94, 67>, built for array X=<44, 55, 12, 42, 94, 18, 6, 67>

Two auxiliary procedures are used to form pyramid A. The first of them inserts the value of *item* that is not less than each of the elements of this pyramid into the top a_0, previously moving the existing items in the direction of the new node:

```
static void InsertTopA(double item)
 {int j=countNH-1; //index of new node
  while (j>0) //moving items until we reach the top
   {arrayA[j]=arrayA[indexTop[j]]; j=indexTop[j]; }
  //inserting a new value into the top
```

```
  a0=arrayA[0]=item; }
```

The second procedure inserts the value of *item* in pyramid *A*, starting from the specified node, so as not to violate the principle (1). To do this, the *item* is first alternately rearranged with higher values until they are less than *item*, and then with **values of** subordinate nodes [7] if they are larger than the *item* (its insertion index actually changes):

```
static void InsertA(int index, double item)
 {//lifting at higher nodes
  while (index>0 && arrayA[indexTop[index]]<item)
   {arrayA[index]=arrayA[indexTop[index]];
    index=indexTop[index]; }
  //lowering towards larger lower-level values
  int indexBottom;
  while (true)
   {indexBottom=indexLeft[index];
    if (arrayA[indexBottom+1]>arrayA[indexBottom])
     indexBottom++;
    if (item<arrayA[indexBottom])
     {arrayA[index]=arrayA[indexBottom];
      index=indexBottom; }
    else break; }
  arrayA[index]=item; }
```

Only the second procedure is sufficient to form the pyramid *A*, but the first procedure inserts the value that is not less than each of the elements of this pyramid into its top without performing additional comparisons, and therefore accomplishes this task much faster. The two procedures for inserting values into pyramid *B* are similar to the above mentioned procedures, but in the second one, the comparisons of the elements are reversed to ensure that the ordering principle is fulfilled (2).

Let us now give a verbal description of the algorithm of sequential formation of pyramids *A* and *B*. Since these pyramids should always contain the same number of elements, the elements of the input array *X* will be sequentially treated in pairs. In the first step of the algorithm we assign $a_0 \leftarrow \min(x_0, x_1), b_0 \leftarrow \max(x_0, x_1)$. The next steps are for the other pairs $\left(i = \overline{1, \lfloor N/2 \rfloor - 1}\right)$. Firstly, we calculate $minPair \leftarrow \min(x_{2i}, x_{2i+1})$ and $maxPair \leftarrow \max(x_{2i}, x_{2i+1})$, and then insert them into the pyramids. Six variants of ordering of the vertices of the pyramids a_0, b_0 and *minPair*, *maxPair* (Figure 2) are possible and, accordingly, six options for inserting the last two variables can take place.

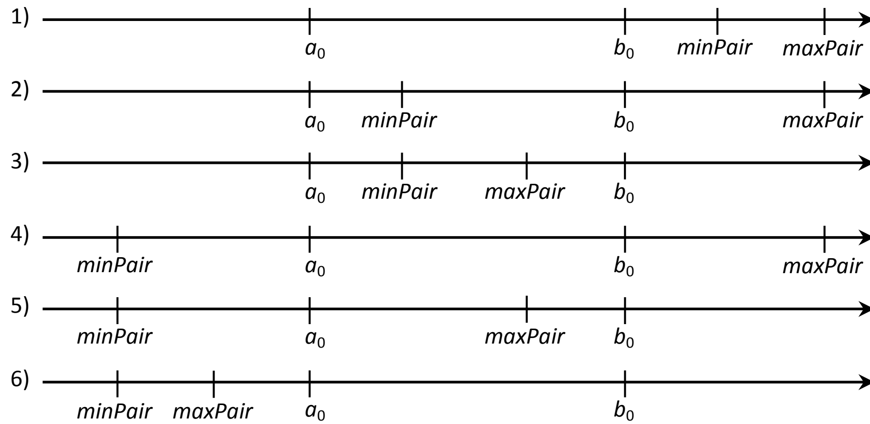

Fig. 2. Options for ordering the values of the vertices of the pyramids a_0, b_0 and the minimum and maximum values of the pair of following array elements X

The code snippet for implementing these inserts could be the following:

```
countNH++;//increased the number of nodes in the pyramids
if (minPair>b0) //first ordering
 {InsertB(countNH-1, maxPair);
  InsertTopA(b0);
  InsertB(0, minPair);
  b0=arrayB[0]; }
else
 if (minPair>=a0)
  {InsertTopA(minPair);
   if (maxPair>b0) //the second option
    InsertB(countNH-1, maxPair);
   else //the third option
    InsertTopB(maxPair); }
 else //minPair<a0
  {InsertA(countNH-1, minPair);
   if (maxPair>b0) //the fourth option
    InsertB(countNH-1, maxPair);
   else
    if (maxPair>=a0) //the fifth option
     InsertTopB(maxPair);
    else //the sixth option
     {InsertTopB(a0);
      InsertA(0, maxPair);
      a0=arrayA[0]; }}
```

As it follows from the implementation of the algorithm, the insertion for the third-order variant is the most quickly performed, when the next values of the input array are placed between the vertices of the pyramid. In this case, the less value is inserted

into the top of pyramid *A* and larger value is inserted into the top of pyramid *B* without additional comparisons. But this variant is extremely rare. The results of experiments on arrays generated randomly showed that the relative frequency of occurrence of the first and sixth variants orderliness is about 25 %, the fourth – 49.9999%, the second and fifth – 0.00004%, and the third – only 0.00002%. An example of the results of the first three steps of the pyramid forming algorithm is shown in Figure 3, and the fourth one is shown in Figure 1. Here, in the second step of pyramid formation, the sixth variant of insertion is implemented, and in the third and fourth steps, the fourth insertion options.

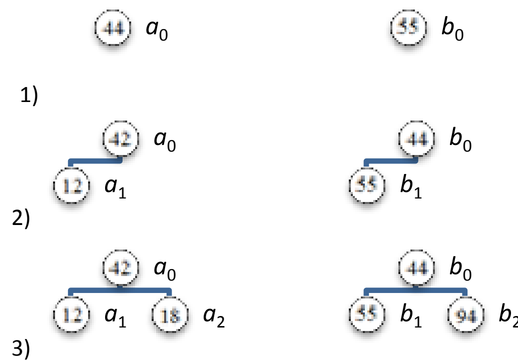

Fig. 3. The results of the first three steps of pyramid formation *A* and *B* of array elements *X*=<44, 55, 12, 42, 94, 18, 6, 67>

In the process of building pyramids *A* and *B*, each element of the input array *X* is actually inserted into one of these two pyramids, and therefore the average computational complexity of this algorithm for determining the median arrays and nested subarrays, both by comparison operations and by the number of assignments is close to $N \log N$. The same calculation is done using binary search with approximately the same number of comparisons, but with much higher number of assignments.

Unfortunately, the use of two binary pyramids of the same size is unsuitable for determining the median of neighboring arrays, because when passing to a neighboring arrays Z_i you will need to remove the item from the pyramids x_{i-1} and then insert an item into them $x_{i+subN-1}$. And if the insertion requires on average only $\log subN$ operations of comparisons, the search for the element to be extracted can run through almost all over the pyramid *A* or *B*, so the average computational complexity of determining the median of adjacent subarrays using two pyramids by comparison operations is $subX \times (subN / 4 + \log subN)$, which exceeds the complexity of determining the same medians using binary search.

6 Experimental results

Let us first analyze the duration of the determinations of the median arrays of 10 million real numbers by the algorithms of different methods (Table 1). We implemented the algorithms of the considered methods in Microsoft Visual Studio 2017 program in C# programming language [2]. To implement the quick sort, we used the standard method of the Array.Sort(). Testing was carried out on a computer with an Intel Pentium 4 processor with 3 GHz clock speed and size of RAM 4Gb.

Table 1. The duration of the determinations of the median arrays of 10 million real numbers algorithms of different methods, ms

Method of determining the median	Array variant			
	Generated randomly	Sorted in ascending order	Sorted in descending order	Of the same elements
Quick sorting	3906	1328	2153	2086
Binary inclusion	over 22.8 million	6992	over 24 million	6953
Hoare's partition	**625**	**273**	**328**	**602**
Two binary pyramids	5718	9521	8833	5030

We see that it is expedient to use Hoare's partition to determine the median of large arrays. It is faster than sorting algorithms because it does not order the elements of the array completely and is also faster than the algorithm of the method of two binary pyramids, because it does not form common hierarchical structures.

Now let us compare the duration of the definitions of the median of 10 thousand nested sub-arrays of real numbers by algorithms of different methods (Table 2).

Table 2. The duration of finding the median of 10 thousand nested arrays of real numbers algorithms of different methods, ms

Method of determining the median	Using pre-ordered data	Array variant			
		Generated randomly	Sorted in ascending order	Sorted in descending order	Of the same elements
Quick sorting	No	11414	4714	6605	6145
	Yes	4868	3877	6233	5498
Hoare's partition	No	4118	2678	2477	2516
	Yes	1070	1063	1102	828
Binary inclusion	Yes	250	16	469	23
Two binary pyramids	Yes	**23**	**16**	**16**	**16**

We see that to determine the median of nested subarrays, it is advisable to modify the processing data of a previously nested subarrays rather than process their data. As it was predicted, the method of two binary pyramids proved to be the most effective and

stable for determining the median of such arrays, since it uses on average $N \log N$ comparisons and the same number of assignments.

Finally, let us analyze the duration of finding the median of 10 thousand adjacent subarrays of real numbers of 5 thousand elements each by algorithms of different methods (Table 3).

Table 3. The duration of the determinations is a median of 10 thousand adjacent subarrays of real numbers 5 thousand elements each by algorithms of different methods, ms

Method of determining the median	Using pre-ordered data	Array variant			
		Generated randomly	Sorted in ascending order	Sorted in descending order	Of the same elements
Quick sorting	No	12790	4678	6544	8192
	Yes	6258	5757	6047	7304
Hoare's partition	No	5125	1117	1453	3609
	Yes	771	734	867	1594
Binary search	Yes	**234**	**591**	**606**	**281**
Two binary pyramids	Yes	477	1138	906	16

As for nested arrays, we see that to determine the median of the next neighbor arrays, it is necessary to adjust the data of the previous sub-array (for sorting algorithms it is sorted counterpart; for Hoare's partition it is the result of permutations of the previous array; for two pyramids method it is previous hierarchical structures) and not process the elements of the next subarray first. As it was predicted, the method of two pyramids does not show the best results for the neighboring arrays, since when searching for an element to extract it can analyze one of the two pyramids completely. The most effective method for determining the median of adjacent arrays was the binary element search method for extraction and insertion, since it uses only $2 \times subX \times \log subN$ comparisons on average and moves elements only between extraction and insertion positions.

7 Conclusions

1. There are no universal methods for determining the medians that are effective for all the sequences of arrays or subarrays - it all depends on the number of elements that differ.
2. To determine the median of individual arrays and adjacent subarrays that differ by more than $subN / \log subN$ elements, it is advisable to use Hoare's partition instead of the known sorting methods, since it rearranges only individual elements and does not order the entire array.
3. While finding the median of adjacent subarrays that differ by no more than $subN / \log subN$ elements, it is advisable to sort the initial arrays and then sequen-

tially perform the binary search for the deletion and insertion positions in it, move the elements between them in the direction of the deletion position and insert the new element.

4. The method of two binary pyramids should be used to determine the median sequence of nested subarrays, since its implementation perform on average $N \log N$ comparisons and as many assignments.

In the future researches to accelerate definitions medians neighboring subarrays we plan to use binary trees with a fixed height, which are expected to reduce the number of items being moved.

References

1. Malistov A.: The Search of medians arrays for linear time. In: Math education, Vol. 21, pp. 265–270 (2017) (In Ru).
2. C# Language Specification. Standard ECMA-334, 5-ed. ECMA International (2017).
3. Knuth D.: The Art of Computer Programming, Vol. 3. Sorting and Searching, 2-ed. In: : Addison Wesley Longman, Massachusetts, 791 p. (1997).
4. Hoare C. A. R.: Quicksort. In: The Computer Journal, No 5 (1), pp. 10-16 (1962).
5. Cohen R.: My Favorite Algorithm: Linear Time Median Finding https://rcoh.me/posts/linear-time-median-finding/ (accessed by Jan 15, 2018).
6. Blum M., Floyd R. W., Pratt V., Rivest R. L., Tarjan R. E.: Time bounds for selection. In: Journal of computer and system sciences, Vol. 7, No 4. pp. 448–461 (1973).
7. Vlasyuk A.: Workshop on Programming in the Turbo Pascal environment. Vol. 1. In: NUWEE, Rivne, 179 p. (2005) (In Ukr.)
8. Williams, J. W.: Algorithm 232 – Heapsort. In: Communications of the ACM, No 7 (6), pp. 347–348 (1964).
9. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms, Third Edition. In: Williams, Moscow, 1328 p. (2014) (In Ru).