Fundamentals of Programming 2

The Quicksort and Heapsort Algorithms

Arkadiusz Chrobot

Department of Computer Science

May 14, 2020

Notes

Outline

Notes

Introduction

This lecture is about two sorting algorithms, which are related to previously discussed topics: the "divide and conquer" method, the recursion and trees. The first of those algorithms is the Quicksort, the second one is the Heapsort.

Notes

Introduction

Before those algorithms will be described, the functions and other element of the source code are introduced that are common for all programs presented in this lecture.

Notes

Introduction
Header Files and Array Type

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  typedef int int_array_type[10];
```


Notes

Introduction
Header Files and Array Type

The program uses functions declared in the included header files to print the content of an array on the screen and to initialize the PRNG which fills the array with numbers. In the last line of the source code presented in the previous slide a type is introduced (5th line) that defines an array of 10 elements of the `int` type.

Notes

Introduction
The `fill_array()` Function

```
1  void fill_array(int_array_type array)
2  {
3      int i;
4      srand(time(0));
5      for(i=0;i<sizeof(int_array_type)/sizeof(array[0]);i++)
6          array[i] = -10+rand()%21;
7  }
```


Notes

Introduction
The `fill_array()` Function

The function presented in the previous slide fills an array with integers ranging from -10 to 10. Those numbers are chosen randomly. The array is passed by the parameter of the `int_array_type` type, which is defined at the beginning of the program. The `sizeof` operator applied to such a type returns the number of bytes that an array of this type would occupy in the memory. The number of the array elements can be calculated by dividing this value by the size of the array first element (5th line). Such an expression is applied in the condition of the `for` loop.

Notes

Introduction
The print_array() Function

```
1  void print_array(int_array_type array)
2  {
3      int i;
4      for(i=0;i<sizeof(int_array_type)/sizeof(*array);i++)
5          printf("%d ",array[i]);
6      printf("\n");
7  }
```


Notes

Introduction
The print_array() Function

The `print_array()` function displays the content of the array passed by the parameter and moves the cursor to the next line on the screen. It differs a little from similar functions presented in the previous semester. The first difference is the type of the parameter. It is the `int_array_type` type defined at the beginning of the program with the use of the `typedef` keyword. The second one is the `printf()` function used for moving the cursor to the next line after the `for` loop stops. The argument for this function is a string that contains only the new line character (6th line). The last difference is the condition for the loop. It is very similar to the one used in the `fill_array()` function, but the first element of the array is referenced with the use of the pointer instead of the square brackets (4th line).

Notes

Introduction
The swap() Function

```
1  void swap(int *first, int *second)
2  {
3      int temporary = *first;
4      *first = *second;
5      *second = temporary;
6  }
```


Notes

Introduction
The swap() Function

The definition of the `swap()` function has been described many times in previous lectures. It swaps the values of two variables and in the programs from this lecture it is used for exchanging values of the array elements.

Notes

Quicksort

The Quicksort algorithm was developed by the British computer scientist C.A.R. Hoare and is one of the most efficient sorting algorithms. Although its worst-case time complexity is $\Theta(n^2)$, where n is the number of element, its average and best-case time complexity is $\Theta(n \cdot log_2(n))$. Constants hidden by the asymptotic notation are small. The Quicksort is an in-place sorting algorithm, but its space complexity is $O(n)$. It is a consequence of the fact, that it is a recursive algorithm implemented in a form of a recursive subroutine, hence it intensively uses the call stack. It can be implemented as an iterative subroutine, but it proves to be a challenging task and the iterative implementation isn't more effective than the recursive one. The Quicksort performs unstable sorting.

Notes

Quicksort and "Divide and Conquer"

The Quicksort algorithm can be described using the "Divide and Conquer" method:

Divide: The $A[p \dots r]$ array is partitioned (values of its elements are swapped) into two nonempty parts $A[p \dots q]$ and $A[q+1 \dots r]$, such that a value of each element in $A[p \dots q]$ is not greater than the value of any element in $A[q+1 \dots r]$. The q index is determined by a partitioning subroutine.

Conquer: The two parts: $A[p \dots q]$ and $A[q+1 \dots r]$ are sorted by applying the Quicksort algorithm recursively.

Merge: Since the Quicksort is in-place algorithm, no additional steps are required to merge the sorted parts: the whole $A[p \dots r]$ array is already sorted.

Notes

Quicksort

The `quicksort()` Function

```
1  void quicksort(int_array_type array, int low, int high)
2  {
3      if(low<high) {
4          int partition_index = partition(array,low,high);
5          quicksort(array, low, partition_index);
6          quicksort(array, partition_index+1, high);
7      }
8  }
```


Notes

Quicksort

The `quicksort()` Function

The `quicksort()` function corresponds to the **Conquer** step in the description presented in the previous slide. It doesn't return any value, but has three parameters. The first one is used for passing the array. By the second and third parameters are passed the indices that specify the area of the array that has to be sorted. Initially, when the `quicksort()` function is called, for example, in the `main()` function, this area covers the whole array. Inside the body of the `quicksort()` function the first index (`low`) is compared with the last index (`high`). If the former is less than the latter then there is a part (area) of the array that still needs to be sorted. Otherwise the function exits. If the condition in the 3rd lined is satisfied then the `partition()` function is invoked that reorders the given part of the array and determines the point where this area is partitioned in two smaller parts. Next, the `quicksort()` function is called twice, for each of the new parts separately.

Notes

Quicksort

The quicksort() Function

The first part is sorted by an instance of the `quicksort()` function that deals with elements of the array that have indices raging from the first index (`low`) to the partition index (`partition_index`), including both of them. The second part consists of elements with indices raging from the partition index (excluding) to the last index (`high`), including. The `partition()` function is defined in the program before the `quicksort()` function, but in the slides it is described after the latter.

Notes

Quicksort

The partition() Function

```
1  int partition(int_array_type array, int low, int high)
2  {
3      int pivot = array[low];
4      int i = low-1, j = high+1;
5
6      while(i<j) {
7          while(array[--j]>pivot)
8              ;
9          while(array[++i]<pivot)
10             ;
11         if(i<j)
12             swap(&array[i],&array[j]);
13     }
14     return j;
15 }
```


Notes

Quicksort

The partition() Function

The `partition()` function corresponds to the **Divide** step in the description that uses the "Divide and Conquer" method. It has three parameters, which have the same meaning as the parameters of the `quicksort()` function. The `partition()` function returns a number, which is an index that specifies the partition point of the currently sorted part of the array. In the 3rd line of the function is declared and initialised a variable named `pivot`, that stores so-called *pivot value* which specifies how the part of the array is reordered. In the function, the value of the first element of the sorted part of the array is assumed as the pivot value (line no. 3). In the 4th line of the function are declared and initialised two variables that are used for indexing the sorted part of the array from the beginning (the `i` variable) and from the end (the `j` variable). Please note, that initially both indices specify elements that are *outside* the sorted part of the array.

Notes

Quicksort

Funkcja partition()

The outer `while` loop (6th line) is repeated as long as the the value of the `i` index is smaller than the value of the `j` index, or in other words, until the indices "meet" or "miss" each other. Inside the loop are performed two other `while` loops. The first one (lines no. 7 and 8) traverses the given part of an array starting from the end toward the beginning and searches for an element that has a value equal to or smaller than the pivot value. The second internal loop (lines no. 9 and 10) traverses the same part of the array but from the beginning to the end and searches for an element with a value greater than or equal to the pivot value. Please note, how these loops are implemented. The searching takes place inside the condition statement of the loops. The pre-increment and pre-decrement operators are applied to the indices to avoid accessing elements of the array that are outside of the sorted part or even accessing elements outside the array itself.

Notes

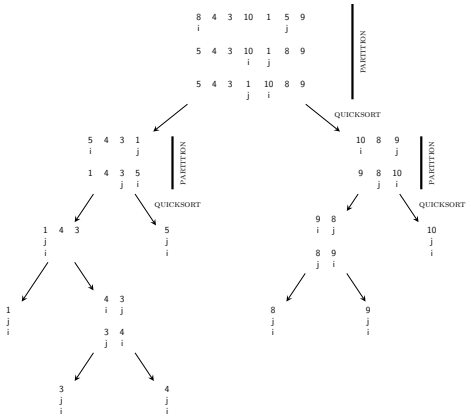
Quicksort

The `partition()` Function

After both internal loops stop, the function performs the conditional statement (11th line) to check whether the `i` index is smaller than the `j` index. If so, then the order of the values in the elements associated with those indices is incorrect, and they have to switch the places. If not, then the outer loop stops and the `j` index specifies a partition point for the sorted part of the array, hence the index is returned by the function (14th line).
In the next slide is a call tree that illustrates how the `quicksort()` function sorts an array that has seven elements that store natural numbers. In the upper part of the tree it is marked which actions are performed by the `quicksort()` function and which by the `partition()` function. In the bottom part of the tree no such description is given, to keep the drawing more legible.

Notes

Quicksort



Notes

The `main()` Function

```
1  int main(void)
2  {
3      int_array_type array;
4      fill_array(array);
5      print_array(array);
6      quicksort(array,0,
7          sizeof(int_array_type)/sizeof(*array)-1);
8      print_array(array);
9      return 0;
10 }
```


Notes

The `main()` Function

An array of the `int_array_type` type is declared in the 3rd line of the `main()` function, which is then initialised with the use of the `fill_array()` function and sorted with the help of the `quicksort()` function. Aside from the array the last function takes indices as arguments, hence the second argument of the function is 0 (the index of the first element). The third one is the number of elements in the array decremented by one (the index of the last element). The number of elements is calculated using the same expression as in the `print_array()` function. After the `quicksort()` function exits, the content of the sorted array is displayed on the screen and the `main()` function also exits returning the 0 value.

Notes

Other Versions

There are several variants of the Quicksort algorithm that can be implemented in a different way than the one already presented. The other frequently used implementation of the Quicksort algorithm is described in the next slides. It's not as effective as the earlier one, but many computer scientists think its more legible, easier to create and less prone to contain mistakes.

Notes

Quicksort

The `quicksort()` Function — Second Version

```
1  void quicksort(int_array_type array, int low, int high)
2  {
3      if(low<high) {
4          int partition_index = partition(array,low,high);
5          quicksort(array, low, partition_index-1);
6          quicksort(array, partition_index+1, high);
7      }
8  }
```


Notes

Quicksort

The `quicksort()` Function — Second Version

There is only one detail that differentiates the definition of the `quicksort()` function presented in the previous slide from the one described earlier. The part of the array that is sorted by the first recursive invocation of the function doesn't include the element specified by the `partition_index` (5th line).

Notes

Quicksort

The `partition()` Function — Second Version

```
1  int partition(int_array_type array, int low, int high)
2  {
3      int pivot = array[low], middle = low, i;
4
5      for(i=low+1; i<=high; i++)
6          if(array[i]<pivot) {
7              middle++;
8              swap(&array[middle],&array[i]);
9          }
10     swap(&array[low],&array[middle]);
11     return middle;
12 }
```


Notes

Quicksort

The partition() Function — Second Version

In this implementation of the `partition()` function only one iteration statement is used and this time it is the `for` loop. Just like in the previous version a few local variables are declared first. The `pivot` variable has the same purpose as in the previous version. The `middle` variable is an index that eventually will specify the element of the sorted part of the array in which the pivot value should be stored. The `i` variable is the loop counter and simultaneously the variable used for indexing the array. The concept of this implementation of the `partition()` function is as follows: the values in the sorted part of the array should be so rearranged that the values smaller than the pivot value ought to be moved to the left and the equal or bigger to the right. Since the value of the first element of the sorted part is chosen as the pivot value then all the other elements should have values greater or equal to the pivot value. If not, then the smaller values should be moved to the left. The rest of the function takes care of it.

Notes

Quicksort

The partition() Function — Second Version

Please note the `for` loop. It iterates over all elements of the sorted part, except the first one. Hence, the `i` index is initialised with a value greater by one than the value of the `low` parameter. The loop stops when the loop counter reaches a value grater than the value of the `high` parameter. Inside the loop, the conditional statement (6th line) checks if the current element of the sorted part of the array stores a value which is less than the pivot value. If so, than the value of the `middle` index is incremented by one and the value of element specified by this index is swapped with the value of the element specified by the `i` index. After the loop stops the value of the first element of the sorted part of the array is swapped with the value of the element specified by the `middle` index (10th line). After that the function returns the `middle` index as the partitioning point for the sorted part of the array.

Notes

Quicksort

The qsort() Function

The Quicksort algorithm is so effective, that the creators of the C language decided to define the `qsort()` function that implements it, as a part of the language standard library. Its prototype is in the `stdlib.h` header file. The function doesn't return any value but has four parameters. The first one, of the `void *` type, is used for passing the array to be sorted. By the second one, the number of array elements is passed and by the third one the size of a single element. The last, forth, parameter is a function pointer that points to a function that compares values of array elements. Its prototype should be as follows:

```
int compare(const void *, const void *);
```

By the parameters are passed pointers to the compared elements. If the value of the first one is greater than the value of the second one then the function ought to return a positive integer number. Otherwise it should return a negative integer number. If the values are equal, the function should return `0`.

Notes

Quicksort

Funkcja qsort()

The way, the `qsort()` is defined allows it to sort any array of any number and type of elements.
The next slides present the definition of a function that compares two elements of an array of elements of the `int` type and show how the `qsort()` function can be invoked.

Notes

Quicksort

The compare_int() Function

```
1  int compare_int(const void *first, const void *second)
2  {
3      return *(int *)first - *(int *)second;
4  }
```


Notes

Quicksort

The compare_int() Function

The definition of the `compare_int()` function is short. It has two pointer parameters named `first` and `second`. The body of the function is basically one statement, in which both pointers are first casted on the `int *` type, then they are dereferenced and the values pointed by those pointers are subtracted from each other. If the result is a negative number, than the first value is smaller than the second one. If the result is positive than the first value is the bigger one. If the result is zero, then they are equal. The result is returned and the function exits.

Notes

Quicksort

The main() Function

```
1  int main(void)
2  {
3      int_array_type array;
4      fill_array(array);
5      print_array(array);
6      qsort((void*)array,
7           sizeof(int_array_type)/sizeof(array[0]),
8           sizeof(array[0]),compare_int);
9      print_array(array);
10     return 0;
11 }
```


Notes

Quicksort

The main() Function

The lines no. 6, 7 and 8 contain the invocation of the `qsort()` function. As the first argument is passed the pointer to the sorted array. It is casted on the `void *` type. The next argument is the number of element of the array, calculated with the use of the same expression which is applied for that purpose in the `fill_array()` function. The size of the first element of the array is passed as the third argument. It can be any element of the array — all have the same size, however the C language standard guarantees that the first element of the array always exists. The pointer to the function that compares elements of the array is passed as the last argument. Please notice, that it is the name of the function.

Notes

Heap

The word *heap* has two meanings in the computer science. It can mean a part of the program memory, where the dynamically allocated variables are created or a binary tree, which has a shape of the full binary tree, or the complete binary tree, and which satisfies the *heap property*. If the heap is the complete binary tree, then the missing nodes in the last level have to be on the right side. The heap is usually *not* implemented in the form of a dynamically allocated data structure, but it is mapped into an array in such a way, that the key of the node is the index of the element and the value of the node is the value of the element. The root is always mapped into the first element of the array. Assuming that the indices of the array start from 1 and that the $index$ denotes an index in the array of a heap internal node then the index of its left child can be calculated using the expression $2 \cdot index$ and the index of the right child with the use of the expression $2 \cdot index + 1$. The parent of any node of the heap, except the root can be calculated using the expression $index/2$, where "/" denotes the integer division.

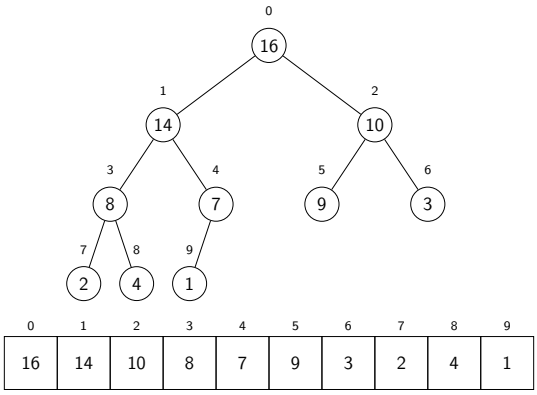
Notes

Heap

The indices of the arrays in the C language start from 0. If such a value is substituted for $index$ then the expressions from the previous slide give incorrect results. There are two solutions for this problem. Either the first element of the array has to be always omitted or the expressions have to be accordingly transformed. In the presented discussion the second possibility is chosen. Thus, the expression for calculating the index of the parent becomes $(index - 1)/2$. The index of the left child is calculated using the following expression: $2 \cdot index + 1$ and for calculating the index of the right child such an expression: $2 \cdot index + 2$ can be applied. In the next slide is an illustration of the heap and its mapping into an array which indices start from zero (so-called *zero-based array*).

Notes

Heap



Notes

Heap

The presented heap is called *max-heap*. The heap property for such a heap is defined as follows: $A[parent(i)] \geq A[i]$, which means that the value of a parent of any node is always greater or equal to the value of the node. The A letter denotes an array. There are also *min-heaps* for which the property is defined as $A[parent(i)] \leq A[i]$. For the rest of the lecture the max-heaps are used. The relation between the heap and the array into which the heap is mapped is given by the following expression: $length(heap) \leq lenght(array)$, where $length$ is the number of the elements of the array or the heap. This expression means that not all of the array elements have to be part of the heap.

Notes

Heapsort

The heaps can be applied for building so-called *priority queues*, but in this lecture their use in the array sorting algorithm, closely related to the selection sort algorithm, is discussed. The algorithm is called the *Heapsort* and just like the Quicksort it performs unstable sorting. When compared with the latter, the Heapsort is slower, but still it is one of the most effective sorting algorithms. Its time complexity for all possible cases is $O(n \cdot log_2(n))$. The Heapsort can be implemented in a recursive (the one is demonstrated in the lecture) or in an iterative form.
The next slides show definitions of functions that calculate indices of the right and left child of a node (the calculation of the parent's index is not applied in this algorithm). Then the function that reestablishes the heap property, the function that builds the heap and finally, the function that sorts the array are presented.

Notes

Heapsort

The `get_left_child_index()` Function

```
1  static inline int get_left_child_index(int index)
2  {
3      return (index << 1) + 1;
4  }
```


Notes

Heapsort

The `get_left_child_index()` Function

The function shown in the previous slide calculates the index of the heap node's left child. To speed up the calculations the bitwise shift left operator is applied instead of the regular multiplication operation. It's possible because the index of the node is multiplied by 2. Additionally, in the function header the `inline` keyword is used which means that the function should be expanded like a macro or optimized in other way by the compiler.

Notes

Heapsort

The `get_right_child_index()` Function

```
1  static inline int get_right_child_index(int index)
2  {
3      return (index << 1) + 2;
4  }
```


Notes

Heapsort

The `get_right_child_index()` Function

The function presented in the previous slide calculates the index of the heap node's right child. The function differs from the previous one only by its name and the applied expression.

Notes

Heapsort

The `heapify()` Function

```
1  void heapify(int_array_type array, int index, unsigned int size)
2  {
3      int left = get_left_child_index(index),
4          right = get_right_child_index(index),
5          largest = index;
6      if(left<=size)
7          if(array[left]>array[index])
8                  largest = left;
9      if(right<=size)
10         if(array[right]>array[largest])
11             largest = right;
12     if(largest!=index) {
13         swap(&array[index],&array[largest]);
14         heapify(array,largest,size);
15     }
16 }
```


Notes

Heapsort

The `heapify()` Function

The `heapify()` function is the fundamental subroutine in the implementation of the Heapsort algorithm. It restores the heap property. The function doesn't return any values but has three parameters. The first one is for passing the array with a mapped heap, in which the heap property is violated. The second one is for passing an index of a node that possibly violates the property. By the last parameter the length of the heap is passed. In the function's body the indices of the left and right child of the node are calculated and stored in local variables named `left` and `right` (lines no. 3 and 4). To the `largest` local variable is assigned the index of the node that possibly violates the heap property (5th line). This variable is used for storing the index of the node from the aforementioned three (the node that likely violates the property and its two children) that has the greatest value. Initially it is assumed that this is the node specified by the `index` parameter and that the heap property is not violated.

Notes

Heapsort

The `heapify()` Function

Next, the function checks if the left child of the node exists, i.e. if its index stored in the `left` variable is within the length of the heap (6th line). If so, it then verifies if the value of this child is greater than the value of the node (7th line). If also this condition is satisfied then the index of the child is assigned to the `largest` variable. Similarly, in the 8th line the function checks if the right child of the node exists. If so, then the function verifies if its value is greater than the value of the node currently specified by the `largest` variable. In this line it can be the node specified by the `index` parameter or its left child. If the value of the right child is grater than the value of that node then the index of this child is assigned to the `largest` variable (11th line). Thus, after the statement in the 11th line is performed in the `largest` variable is stored the index of the node that has the greatest value of the following three: the node that likely violates the heap property and its two children.

Notes

Heapsort

The heapify() Function

In the 12th line the function checks if the value of `largest` variable is different than the value of the `index` parameter. If not, then the heap property is not violated and the function exits. If so, then the function swaps the values of the nodes specified by the `index` and `largest` variables (13th line). This however can violate the heap property in the bottom part of the heap, i.e. it can be violated in the subtree where the node specified by the `largest` index is a root. Thats why, the `heapify()` function calls itself recursively for that node (14th line).

Notes

Heapsort

The build_heap() Function

```
1  void build_heap(int_array_type array)
2  {
3      int i;
4      const int number_of_elements =
5          sizeof(int_array_type)/sizeof(*array);
6      for(i=number_of_elements/2;i>=0;i--)
7          heapify(array,i,number_of_elements-1);
8  }
```


Notes

Heapsort

The build_heap() Function

The `build_heap()` function creates a heap in an array that is to be sorted. It uses to this end the `heapify()` function. The `build_heap()` function doesn't return any value but takes one argument, which is the array where it creates the heap. A constant that describes the number of elements in the array is defined in the 4th and 5th lines. The heap is created in the `for` loop. Please note, that the loop iterates over the elements of the array starting in the middle and downward. The question arises, why the second (upper) part of the array is not covered by the loop? The function assumes that the length of the heap is equal to the length of the array. It means that the elements that belong to the upper part of the array are the leaves of the heap (or in other words are a single element heaps). The function assures that those elements will also be included in the heap by applying the `heapify()` function for the first part of the array — the latter function will take care of it.

Notes

Heapsort

The heapsort() Function

```
1  void heapsort(int_array_type array)
2  {
3      int last_index = sizeof(int_array_type)/sizeof(*array)-1;
4      int i;
5
6      build_heap(array);
7      for(i=last_index;i>0;i--){
8          swap(&array[0],&array[i]);
9          heapify(array,0,--last_index);
10     }
11 }
```


Notes

Heapsort

The heapsort() Function

The heapsort() function sorts the array. It doesn't return any value, but takes as the argument an array for sorting. In the function body is declared and initialised the last_index variable. Its value is the index of the last element of the array belonging to the heap that also specifies the length of the heap. First, the function creates the heap in the array by invoking the build_heap() function and then, in the for loop iterates over the array starting from the last element (initially it is also the last node of the heap) and finishing in the second element. In each of the iterations it swaps the value of the first element of the array with the element specified by the loop counter (the i variable), and then it restores the heap property starting with the first element of the array. What is the purpose for such steps? The greatest value in the heap is in its root, which is mapped into the first element of the array. In the sorted in ascending order array this value should be stored in the last element. Thus, those elements should exchange their values (8th line).

Notes

Heapsort

The heapsort() Function

This modification can however violate the heap property. That's why the heapsort() function calls the heapify() function for the first element of the array. But this time, the last element of the array is excluded from the heap, because now it has the proper value. In each subsequent iteration of the for loop the length of the heap is decremented by one and the first element of the array (the root of the heap) exchanges its value with the node of the heap specified by the i index. After the loop stops the array is finally sorted.

Notes

Heapsort

The main() Function

```
1  int main(void)
2  {
3      int_array_type array;
4      fill_array(array);
5      print_array(array);
6      heapsort(array);
7      print_array(array);
8      return 0;
9  }
```


Notes

Heapsort

The main() Function

The only difference in the main() functions of the example programs demonstrating the Quicksort and the Heapsort algorithms is that the latter calls the heapsort() function instead of the quicksort() function (6th line). The former function takes as an argument the array to be sorted.

Notes

Summary

Both introduced algorithms for sorting arrays belong to the most efficient in this category, but in the best and average case the Quicksort is slightly better. However, the worst-case time complexity for the latter algorithm is $\Theta(n^2)$ and that happens when the array is already sorted. In this case the algorithm partitions the array into two part, one of them having only one element and the other consisting of the rest of the elements of the original part of the array. The best partitions are those that result in two parts which have equal (with the respect to one element) number of elements. To avoid the worst case, values of several randomly chosen elements can be exchanged. There is no guarantee that those changes don't result in sorting the array, but they are likely to disturb the order of the values if the array is already sorted. Other solution to the wost-case scenario is to chose the pivot value randomly from all the values in the sorted part of the array. However, none of the methods make it certain that the worst case doesn't happen.

Notes

Summary

The Heapsort algorithm has the advantage over the Quicksort algorithm that the recursion can be totally eliminated from its implementation. This can reduce using the call stack. The space complexity of the algorithm can be constant[1].
Statistically the Quicksort algorithm is more frequently used than the Heapsort, but in some applications it is better to use the latter one. An example of such applications are remote services, which need to sort the received data. If the Quicksort algorithm was used in such services then they would be vulnerable to the Denial of Service (DoS) attacks. The attackers would only need to provide specially prepared input data for the services.

[1] Such implementations of the Heapsort algorithm are described in the following books: Jon Bentley, "Programming Pearls" and A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms".

Notes

Questions

?

Notes

The End

Thank You For Your Attention!

Notes