WORKSHEET, WEEK 08: SORTING

Various sorting algorithms are introduced here, because they illustrate algorithm building paradigms (*Anany Levitin. Introduction to the design & analysis of algorithms*). Depending on the context different sorting algorithms may be needed.

8.1 Concepts and Facts

Statement (lower bound for sorting): Any general sorting algorithm receiving an input of n distinct items and producing a sorted output of these items (without assuming anything additional about the input) needs at least $\lceil \log_2 n! \rceil$ comparisons.

Proof: Any decision tree with k levels (i.e. a sorting algorithm making k comparisons, can distinguish at most 2^k different cases. On the other hand, there are $n!$ different ways how n sortable items can be arranged in the input. Thus we must have $2^k \geq n!$ or $k \geq \lceil \log_2 n! \rceil$. (See Comparison Sort.)

Definition: A sorting algorithm is called *stable* iff any two items with equal keys are not swapped: (and thus preserve their initial order). Algorithms that are not guaranteed to preserve such order are *unstable*.

Definition: A sorting algorithm that does not need to know the number of sortable items ahead of the time is called *online*. Those algorithms that receive full sortable array right at the start are called *offline*.

Definition: A sorting algorithm is called in-place, iff it uses just the original array to store the sortable items (plus some local variables). Sorting algorithms that need to allocate new memory for the sortable items are *outplace*.

MergeSort algorithm: This algorithm is a typical illustration of the Divide and Conquer paradigm.

```
MERGESORT($A, p, r$):
1  if $p < r$
2      $q = \lfloor (p+r)/2 \rfloor$
3      MERGESORT($A, p, q$)
4      MERGESORT($A, q+1, r$)
5      MERGE($A, p, q, r$)
```

MergeSort is outplace algorithm – it may waste memory, if run on large arrays. On the other hand, MergeSort is nearly optimal regarding the number of comparisons needed, so it may be helpful, if comparing two items takes much time.

Quicksort algorithm: QuickSort has several flavors; this is one of the easiest, but it sometimes needs one extra swap – see Line 13. This variant of Quicksort always uses the leftmost element of the input area as a pivot – it is easy to understand, but may be inefficient, if the input array is nearly sorted already. More advanced Quicksort flavors are randomized or choose the pivot differently.

```
QUICKSORT($A[\ell \dots r]$):
1.  if $\ell < r$:
2.      $i = \ell$          ($i$ increases from the left, searches elements $\geq$ than pivot)
3.      $j = r + 1$      ($j$ decreases from the right, searches elements $\leq$ than pivot.)
4.      $v = A[\ell]$       ($v$ is the pivot.)
5.      while $i < j$:
6.          $i = i + 1$
7.          while $i < r$ and $A[i] < v$:
8.              $i = i + 1$
9.          $j = j - 1$
10.         while $j > \ell$ and $A[j] > v$:
11.             $j = j - 1$
12.         $A[i] \leftrightarrow A[j]$
13.     $A[i] \leftrightarrow A[j]$    (Undo the extra swap at the end)
14.     $A[j] \leftrightarrow A[\ell]$    (Move pivot to its proper place)
15.     QUICKSORT($A[\ell \dots j - 1]$)
16.     QUICKSORT($A[j + 1 \dots r]$)
```

Quicksort algorithm has good characteristics for nearly all inputs – it sorts in place (does not need much memory), it usually is quite optimal and also easy to implement. It may behave badly for certain special inputs (for example, if the input array is nearly sorted already).

```
BUBBLESORT($A[0 \dots n - 1]$):
1.  do:
2.      $swapped$ = FALSE
3.      for $i$ in RANGE$(1, n)$:   (from $1$ to $n - 1$ inclusive)
4.          if $A[i - 1] > A[i]$:
5.              $A[i - 1] \leftrightarrow A[i]$
6.              $swapped$ = TRUE
7.  while $swapped$:
```

8.2 Problems

Problem 1:

(A) Find the $O(g(n))$ for the following function: $\log_2 n!$.

(B) Some algorithm receives n items as its input and then calls function $f(x_1, x_2, x_3, x_4)$ for any ordered quadruplet x_1, x_2, x_3, x_4 received in the input. Assume that $f(\dots)$ runs in constant time. Find the time complexity of the whole algorithm.

(C) Some algorithm receives n items as its input and then calls a function f on all subsets of the received items having size $\lfloor n/4 \rfloor$. Assume that $f(\dots)$ runs in constant time. Find the time complexity of the whole algorithm.

(D) What is the lower bound of comparisons needed to sort an array of 5 elements (assume they are all different)?

Problem 2: An array of 10 elements is used to initialize a minimum heap (as the first stage of the Heap sort algorithm):

$$\{5, 3, 7, 10, 1, 2, 9, 8, 6, 4\}$$

Assume that the minimum heap is initialized in the most efficient way (inserting elements level by level – starting from the bottom levels). All slots are filled in with the elements of the 10-element array in the order they arrive.

(A) How many levels will the heap tree have? (The root of the heap is considered L_0 – level zero. the last level is denoted by L_{k-1}. Just find the number k for this array.)

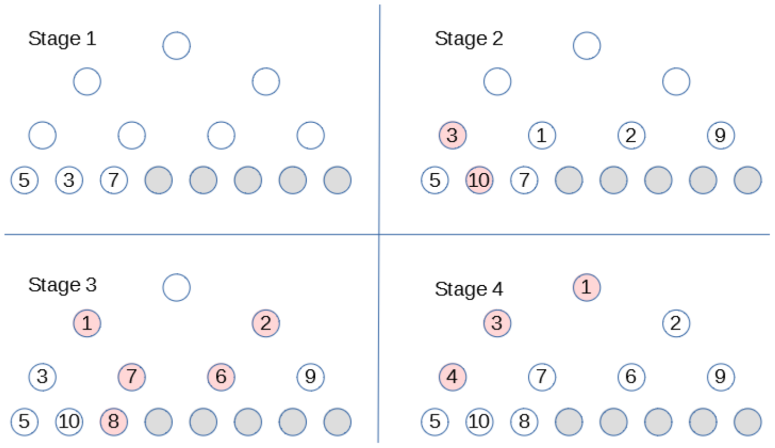
(B) Draw the intermediate states of the heap after each level is filled in. Represent the heap as a binary tree. (If some level L_k is only partially filled and contains less than 2^k nodes, please draw all the nodes as little circles, but leave the unused nodes empty.)

(C) What is the total count of comparisons ($a < b$) that is necessary to build the final minimum heap? (In this part you can assume the worst case time complexity – it is not necessarily achieved for the array given above.)

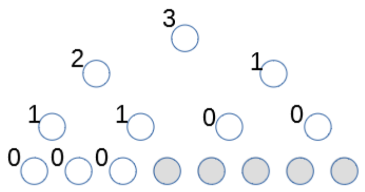
Answer:

(A) 10 elements need a tree with four levels (complete tree with 10 nodes). The last level L_3 will have just three nodes filled in.

(B) See the picture with all four stages of adding elements (unused slots are gray; the nodes that swap their places during the downheap operations are shown in pink).



(C) In a downheap operation (when you add a new node on top of two other nodes), you first need to compare the two siblings, then compare their parent with the smallest of the two siblings (and if it is larger than its child, then swap). So every time some node moves one level down, you need to spend at most two comparisons.



For our complete tree (with five grayed out slots in the last level), the worst case happens, if every node inserted at height h needs to spend $2h$ comparisons to travel to the very bottom (if we assume the worst case – that it is larger than everything that has been inserted so far). So the total number of comparisons is $2 \cdot (1+1+1+2+3) = 16$. In general, this time should grow as $O(n)$, where n is the number of items in the heap being built.

Problem 3:

(A) Run this pseudocode for one invocation $\text{QUICKSORT}(A[0..11])$, where the table to sort is the following:

$$13, 0, 23, 1, 8, 9, 29, 16, 8, 24, 6, 11.$$

Draw the state of the array every time you swap two elements (i.e. execute $A[k_1] \leftrightarrow A[k_2]$ for any k_1, k_2).

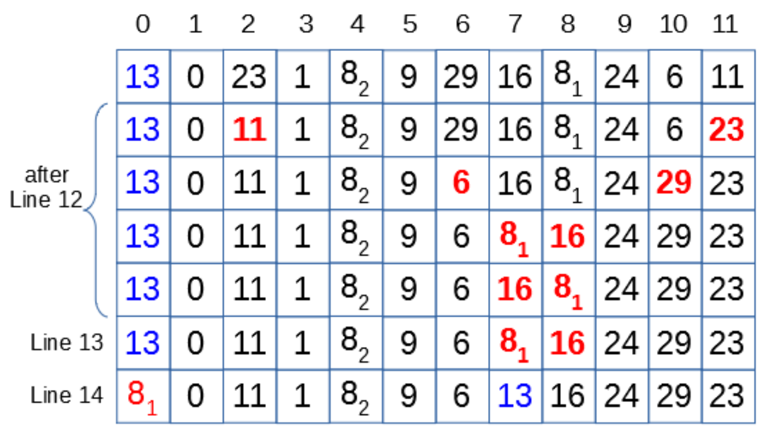
(B) Continue with the first recursive call of QUICKSORT() (the original call QUICKSORT($A[0..11]$) is assumed to be the 0th call of this function). Draw the state of the array every time you swap two elements.

(C) Decide which is the second recursive call of QUICKSORT() and draw the state of the array every time you swap two elements. Show the end-result after this second recursive call at the very end.

Answer:

Your answer can be simple lists of numbers (without any grid lines or additional markings). Just try to keep the lists of numbers aligned.

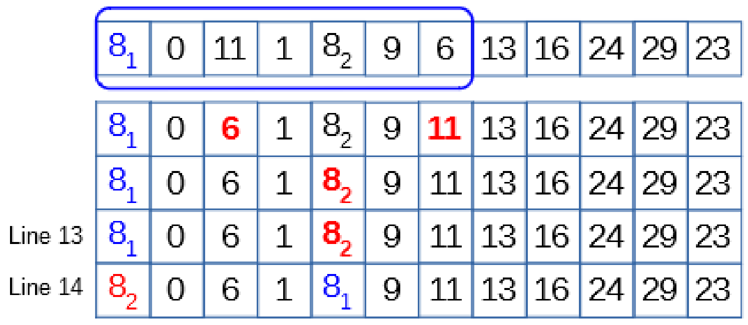
(A) Swaps during the 0th call:



	0	1	2	3	4	5	6	7	8	9	10	11
	13	0	23	1	8_2	9	29	16	8_1	24	6	11
	13	0	11	1	8_2	9	29	16	8_1	24	6	23
after Line 12	13	0	11	1	8_2	9	6	16	8_1	24	29	23
	13	0	11	1	8_2	9	6	8_1	16	24	29	23
	13	0	11	1	8_2	9	6	16	8_1	24	29	23
Line 13	13	0	11	1	8_2	9	6	8_1	16	24	29	23
Line 14	8_1	0	11	1	8_2	9	6	13	16	24	29	23

(B) Since this example contains two elements equal to 8, we added subscripts to them (to show clearly, where every one is being swapped). As integer numbers they are fully identical to the Quicksort algorithm. (Still, the Quicksort algorithm does redundant swaps on them.)

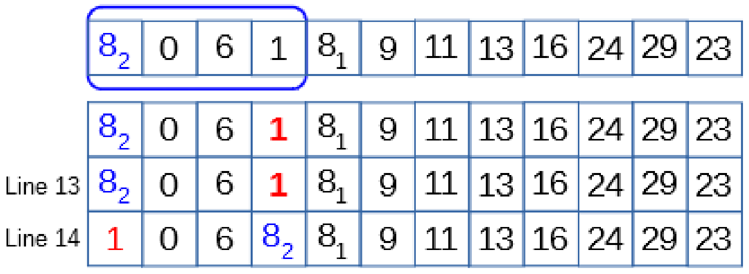
Swaps during the first recursive call.



	8_1	0	11	1	8_2	9	6	13	16	24	29	23
	8_1	0	6	1	8_2	9	11	13	16	24	29	23
	8_1	0	6	1	8_2	9	11	13	16	24	29	23
Line 13	8_1	0	6	1	8_2	9	11	13	16	24	29	23
Line 14	8_2	0	6	1	8_1	9	11	13	16	24	29	23

(C) Notice that the second recursive call happens within the first recursive call (sorting the left side of the left half).

Swaps during the second recursive call:



Problem 4:

Consider the BubbleSort algorithm (see the beginning of the worksheet) for a 0-based array $A[0]\ldots A[n-1]$ of n elements.

(A) How many comparisons (`A[i-1] > A[i]`) in this algorithm are used to sort the given array. Show the state of the array after each `for` loop in the pseudocode is finished.

$$A[0]=9,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ A[9]=8.$$

(B) How many comparisons (`A[i-1] > A[i]`) in this algorithm are used to sort the following array:

$$A[0]=1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ A[9]=0.$$

Answer:

(A) 18 comparisons, 2 executions of the **for** loop:

After the first **for** loop the array is sorted:

$$A[0]=0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ A[9]=9.$$

After the second **for** loop and 9 more comparisons no further swaps occur and the algorithm stops. The array is still the same:

$$A[0]=0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ A[9]=9.$$

(B)

90 comparisons, 10 executions of the **for** loop:

After the first **for** loop:

$$A[0]=1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 0,\ A[9]=9.$$

After the second **for** loop:

$$A[0]=1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 0,\ 8,\ A[9]=9.$$

After the ninth **for** loop:

$$A[0]=0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ \ 8,\ A[9]=9.$$

After the tenth **for** loop the array stays the same and the algorithm stops:

$$A[0]=0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ \ 8,\ A[9]=9.$$

Note: Small values near the end of the list will slow down the Bubble sort considerably. The authors of an accelerated Bubble-sort variant (Comb sort) call such values *turtles*. See https://bit.ly/3mmS6C4.

Problem 5:

We have a 1-based array with 11 elements: $A[1], \ldots, A[11]$. We want to sort it efficiently. Run the MergeSort on this array (see the beginning of the worksheet).

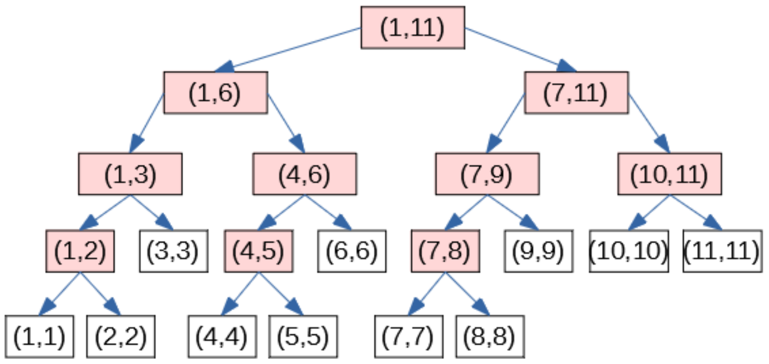
Assume that initially you call this function as MERGESORT(A,1,11), where $p = 1$ and $r = 11$ are the left and the right endpoint of the array being sorted (it includes both ends).

(A) What is the total number of calls to MERGESORT for this array (this includes the initial call as well as the recursive calls on lines 3 and 4 of this pseudocode).

(B) How many comparisons are needed (in the worst case) to sort an array of 11 items by the MergeSort algorithm?

(C) Evaluate $\log_2 11!$ using Stirlings formula or a direct computation. What is the theoretical lower bound on the number of comparisons to sort 11 items?

Answer:



The recursive calls of MERGESORT are shown in the figure – just the parameters p, r for each call. For example, MERGESORT$(A, 1, 11)$ computes $q = \lfloor (1 + 11)/2 \rfloor = 6$, and causes two more calls to MERGESORT$(A, 1, 6)$ and MERGESORT$(A, 7, 11)$ respectively. On the other hand, if $p = r$, then the recursive calls do not happen (one-element list is already sorted). So there are exactly 11 external nodes (leaves) in the recursion tree.

Since the tree of calls is full, it also has 10 internal nodes (shown pink in the picture). The total number of these nodes is $10 + 11 = 21$.