Algorithms and Data Structures **Summer 2008**
K. Mehlhorn and R. Seidel **We. Sept 3rd, morning**
Exercise 5

Motivation

We study an optimized version of quicksort (Section 5.4.2 in Mehlhorn/Sanders). It works in-place, and is fast and space-efficient. Figure 1 shows the pseudocode, and Figure 2 shows a sample execution. The refinements are nontrivial and we need to discuss them carefully.

```
Procedure qSort(a : Array of Element; ℓ, r : ℕ)            // Sort the subarray a[ℓ..r]
  while r − ℓ + 1 > n0 do                                  // Use divide-and-conquer.
    j := pickPivotPos(a, ℓ, r)                             // Pick a pivot element and
    swap(a[ℓ], a[j])                                       // bring it to the first position.
    p := a[ℓ]                                              // p is the pivot now.
    i := ℓ; j := r
    repeat                                                 // a: | ℓ    i→  ←j    r |
      while a[i] < p do i++                                // Skip over elements
      while a[j] > p do j--                                // already in the correct subarray.
      if i ≤ j then                                        // If partitioning is not yet complete,
        swap(a[i], a[j]); i++; j--                         // (*) swap misplaced elements and go on.
    until i > j                                            // Partitioning is complete.
    if i < (ℓ + r)/2 then  qSort(a, ℓ, j); ℓ := i          // Recurse on
    else                   qSort(a, i, r); r := j          // smaller subproblem.
  endwhile
  insertionSort(a[ℓ..r])                                   // faster for small r − ℓ
```

Figure 1: Refined quicksort for arrays

The function *qsort* operates on an array a. The arguments ℓ and r specify the subarray to be sorted. The outermost call is $qsort(a,1,n)$. If the size of the subproblem is smaller than some constant n_0, we resort to a simple algorithm[1] such as insertion sort. The best choice for n_0 depends on many details of the machine and compiler and needs to be determined experimentally; a value somewhere between 10 and 40 should work fine under a variety of conditions.

[1] Some authors propose leaving small pieces unsorted and cleaning up at the end using a single insertion sort. Although this trick reduces the number of instructions executed, the solution shown is faster on modern machines because the subarray to be sorted will already be in cache.

i	$\rightarrow$							$\leftarrow$	j
3	6	8	1	0	7	2	4	5	9
2	6	8	1	0	7	**3**	4	5	9
2	**0**	8	1	**6**	7	3	4	5	9
2	0	**1**	**8**	**6**	7	3	4	5	9
		j	i						

```
3 6 8 1 0 7 2 4 5 9
2 0 1|8 6 7 3 4 5 9
1 0|2|5 6 7 3 4|8 9
0 1| |4 3|7 6 5|8 9
   | |3 4|5 6|7|
   | |   |5 6| |
```

Figure 2: Execution of *qSort* (Fig. 1) on $(3,6,8,1,0,7,2,4,5,9)$ using the first element as the pivot and $n_0 = 1$. The *left-hand side* illustrates the first partitioning step, showing elements in **bold** that have just been swapped. The *right-hand side* shows the result of the recursive partitioning operations

The pivot element is chosen by a function *pickPivotPos* that we shall not specify further. The correctness does not depend on the choice of the pivot, but the efficiency does. Possible choices are the first element; a random element; the median ("middle") element of the first, middle, and last elements; and the median of a random sample consisting of k elements, where k is either a small constant, say three, or a number depending on the problem size, say $\lceil \sqrt{r-\ell+1}\, \rceil$. The first choice requires the least amount of work, but gives little control over the size of the subproblems; the last choice requires a nontrivial but still sublinear amount of work, but yields balanced subproblems with high probability. After selecting the pivot p, we swap it into the first position of the subarray (= position ℓ of the full array).

The repeat–until loop partitions the subarray into two proper (smaller) subarrays. It maintains two indices i and j. Initially, i is at the left end of the subarray and j is at the right end; i scans to the right, and j scans to the left. After termination of the loop, we have $i = j+1$ or $i = j+2$, all elements in the subarray $a[\ell..j]$ are no larger than p, all elements in the subarray $a[i..r]$ are no smaller than p, each subarray is a proper subarray, and, if $i = j+2$, $a[i+1]$ is equal to p. So, recursive calls *qSort*(a,ℓ,j) and *qsort*(a,i,r) will complete the sort. We make these recursive calls in a nonstandard fashion; this is discussed below.

1. Is it OK to change the scan loops into

   while $a[i] \le p$ **do** i++
   while $a[j] \ge p$ **do** j- -

   Be aware that array elements are allowed to be equal.

2. Argue correctness of the partitioning step.

3. Is the change in the scan loops OK if array elements are known to be distinct.

The refined quicksort handles recursion in a seemingly strange way. Recall that we need to make the recursive calls *qSort*(a,ℓ,j) and *qSort*(a,i,r). We may make these calls in either order. We exploit this flexibility by making the call for the smaller subproblem first. The call for the larger subproblem would then be the last thing done in *qSort*. This situation is known as *tail* *recursion* in the programming-language literature. Tail recursion can be eliminated by setting the parameters (ℓ and r) to the right values and jumping to the first line of the procedure. This is precisely what the while loop does. Why is this manipulation useful? Because it guarantees that the recursion stack stays logarithmically bounded; the precise bound is $\lceil \log(n/n_0) \rceil$. This follows from the fact that we make a single recursive call for a subproblem which is at most half the size.

1. What is the maximal depth of the recursion stack without the “smaller subproblem first” strategy? Give a worst-case example.

Have fun with the solution!