Heapsort and Quicksort

We will see two more sorting algorithms in this lecture. The first, heapSort, is very interesting theoretically. It sorts an array with n items in time $\Theta(n \lg n)$ *in place* (i.e., it doesn't need much extra memory). The second algorithm, quickSort, is known for being the most efficient sorting algorithm in practice, although it has a rather high worst-case running time of $\Theta(n^2)$.

8.1 Heapsort

To understand the basic principle behind heapSort, it is best to recall maxSort (Algorithm 8.1), a simple but slow sorting algorithm (given as Exercise 4 of Lecture Note 2). The algorithm maxSort repeatedly picks the maximum key in the subarray it currently considers and puts it in the last place of this subarray, where it belongs. Then it continues with the subarray containing all but the last item.

Algorithm maxSort(A)

1. **for** $j \leftarrow A.length - 1$ **downto** 1 **do**
2. $\quad m \leftarrow 0$
3. $\quad$ **for** $i = 1$ **to** j **do**
4. $\quad\quad$ **if** $A[i].key > A[m].key$ **then** $m \leftarrow i$
5. $\quad$ exchange $A[m], A[j]$

Algorithm 8.1

The algorithm heapSort follows the same principle, but uses a heap to find efficiently the maximum in each step. We will define heapSort by building on the methods of the Lecture Note on Priority Queues and Heaps. However, for heapSort, we only need the following two methods:

- removeMax(): Return and remove an item with maximum key.
- buildHeap(A) (and by implication, heapify()): Turn array A into a heap.

Notice that because we provide all the items at the beginning of the algorithm (our use of the heap is *static* rather than *dynamic*), we do not need the method insertItem() as an individual method. With the implementations explained in the Lecture Note on Priority Queues and Heaps, removeMax() has a running time of $\Theta(\lg n)$ and buildHeap has a running time of $\Theta(n)$.

With these methods available, the implementation of heapSort is very simple. Consider Algorithm 8.2. To see that it works correctly, observe that an array A of size n is in sorted (increasing) order *if and only if* for every j with $1 \leq j \leq A.length - 1 = n - 1$, the entry $A[j]$ contains a maximum element of the subarray $A[0 \ldots j]$.

Line 3 of heapSort() ensures that at each index from $A.length - 1$ down to 0, we always insert a maximum element of the remaining collection of elements into $A[j]$ (and remove it from the heap, reducing the collection of elements there). Hence it sorts correctly.

Algorithm heapSort(A)

1. buildHeap(A)
2. **for** $j \leftarrow A.length - 1$ **downto** 0 **do**
3. $\quad A[j] \leftarrow$ removeMax()

Algorithm 8.2

The analysis is easy: Line 1 requires time $\Theta(n)$, as shown at the end of the Lecture Note on Priority Queues and Heaps. An iteration of the loop in lines 2–3 for a particular index j requires time $\Theta(\lg j)$ (because we work from the top of the array down, therefore when we consider index j, the heap contains only $j+1$ elements). Thus we get

$$T_{\text{heapSort}}(n) = \Theta(n) + \sum_{j=2}^{n} \Theta(\lg(j)) = \Theta(n) + \Theta\Big(\sum_{j=2}^{n} \lg(j)\Big).$$

Since for $j \leq n$ we have $\lg(j) \leq \lg(n)$,

$$\sum_{j=2}^{n} \lg(j) \leq (n-1)\lg(n) = O(n \lg n).$$

Since for $j \geq n/2$ we have $\lg(j) \geq \lg(n) - 1$,

$$\sum_{j=2}^{n} \lg(j) \geq \sum_{j=\lceil n/2 \rceil}^{n} (\lg(n) - 1) = \lfloor n/2 \rfloor (\lg(n) - 1) = \Omega(n \lg(n)).$$

Therefore $\sum_{j=2}^{n} \lg(j) = \Theta(n \lg n)$ and

$$T_{\text{heapSort}}(n) = \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n).$$

In the case where all the keys are *distinct*, we are guaranteed that for a constant fraction of the removeMax calls, the call takes $\Omega(\lg \widehat{n})$ time, where $\widehat{n}$ is the number of items at the time of the call. This is not obvious, but it is true—it is because we copy the "last" cell over onto the root before calling heapify. So in the case of distinct keys, the "best case" for heapSort is also $\Omega(n \lg n)$ [1].

By recalling the algorithms for buildHeap and (the array-based version of) removeMax it is easy to verify that heapSort sorts in place. After some thought it can be seen that heapSort is *not* stable.

[1] This is difficult to prove. The result is due to B. Bollobas, T.I. Fenner and A.M. Frieze, "On the Best Case of Heapsort", *Journal of Algorithms*, **20**(2), 1996.

8.2 Quicksort

The sorting algorithm quickSort, like mergeSort, is based on the *divide-and-conquer* paradigm. As opposed to mergeSort and heapSort, quickSort has a relatively bad worst case running time of $\Theta(n^2)$. However, quickSort is very fast in practice, hence the name. Theoretical evidence for this behaviour can be provided by an *average case analysis*. The average-case analysis of quickSort is too technical for *Informatics 2B*, so we will only consider worst-case and best-case here. If you take the 3rd-year *Algorithms and Data Structures (ADS)* course, you will see the average-case analysis there.

The algorithm quickSort works as follows:

(1) If the input array has less than two elements, there is nothing to do.

Otherwise, *partition* the array as follows: Pick a particular key called the *pivot* and divide the array into two subarrays, of which the first only contains items whose key is smaller than or equal to the pivot and the second only items whose key is greater than or equal to the pivot.

(2) Sort the two subarrays recursively.

Note that quickSort does most work in the "divide" step (i.e., in the partitioning routine), whereas in mergeSort the dividing is trivial, but the "conquer" step must reassemble the recursively sorted subarrays using the merge method. This is not necessary in quickSort, because after the first step all elements in the first subarray are smaller than those in the second subarray. A problem, which is responsible for the bad worst-case running time of quickSort, is that the partitioning step is not guaranteed to divide the array into two subarrays of the same size (if we could enforce this somehow, we would have a $\Theta(n \lg n)$ algorithm). If we implement partitioning in an obvious way, all items can end up in one of the two subarrays, and we only reduce the size of our problem by 1.

Algorithm 8.3 is a pseudo-code implementation of the main algorithm.

Algorithm quickSort(A, i, j)

1. **if** $i < j$ **then**
2. $\quad$ $split \leftarrow$ partition(A, i, j)
3. $\quad$ quickSort$(A, i, split)$
4. $\quad$ quickSort$(A, split + 1, j)$

Algorithm 8.3

All the work is done in the partitioning routine. First, the routine must pick a pivot. The simplest choice is to pick the key of the first element. We want to avoid setting up a new, temporary array, because that would be a waste of memory space and also time (e.g., for initialising the new array and copying elements back to the old one in the end). Algorithm 8.4 is an in-place partitioning routine. Figure 8.5 illustrates how partition works.

Algorithm partition(A, i, j)

1. $pivot \leftarrow A[i].key$
2. $p \leftarrow i - 1$
3. $q \leftarrow j + 1$
4. **while** TRUE **do**
5. $\quad$ **do** $q \leftarrow q - 1$ **while** $A[q].key > pivot$
6. $\quad$ **do** $p \leftarrow p + 1$ **while** $A[p].key < pivot$
7. $\quad$ **if** $p < q$ **then**
8. $\quad\quad$ exchange $A[p]$, $A[q]$
9. $\quad$ **else return** q

Algorithm 8.4

To see that partition works correctly, we observe that after each iteration of the main loop in lines 4–8, for all indices r such that $q < r \leq j$ we have $A[r].key \geq pivot$, and for all r such that $i \leq r < p$ we have $A[r].key \leq pivot$. Actually, after all iterations except maybe the last we also have $A[p].key \leq pivot$. Formally, we can establish this by an induction on the number of iterations of the loop. In addition, we must verify that our return value q is greater than or equal to i and smaller than $j - 1$. It is greater than or equal to i because at any time during the computation we have $A[i].key \leq pivot$. The q returned is smaller than j, because after the first iteration of the main loop we have $q \leq j$ and $p = 1$. If q is already smaller than j, it remains smaller. If not, we still have $p < q$. So there will be a second iteration, after which $q \leq j - 1 < j$.

We can now use the properties derived to show that Algorithm 8.3 is correct. Firstly recall that we set *split* to be the value of q returned. The fact that $i \leq q \leq j - 1 < j$ means that $A[i..q]$ and $A[q+1..j]$ are non-empty. Thus each call is on a portion of the array that is smaller than the input one and so the algorithm halts; otherwise there is the danger of an infinite recursion which would result if one portion was always empty. We must also show that the keys in the two portions are appropriate, i.e., at the time of call $A[r] \leq pivot$ for $i \leq r \leq q$ and $A[r] \geq pivot$ for $q + 1 \leq r \leq j$. The second inequality follows immediately from above (since we have $A[r].key \geq pivot$ for $q < r \leq j$). To derive the first one observe first that when partition(A, i, j) halts we have either $p = q$ or $p = q + 1$. From the preceding paragraph we know that $A[r].key \leq pivot$ for $i \leq r < p$. So if $p = q + 1$ we are done. Otherwise $p = q$ and we just need to verify that $A[p] \leq pivot$. For this we note that the loop on line 5 terminates when the test $A[q].key > pivot$ fails (for the current value of q). So the last time it was executed we had $A[q].key \not> pivot$, i.e., $A[q].key \leq pivot$. Since this happens for the final value of q and $p = q$ we are done.

Let us analyse the running time. Let $n = j - i + 1$. We observe that during the execution of partition, we always have $q \geq p - 1$, because all keys below p are always smaller than or equal to $pivot$ and all keys above q are always larger than or equal to $pivot$. This implies that the running time (best-case, average-case and

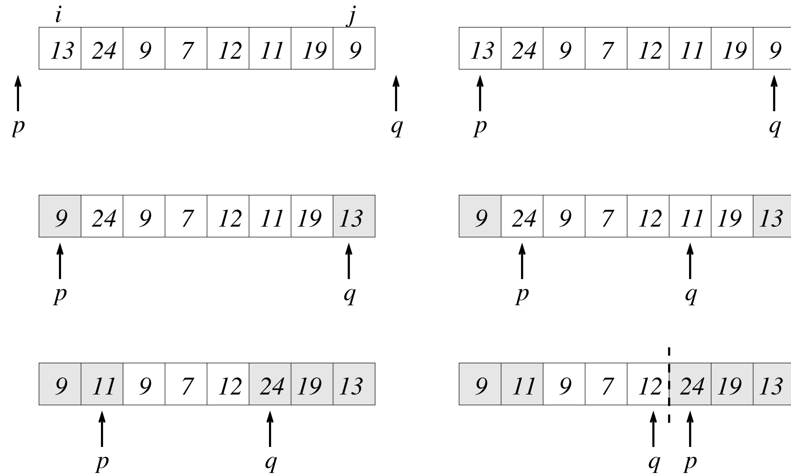

Figure 8.5. The operation of partition

worst-case) of partition is $\Theta(n)$. The reason for the runtime, briefly, is that we must move p and q so they are either equal or one passes the other. Since they start $\Theta(n)$ entries apart and each basic step either increases p by 1 or decreases q by 1 (possibly accompanied by the swapping two entries) we must make this many moves and each move costs constant time.

Unfortunately, this does not give us a simple recurrence for the running time of quickSort, because we do not know where the array will be split. All we get is the following:

$$T_{\text{quickSort}}(n) = \max_{1 \le s \le n-1}\big(T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n-s)\big) + \Theta(n).$$

It can be shown that this implies that the worst-case running-time $T_{\text{quickSort}}(n)$ of quickSort satisfies $T_{\text{quickSort}}(n) = \Theta(n^2)$.

We discuss the intuition behind this. The worst-case (intuitively speaking) for quickSort is that the array is always partitioned into subarrays of sizes 1 and $n-1$, because in this case we get a recurrence

$$T(n) = T(n-1) + T(1) + \Theta(n) = T(n-1) + \Theta(n),$$

which implies $T(n) = \Theta(n^2)$. Of course we need to also justify our assumption that this combination of partitions could happen—in fact, one example of a case which causes this bad behaviour is the case when the input array A is initially sorted, which is not uncommon for some applications.

The best case arises when the array is always split in the middle. Then we get the same recurrence as for mergeSort,

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n),$$

which implies $T(n) = \Theta(n \lg(n))$. Fortunately, the typical or "average" case is much closer to the best case than to the worst case. A mathematically complicated analysis shows that for random arrays with each permutation of the keys being equally likely, the average running time of quickSort is $\Theta(n \lg n)$. Essentially good splits are much more likely than bad ones so on average the runtime is dominated by them.

Nevertheless, the poor running time on sorted arrays (and similarly on nearly sorted arrays) is a problem. It can be avoided, though, by choosing the pivot differently. For example, if we take the key of the middle item $A[\lfloor (i+j)/2 \rfloor]$ as the pivot, the running time on sorted arrays becomes $\Theta(n \lg n)$, because they will always be partitioned evenly. But there are other worst case arrays with a $\Theta(n^2)$ running time for this choice of the pivot. A strategy that avoids this worst-case behaviour altogether, with high probability, is to choose the pivot randomly (this is a topic for a more advanced course). This strategy is an example of a *randomised algorithm*, an approach that has gained increasing importance. Note that the runtime of this algorithm can vary between different runs on the same input. This approach for quicksort guarantees that if we run the algorithm over a large number of times on inputs of size n then the runtime will be $O(n \lg n)$ no matter what inputs are supplied. The proof of this claim is beyond the scope of this course, it can be found in [CLRS].

We conclude with an observation that quickSort is an in-place sorting algorithm. However, it is not too difficult to come up with examples to show it is not stable.

8.3 Further Reading

If you have [GT], look for heapSort in the "Priority Queues" chapter, and quicksort in the "Sorting, Sets and Selection" chapter. [CLRS] has an entire chapter titled "Heapsort" and another chapter on "Quicksort" (ignore the advanced material).

Exercises

1. Suppose that the array
$$A = \langle 5, 0, 3, 11, 9, 8, 4, 6 \rangle.$$
is sorted by heapSort. Show the status of the heap after buildHeap(A) and after each iteration of the loop. (Represent the heap as a tree.)

2. Consider the enhanced version printQuickSort of quickSort displayed as Algorithm 8.6. Line 1 simply prints the keys of $A[i], \ldots, A[j]$ on a separate line of the standard output. Let $A = \langle 5, 0, 3, 11, 9, 8, 4, 6 \rangle$.

   What does printQuickSort$(A, 0, 7)$ print?

3. Give an example to show that quickSort is not stable.

Algorithm printQuickSort(A, i, j)

1. **print** $A[i \ldots j]$
2. **if** $i < j$ **then**
3. $\quad split \leftarrow$ partition(A, i, j)
4. $\quad$ quickSort$(A, i, split)$
5. $\quad$ quickSort$(A, split + 1, j)$

Algorithm 8.6