Sorting, Merge Sort and the Divide-and-Conquer Technique

This and a subsequent next lecture will mainly be concerned with *sorting algorithms*. Sorting is an extremely important algorithmic problem that often appears as part of a more complicated task. Quite frequently, huge amounts of data have to be sorted. It is therefore worthwhile to put some effort into designing efficient sorting algorithms.

In this lecture, we will mainly consider the mergeSort algorithm. mergeSort is based on the principle of *divide-and-conquer*. Therefore in this note we will also discuss divide-and-conquer algorithms and their analysis in general.

7.1 The Sorting Problem

The problem we are considering here is that of sorting a collection of *items* by their *keys*, which are assumed to be comparable. We assume that the items are stored in an array that we will always denote by A. The number of items to be sorted is always denoted by n. Of course it is also conceivable that the items are stored in a linked list or some other data structure, but we focus on arrays. Most of our algorithms can be adapted for sorting linked lists, although this may go along with a loss in efficiency. By assuming we work with arrays, we assume we may keep all the items in memory at the same time. This is not the case for large-scale sorting, as we will see in a later lecture.

We already saw the insertionSort sorting algorithm in Lecture 2 of this thread. It is displayed again here as Algorithm 1. Recall that the asymptotic worst-case

Algorithm insertionSort(A)

1. **for** $j \leftarrow 1$ **to** $A.length - 1$ **do**
2. $\quad a \leftarrow A[j]$
3. $\quad i \leftarrow j - 1$
4. $\quad$ **while** $i \geq 0$ and $A[i].key > a.key$ **do**
5. $\quad\quad A[i+1] \leftarrow A[i]$
6. $\quad\quad i \leftarrow i - 1$
7. $\quad A[i+1] \leftarrow a$

Algorithm 1

running time of insertionSort is $\Theta(n^2)$. For a sorting algorithm, this is quite poor. We will see a couple of algorithms that do much better.

7.2 Important characteristics of Sorting Algorithms

The main characteristic we will be interested in for the various sorting algorithms we study is worst-case running-time (we will also refer to best-case and average-case from time to time). However, we will also be interested in whether our sorting algorithm is *in-place* and whether it is *stable*.

Definition 2 *An sorting algorithm is said to be an* in-place *algorithm if it can be (simply) implemented on the input array, with only $O(1)$ extra space (extra variables).*

Clearly it is useful to have an in-place algorithm if we want to minimize the amount of space used by our algorithm. The issue of minimizing space only really becomes important when we have a large amount of data, for applications such as web-indexing.

Definition 3 *A sorting algorithm is said to be* stable *if for every i, j such that $i < j$ and $A[i].key = A[j].key$, we are guaranteed that $A[i]$ comes before $A[j]$ in the output array.*

Stability is a "desirable" property for a sorting algorithm. We won't really see why it is useful in Inf2B (if you take the 3rd-year *Algorithms and Data Structures* course, you'll see that it can lead to a powerful reduction in running-time for some cases of sorting[1]).

If we refer back to insertionSort, we can see that it is clearly an *in-place* sorting algorithm, as it operates directly on the input array. Also, notice that the test $A[i].key > a.key$ in line 4. is a strict inequality and so that when $A[i]$ is inserted into the (already sorted) array $A[1 \ldots i-1]$, it will be inserted *after* any element with the same key within $A[1 \ldots i-1]$. Hence insertionSort is stable.

7.3 Merge Sort

The idea of merge sort is very simple: split the array A to be sorted into halves, sort both halves recursively, and then *merge* the two sorted subarrays together to one sorted array. Algorithm 4 implements this idea in a straightforward manner. To make the recursive implementation possible, we introduce two additional parameters i and j marking the boundaries of the subarray to be sorted; mergeSort(A, i, j) sorts the subarray $A[i \ldots j] = \langle A[i], A[i+1], \ldots, A[j] \rangle$. To sort the whole array, we obviously have to call mergeSort$(A, 0, n-1)$.

The key to MergeSort is the merge() method, called as merge(A, i, mid, j) in Algorithm 4. Assuming that $i \leq mid < j$ and that the subarrays $A[i \ldots mid]$ and $A[mid+1 \ldots j]$ are both sorted, merge(A, i, mid, j) sorts the whole subarray $A[i \ldots j]$. The merging is achieved by first defining a new array B to hold the sorted data. Then we initialise an index k for the $A[i \ldots mid]$ subarray to i, and index ℓ for the $A[mid+1 \ldots j]$ subarray to $mid+1$. We walk these indices up the array, at each step storing the minimum of $A[k]$ and $A[\ell]$ in B, and then incrementing that index and the "current index" of B.

[1] This is the Radix-sort algorithm, if you want to look this up.

Algorithm mergeSort(A, i, j)

1. **if** $i < j$ **then**
2. $\quad mid \leftarrow \lfloor \frac{i+j}{2} \rfloor$
3. $\quad$ mergeSort(A, i, mid)
4. $\quad$ mergeSort$(A, mid+1, j)$
5. $\quad$ merge(A, i, mid, j)

Algorithm 4

Algorithm 5 shows an implementation of merge(). Note that at the end of the initial merge (loop on line 3) there might be some entries left in the lower half of A or the upper half but not both; so at most one of the loops on lines 11 and 15 would be executed (these just append any left over entries to the end of B). Figure 6 shows the first few steps of merge() on an example array.

Let us analyse the running time of merge. Let $n = j-i+1$, i.e., n is the number of entries in the subarray $A[i \ldots j]$. There are no nested loops, and clearly each of the loops of merge is iterated at most n times. Thus the running time of merge is $O(n)$. Since the last loop is certainly executed n times, the running time is also $\Omega(n)$ and thus $\Theta(n)$. Actually, the running time of merge is $\Theta(n)$ no matter what the input array A is (i.e., even in the best case).

Now let us look at mergeSort. Again we let $n = j - i + 1$. We get the following expression for the running time $T_{\text{mergeSort}}(n)$:

$$\begin{aligned} T_{\text{mergeSort}}(1) &= \Theta(1), \\ T_{\text{mergeSort}}(n) &= \Theta(1) + T_{\text{mergeSort}}(\lceil n/2 \rceil) + T_{\text{mergeSort}}(\lfloor n/2 \rfloor) + T_{\text{merge}}(n). \end{aligned}$$

Since we already know that the running time of merge is $\Theta(n)$, we can simplify this to

$$T_{\text{mergeSort}}(n) = T_{\text{mergeSort}}(\lceil n/2 \rceil) + T_{\text{mergeSort}}(\lfloor n/2 \rfloor) + \Theta(n).$$

We will see in §7.5 that solving this recurrence yields

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

Our analysis will actually show that the running time of mergeSort is $\Theta(n \lg(n))$ no matter what the input is, i.e., even in the best case.

The runtime of $\Theta(n \lg(n))$ is not surprising. The recurrence shows that for an input of size n the cost is that of the two recursive calls plus a cost proportional to n. Now each recursive call is to inputs of size essentially $n/2$. Each of these makes two recursive calls for size $n/4$ and we also charge a cost proportional to $n/2$. So the cost of the next level is a one off cost proportional to n (taking account of the two $n/2$ one off costs) and 4 calls to size $n/4$. So we see that at each recursive call we pay a one off cost that is proportional to n and then make 2^r calls to inputs of size $n/2^r$. The base case is reached when $n/2^r = 1$, i.e., when $r = \lg(n)$. So we have $\lg(n)$ levels each of which has a cost proportional to n.

Algorithm merge(A, i, mid, j)

1. initialise new array B of length $j - i + 1$
2. $k \leftarrow i$; $\ell \leftarrow mid + 1$; $m \leftarrow 0$
3. **while** $k \leq mid$ and $\ell \leq j$ **do** (merge halves of A into B)
4. $\quad$ **if** $A[k].key \leq A[\ell].key$ **then**
5. $\quad\quad B[m] \leftarrow A[k]$
6. $\quad\quad k \leftarrow k + 1$
7. $\quad$ **else**
8. $\quad\quad B[m] \leftarrow A[\ell]$
9. $\quad\quad \ell \leftarrow \ell + 1$
10. $\quad m \leftarrow m + 1$
11. **while** $k \leq mid$ **do** (copy anything left in lower half of A to B)
12. $\quad B[m] \leftarrow A[k]$
13. $\quad k \leftarrow k + 1$
14. $\quad m \leftarrow m + 1$
15. **while** $\ell \leq j$ **do** (copy anything left in upper half of A to B)
16. $\quad B[m] \leftarrow A[\ell]$
17. $\quad \ell \leftarrow \ell + 1$
18. $\quad m \leftarrow m + 1$
19. **for** $m = 0$ **to** $j - i$ **do** (copy B back to A)
20. $\quad A[m + i] \leftarrow B[m]$

Algorithm 5

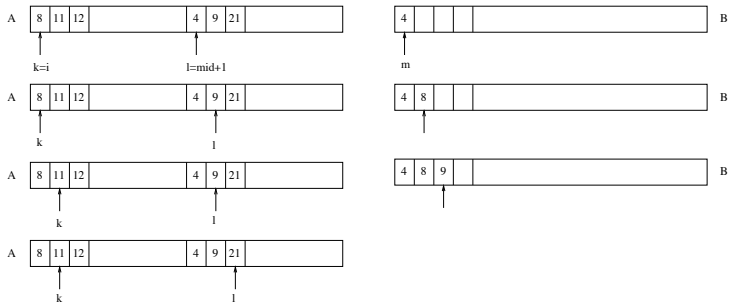

Figure 6. The first few steps of merge() on two component subarrays.

Thus the total cost is proportional to $n \lg(n)$. We will prove the correctness of this sketch argument below.

Since $n \lg(n)$ grows much slower than n^2, mergeSort is much more efficient than insertionSort. As simple experiments show, this can also be observed in practice. So usually mergeSort is preferable to insertionSort. However, there are situations where this is not so clear. Suppose we want to sort an array that is almost sorted, with only a few items that are slightly out of place. For such "almost sorted" arrays, insertionSort works in "almost linear time", because very few items have to be moved. mergeSort, on the other hand, uses time $\Theta(n \lg(n))$ *no matter what the input array looks like*. insertionSort also tends to be faster on very small input arrays, because there is a certain overhead caused by the recursive calls in mergeSort.

Maybe most importantly, mergeSort wastes a lot of space, because merge copies the whole array to an intermediate array. Clearly mergeSort is *not* an in-place algorithm in the sense of Definition 2. As to the question of stability for mergeSort, line 4 of merge achieves stability for mergeSort *only because* it has a *non-strict* inequality for testing when we should put the item $A[k]$ (from the left subarray) down before the item $A[\ell]$ (from the right subarray). In some places you may see non-stable versions of mergeSort, where a strict inequality is used in line 4.

7.4 Divide-and-Conquer Algorithms

The *divide-and-conquer* technique involves solving a problem in the following way:

(1) Divide the input instance into several instances of the same problem of smaller size.

(2) Recursively solve the problem on these smaller instances.

(3) Combine the solutions for the smaller instances to a solution for the original instance (so after the recursive calls, we do some "extra work" to find the solution for our original instance).

Obviously, mergeSort is an example of a divide-and-conquer algorithm.

Suppose now that we want to analyse the running time of a divide-and-conquer algorithm. Denote the size of the input by n. Furthermore, suppose that the input instance is split into a instances of sizes $n_1, \ldots, n_a$, for some constant $a \geq 1$. Then we get the following recurrence for a running time $T(n)$:

$$T(n) = T(n_1) + \ldots + T(n_a) + f(n),$$

where $f(n)$ is the time required by steps (1) (for setting up the recursions) and (3) (for the "extra work"). Typically, $n_1, \ldots, n_k$ are all of the form $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ for some $b > 1$. Disregarding floors and ceilings for a moment, we get a recurrence of the form:

$$T(n) = a\,T(n/b) + f(n).$$

If we want to take floors and ceilings into account, we have to replace this by $T(n) = a_1 T(\lfloor n/b \rfloor) + a_2 \cdot T(\lceil n/b \rceil) + f(n)$. It is usually the case that the asymptotic growth rate of the solution to the recurrence does not depend on floors and ceilings. Throughout this lecture note we will usually disregard the floors and ceilings (this can be justified formally).

To specify the recurrence fully, we need also to say what happens for small instances that can no longer be divided, and we must specify the function f. We can always assume that for input instances of size below some constant n_0 (typically, $n_0 = 2$), the algorithm requires $\Theta(1)$ steps. A typical function f occurring in the analysis of divide-and-conquer algorithms has an asymptotic growth rate of $\Theta(n^k)$, for some constant k.

To conclude: The analysis of divide-and-conquer algorithms usually yields recurrences of the form

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0, \\ a\,T(n/b) + \Theta(n^k) & \text{if } n \geq n_0, \end{cases} \tag{7.1}$$

where $n_0, a \in \mathbb{N}$, $k \in \mathbb{N}_0$, and $b \in \mathbb{R}$ with $b > 1$ are constants. (We normally assume that n is a power of b so that n/b is an integer as the recurrence "unwinds." Without this assumption we must use floor or ceiling functions as appropriate. The fact is that the asymptotic growth rate for the type of recurrence shown; see CLRS for a discussion.)

7.5 Solving Recurrences

Example 7. Recall that for the running time of merge sort we got the following recurrence:

$$T_{\mathsf{mergeSort}}(n) = \begin{cases} \Theta(1) & \text{if } n < 2, \\ T_{\mathsf{mergeSort}}(\lceil n/2 \rceil) + T_{\mathsf{mergeSort}}(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

We first assume that n is a power of 2, say, $n = 2^\ell$; in this case we can omit floors and ceilings. Then

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 2\big(2T(n/4) + \Theta(n/2)\big) + \Theta(n) \\ &= 4T(n/4) + 2\Theta(n/2) + \Theta(n) \\ &= 4\big(2T(n/8) + \Theta(n/4)\big) + 2\Theta(n/2) + \Theta(n) \\ &= 8T(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n) \\ &\vdots \\ &= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i \Theta(n/2^i) \\ &\vdots \\ &= 2^\ell T(n/2^\ell) + \sum_{i=0}^{\ell-1} 2^i \Theta(n/2^i) \end{aligned}$$

$$\begin{aligned} &= nT(1) + \sum_{i=0}^{\ell-1} 2^i \Theta(n/2^i) \\ &= n\Theta(1) + \sum_{i=0}^{\lg(n)-1} 2^i \Theta(n/2^i) \\ &= \Theta(n) + \Theta(n\lg(n)) \\ &= \Theta(n\lg(n)). \end{aligned}$$

Strictly speaking the argument just given, usually referred to as unwinding the recurrence, is incomplete (those dots in the middle). We have taken it on trust that the last lines are the correct outcome. We can easily fill in this "gap" using induction (e.g. by proving that the claimed solution is indeed correct). However as long as the unwinding process is carried out carefully the induction step is a matter of routine and is usually omitted. You should however be able to supply it if asked.

Although we have only treated the special case when n is a power of 2, we can now use a trick to prove that we have $T(n) = \Theta(n\lg n)$ for all n. We make the reasonable assumption that $T(n) \leq T(n+1)$ (this can be proved from the recurrence for $T(n)$ by induction). Now notice that for every n, there is some exponent k such that $n \leq 2^k < 2n$. This is the fact we need. We observe that $T(n) \leq T(2^k) = O(2^k \lg(2^k)) = O(2n\lg(2n)) = O(n\lg n)$. Likewise for every $n \geq 1$ there is some k' such that $n/2 < 2^{k'} \leq n$. Hence $T(n) = \Omega((2^{k'})\lg(2^{k'})) = \Omega((n/2)\lg(n/2)) = \Omega(n\lg n)$. Putting these two together we have $T(n) = \Theta(n\lg n)$.

An alternative approach is to use the full recurrence directly though this is rather tedious. Note though that once we have a good guess at what the solution is we can prove its correctness by using induction and the full recurrence, we do need to unwind the complicated version.

Before moving on it is worth explaining one possible error that is sometimes made when unwinding recurrences. We have

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 2\big(2T(n/4) + \Theta(n/2)\big) + \Theta(n) \\ &= 4T(n/4) + 2\Theta(n/2) + \Theta(n) \end{aligned}$$

It is very tempting to simplify the last line to $4T(n/4) + \Theta(n)$ on the basis that $2\Theta(n/2) + \Theta(n) = \Theta(n)$ and do likewise at each stage. It is certainly true that $2\Theta(n/2) + \Theta(n) = \Theta(n)$. However such replacements are only justified if we have a *constant* number of $\Theta(\cdot)$ terms. In our case we unwind the recurrence to obtain $\lg(n)$ terms and this is not a constant. You are strongly encouraged to unwind the recurrence with the $\Theta(n)$ replaced by cn where $c > 0$ is some constant. Observe that at each stage we obtain a multiple of cn as an additive term (cn, $2cn$, $3cn$ etc.). Of course after any fixed number of stages we just have a constant multiple of n. But after $\log(n)$ stages this is not the case.

By further exploiting the idea used in this example, one can prove the following general theorem that gives solutions to recurrences of the form (7.1). Recall that $\log_b(a)$ denotes the logarithm of a with base b, i.e., we have

$$c = \log_b(a) \iff b^c = a.$$

For example, $\log_2(8) = 3$, $\log_3(9) = 2$, $\log_4(8) = 1.5$.

The following theorem is called the *Master Theorem* (for solving recurrences). It makes life easy by allowing us to "read-off" the $\Theta(\cdot)$ expression of a recurrence without going through a proof as we did for mergeSort:

Theorem 8. *Let $n_0 \in \mathbb{N}$, $k \in \mathbb{N}_0$ and $a, b \in \mathbb{R}$ with $a > 0$ and $b > 1$, and let $T : \mathbb{N} \to \mathbb{N}$ satisfy the following recurrence:*

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < n_0; \\ aT(n/b) + \Theta(n^k), & \text{if } n \geq n_0. \end{cases}$$

Let $e = \log_b(a)$; we call e the critical exponent. *Then*

$$T(n) = \begin{cases} \Theta(n^e), & \text{if } k < e \quad \text{(I)}; \\ \Theta(n^e \lg(n)), & \text{if } k = e \quad \text{(II)}; \\ \Theta(n^k), & \text{if } k > e \quad \text{(III)}. \end{cases}$$

The theorem remains true if we replace $aT(n/b)$ in the recurrence by $a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil)$ for $a_1, a_2 \geq 0$ with $a_1 + a_2 = a$.

In some situations we do not have a $\Theta(n^k)$ expression for the extra cost part of the recurrence but do have a $O(n^k)$ expression. This is fine, the Mater Theorem still applies but gives us just an upper bound for the runtime, i.e., in the three cases for the value of $T(n)$ replace Θ by O.

Examples 9.

(1) Now reconsider the recurrence for the running time of mergeSort:

$$T_{\mathsf{mergeSort}}(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ T_{\mathsf{mergeSort}}(\lceil n/2 \rceil) + T_{\mathsf{mergeSort}}(\lfloor n/2 \rfloor) + \Theta(n), & \text{if } n \geq 2. \end{cases}$$

In the setting of the Master Theorem, we have $n_0 = 2$, $k = 1$, $a = 2$, $b = 2$. Thus $e = \log_b(a) = \log_2(2) = \lg(2) = 1$. Hence $T_{\mathsf{mergeSort}}(n) = \Theta(n\lg(n))$ by case (II).

(2) Recall the recurrence for the running time of binary search:

$$T_{\mathsf{binarySearch}}(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ T_{\mathsf{binarySearch}}(\lfloor n/2 \rfloor) + \Theta(1), & \text{if } n \geq 2. \end{cases}$$

Here we take $n_0 = 2$, $k = 0$, $a = 1$, $b = 2$. Thus $e = \log_b(a) = \log_2(1) = 0$ and therefore $T_{\mathsf{binarySearch}}(n) \in \Theta(\lg(n))$ by case (II) of the Master Theorem.

(3) Let T be a function satisfying

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ 3T(n/2) + \Theta(n), & \text{if } n \geq n_0. \end{cases}$$

Then $e = \log_b(a) = \log_2(3) = \lg 3$. Hence $T(n) \in \Theta(n^{\lg(3)})$ by case (I).

(4) Let T be a function satisfying

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < 2; \\ 7T(n/2) + \Theta(n^4), & \text{if } n \geq n_0. \end{cases}$$

Then $e = \log_b(a) = \log_2(7) < 3$. Hence $T(n) \in \Theta(n^4)$ by case (III) of the Master Theorem.

Exercises

1. Solve the following recurrences using the Master Theorem:

   (a) $T(n) = 3T(n/2) + n^5$

   (b) $T(n) = 4T(n/3) + n$

   (c) $T(n) = 8T(n/3) + n^2$

   (d) $T(n) = 8T(n/3) + n$

   (e) $T(n) = 8T(n/3) + n\lg(n)$

   Remark: The Master Theorem is not directly applicable in this case. Find a way to use it anyway. In each case $T(n)$ is assumed to be $\Theta(1)$ for some appropriate base case.

2. Let $T(n)$ be the function satisfying the following recurrence:

$$T(n) = \begin{cases} 3, & \text{if } n = 1; \\ T(n-1) + 2n, & \text{if } n \geq 2. \end{cases}$$

   Solve the recurrence and give an explicit formula for T.

   Try to derive a general formula for the asymptotic growth rate of functions satisfying recurrences of the form

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq n_0; \\ T(n-k) + f(n), & \text{if } n > n_0, \end{cases}$$

   where $k, n_0 \in \mathbb{N}$ such that $1 \leq k \leq n_0$ and $f : \mathbb{N} \to \mathbb{N}$ is a function.

3. Prove that $\sum_{i=0}^{\lg(n)-1} 2^i \Theta(n/2^i) = \Theta(n \lg(n))$, this was used in solving the recurrence for $T_{\mathsf{mergeSort}}(n)$. This is much easier than it looks, remember that to say $f = \Theta(g)$ means there is an $n_0 \in \mathbb{N}$ and constants $c_1, c_2 \in \mathbb{R}$ both strictly bigger than 0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. Recall also the formula for summing geometric series: $\sum_{i=0}^{s} r^i = (1 - r^{i+1})/(1 - r)$.