

SORTING

Some of the techniques of this course can be applied to the following problem:

The sorting problem: Given n different numbers a_1, \dots, a_n , arrange them in increasing order.

Two commonly used algorithms for this problem are “bubble sort” and “merge sort.” We want to study the worst-case number of steps these algorithms require.

In these and other sorting algorithms, the “atomic action” is looking at two numbers a_i and a_j and determining whether $a_i < a_j$ or $a_j < a_i$. If we call this action a *comparison*, then what we seek is the worst-case number of comparisons required to put n different numbers in increasing order.

Bubble sort: (This algorithm also goes by other names; it is called selection sort in the textbook.)

1. Go through all the numbers and find the largest one. Put it last.
2. Repeat the process with the remaining numbers.

The first time through, it takes $n - 1$ comparisons to find the largest number. (We compare a_1 with a_2 , then we compare the winner with a_3 , and so forth.) Then we repeat the process with the remaining $n - 1$ numbers, using $n - 2$ comparisons. And so forth, until there is only one number left (the smallest one). The total number of comparisons is

$$(n - 1) + (n - 2) + \cdots + 1 + 0 = (n - 1)n/2$$

so the number of comparisons is $\Theta(n^2)$. Thus bubble sort is a polynomial-time algorithm, which is good. But can we do better? Sorting is such a common operation, often applied to huge files, that having a faster algorithm can be quite helpful.

Merge sort: See page 176 or 310. This is also a recursive algorithm.

1. Sort the first half of the numbers.
2. Sort the second half.
3. Merge the two sorted lists.

And to sort each half, we use this same algorithm.

How many comparisons does merge sort need for n numbers? Let c_n be the number of comparisons. Then sorting the first half will take $c_{\lfloor \frac{n}{2} \rfloor}$ comparisons, and similarly sorting the second half will take $c_{\lfloor \frac{n+1}{2} \rfloor}$ comparisons (where for odd n we put the extra number into the second half). Then merging the two sorted lists requires, in the worst case, $n - 1$ comparisons (see pages 174-175 or 308-309). Therefore c_n satisfies the recurrence relation

$$c_n = c_{\lfloor \frac{n}{2} \rfloor} + c_{\lfloor \frac{n+1}{2} \rfloor} + n - 1$$

and the initial condition $c_1 = 0$.

From these two equations, we can calculate c_n for each n :

$$c_1 = 0, \quad c_2 = 1, \quad c_3 = 3, \quad c_4 = 5, \quad c_5 = 8,$$

and so forth. But to determine the growth rate of c_n as n increases, we want an explicit solution of the recurrence relation.

First, in the special case where n is a power of 2 (so that cutting the list in half at each stage is simple), then the methods of the section *Solving recurrence relations* yield the solution

$$c_{2^k} = (k - 1)2^k + 1$$

(see page 177 or 311). The correctness of this solution is easy to verify by induction on k .

Secondly, for any n we can put it in between two powers of 2:

$$2^{k-1} < n \leq 2^k \quad \text{so that} \quad k - 1 < \log_2 n \leq k.$$

Then we have

$$c_{2^{k-1}} \leq c_n \leq c_{2^k}.$$

and therefore

$$(k-2)2^{k-1} + 1 \leq c_n \leq (k-1)2^k + 1.$$

We can convert this to all- n inequalities

$$[(\log_2 n) - 2] \frac{n}{2} + 1 \leq c_n \leq (\log_2 n)(2n) + 1$$

(where we have used the facts that $n/2 \leq 2^{k-1}$ and $2^k < 2n$). Here we have c_n bounded by two functions, each of which is $\Theta(n \log_2 n)$. From this it follows that c_n itself is $\Theta(n \log_2 n)$:

Theorem 7.3.10: The number of comparisons for the merge sort algorithm applied to n numbers is $\Theta(n \log_2 n)$ in the worst case.

We know from our work in §4.3 that $n \log_2 n$ grows more slowly than n^2 . This shows that merge sort is faster than bubble sort, for large n .

Can we do even better? Here the answer is No. We need to imagine some unspecified algorithm, sorting n numbers by comparing two at a time. The work can be pictured in tree form. We start at the root, and make a comparison. Based on the outcome of that comparison (and there are two possible outcomes of a comparison), we go to one or the other of the children of the root. And in general, at each vertex, we compare two numbers. Based on the outcome, we either announce that we know the answer (in which case the vertex is a leaf), or we move down to one of the two children.

This picture is on page 299 or 417 (for the case $n = 3$). That is, the algorithm determines a rooted “decision” tree, in which each vertex has at most two children. There must be at least $n!$ leaves of the tree, because there are $P(n, n) = n!$ different possible answers to the sorting problem. And the worst-case number of comparisons is the longest length of a simple path from the root to a leaf. This is simply the height of the tree.

That is, for any algorithm, the worst-case number of comparisons is the height of rooted tree with at least $n!$ leaves, where each vertex has at most two children.

Theorem (page 287 or 405): Let T be a rooted tree of height h , in which each vertex has at most k children. Then the number of leaves is at most k^h .

Proof: The number of vertices of level 1 is at most k , the number of vertices of level 2 is at most k^2 , and (by induction), the number of vertices of level q is at most k^q . All the leaves have level h or smaller. The maximum possible number of leaves is k^h . \dashv

Apply this theorem to our analysis of a sorting algorithm. Here $k = 2$; each vertex has at most two children. And the number of leaves is at least $n!$. The conclusion is that

$$n! \leq \text{number of leaves} \leq 2^h$$

where h is the height. And therefore $\log_2(n!) \leq h$. Moreover, h is the worst-case number of comparisons. Thus we come to the following result.

Theorem (page 300 or 418): For any sorting algorithm based on comparisons, the number of comparisons to sort n numbers is at least $\log_2(n!)$ in the worst case.

Combining this result with the earlier one, we see that sorting can be done (by merge sort) in a number of comparisons that is $\Theta(n \log_2 n)$, but that we cannot hope to do better than $\log_2(n!)$. How do these upper and lower bounds compare? They are the same, up to Θ -equivalence!

Theorem: The functions $n \log_2 n$ and $\log_2(n!)$ are Θ of each other. (This is Example 4.3.9 on page 161f of the section *Analysis of algorithms*.)

In addition to the proof in the book, there is another argument, using Stirling's asymptotic approximation for the factorial function:

$$n! \sim (n/e)^n \sqrt{2\pi n}$$

Being an asymptotic approximation means that their ratio converges to 1

$$\frac{n!}{(n/e)^n \sqrt{2\pi n}} \rightarrow 1$$

as n goes to ∞ . Taking logarithms (the log function is continuous, so it preserves limits), we get

$$\log_2(n!) - n \log_2 n + n \log_2 e - \frac{1}{2} \log_2 n - \frac{1}{2} \log_2 2\pi \rightarrow 0.$$

Now divide through by $n \log_2 n$:

$$\frac{\log_2(n!)}{n \log_2 n} - \frac{n \log_2 n}{n \log_2 n} + \frac{n \log_2 e}{n \log_2 n} - \frac{\frac{1}{2} \log_2 n}{n \log_2 n} - \frac{\frac{1}{2} \log_2 2\pi}{n \log_2 n} \rightarrow 0.$$

Now let's examine the five fractions on the left side, one-by-one. The fifth one converges to 0 (as $n \rightarrow \infty$). The fourth one converges to 0. The third one converges to 0. And the second one *is* 1. We conclude that the first one converges to 1. \dashv

To summarize: For the sorting problem, on the one hand we have the merge sort algorithm, for which the worst-case number of comparisons is $\Theta(n \log_2 n)$. On the other hand, no other algorithm can do better than this, up to a multiplicative constant.

(It is interesting how many different parts of Math 61 are involved in reaching this conclusion.)