SI 335, Unit 2: Sorting Part I

Daniel S. Roche (roche@usna.edu)

Spring 2013

1 Quadratic-Time Sorting

The Sorting problem is perhaps the most well-studied (and widely taught) algorithmic problems in computer science. There are a lot of reasons for this: the problem is very important and comes up in a number of practical situations, there are non-trivial fast algorithms that make a big difference even on relatively small-sized problems, and the problem is very well-understood so we can answer almost all the hard questions about it.

In summary: you're going to learn about sorting in an algorithms class. Formally, we will define the problem as taking an input array A containing n objects that are *comparable*, and producing an array with the same objects, in increasing order. (By "comparable" we just mean that any one object is always less than, equal to, or greater than any other object.)

Here are two algorithms you should already be pretty familiar with:

SelectionSort

```
def selectionSort(A):
    for i in range(0, len(A)-1):
        m = i
        for j in range(i+1, len(A)):
            if A[j] < A[m]:
                m = j
        swap(A, i, m)
```

InsertionSort

```
def insertionSort(A):
    for i in range(1, len(A)):
        j = i - 1
        while j >= 0 and A[j] > A[j+1]:
            swap(A, j, j+1)
            j = j - 1
```

The first thing to ask is always how do we know that the algorithm is correct. In both these cases, the algorithms work by iteratively maintaining a sorted sub-array in $A[0..i]$, and growing i. You should be able to come up with a loop invariant for either of these algorithms that could be used to more formally show they are correct.

The second question we ask is usually about performance. In this case, we are interested in the worst-case running time as a function of n. For both algorithms, it is easy to see that there are two nested loops, each of which runs at most n times. So the total worst-case cost of each is $O(n^2)$.

1.1 Summations: Arithmetic Sequence

But what we really want to *compare* algorithms is a tight big-Θ bound on the worst-case cost. For SelectionSort this is pretty easy. Neither of the nested loops can every terminate early, meaning they will run the same number of times on any size-n input. So we can just calculate the number of times step 4 is executed:

$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1)$

Now we could split this last summation into three parts and use formulas to sum them up separately. But it would be much easier to write out a few terms of the summation to get the pattern, and then re-write it in an easier form:

$(n-1) + (n-2) + (n-3) + \cdots + 1 = \sum_{i=1}^{n-1} i.$

Now we need a formula. This is called an *arithmetic series*, which means that the difference between successive terms is constant (in this case, it's 1). The general formula is

$\sum_{i=0}^{m} (a+bi) = (m+1)(2a+bm)/2.$

In this formula, a is the smallest term in the series, b is the difference between consecutive terms, and $a+bm$ is the last term in the series. For the summation at hand, we have $a=1$, $b=1$, and $a+bm = n-1$. which gives $m = n-2$. So the total sum is $n(n-1)/2$. Because this is $\Theta(n^2)$, so is the worst-case running time of SelectionSort.

1.2 Worst-Case Family of Examples

Analyzing the worst-case of InsertionSort is not so easy. We know that there is an $O(n^2)$ upper bound on the running time, but what about a matching *lower bound*? Who's to say that the second condition of the inner while loop doesn't always make the loop terminate early?

What we need for a big-Ω bound is a *family of examples*, for arbitrarily large sizes n, that give the worst-case performance. Fortunately, this is a pretty simple algorithm, and we can see that the worst-case example is when the array is sorted in *reverse order*, i.e. from highest to lowest. Then the second condition of the inner while loop is *never* true, so the total number of iterations through step 4 is exactly

$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = n(n-1)/2,$

the same as SelectionSort. From this family of examples, we have a lower bound of $\Omega(n^2)$ for the worst-case cost, which matches the upper bound from before to give a worst-case cost of $\Theta(n^2)$ for the algorithm.

1.3 Recursive Analysis

The SelectionSort algorithm is also very easy to write recursively:

SelectionSort

```
def selectionSortRec(A, start=0):
    if (start < len(A) - 1):
        m = minIndex(A, start)
        swap(A, start, m)
        selectionSortRec(A, start + 1)
```

Of course then we also have to define the minIndex function, whose expected behavior should be pretty obvious:

minIndex

```
def minIndex(A, start=0):
    if start >= len(A) - 1:
        return start
    else:
        m = minIndex(A, start+1)
```

```
    if A[start] < A[m]:
        return start
    else:
        return m
```

Now let's analyze the running time of these algorithms. For recursive algorithms, we define a function, generally $T(n)$, to denote the worst-case number of primitive operations required for an input of size n. Then we *model the structure of the recursion with an equation.* That is, because the function is recursive (calls itself on smaller inputs), we can write $T(n)$ in terms of smaller values of the function T.

First, for minIndex, we have

- $T(n) = 1$ when $n \leq 1$ (the base case)
- $T(n) = 4 + T(n-1)$ when $n \geq 2$ (the recursive case)

This kind of formula, where a function is written in terms of itself, is called a *recurrence relation.* Eventually we will notice some patterns and be able to write down what $T(n)$ is right away. But for now, let's expand it a bit to figure it out:

$T(n) = 4 + T(n-1) = 4 + 4 + T(n-2) = 4 + 4 + 4 + T(n-3)$

Do you see the pattern developing? Let's make a variable k for the number of recursive steps and write $T(n) = 4k + T(n-k)$. Now we know that $T(1) = 1$, so when $n - k = 1$, we are done. So just plug in $n - 1$ for k and we have it: $T(n) = 4(n-1) + 1 = 4n - 3$.

Therefore the worst-case cost of minIndex is $\Theta(n)$. Now let's play the same game with the SelectionSort algorithm. We'll leave $T(n)$ as the worst-case cost of minIndex, and define $S(n)$ to be the worst-case cost of SelectionSort. Here's the recurrence relation:

- $S(n) = 1$ when $n \leq 1$
- $S(n) = 3 + T(n) + S(n-1)$ when $n \geq 2$

Fortunately, we already figured out that $T(n) = 4n - 3$, so the second part becomes $S(n) = 4n + S(n-1)$. Now we start expanding and find the pattern:

$S(n) = 4n + S(n-1) = 4n + 4n + S(n-2) = 4n + 4n + 4n + S(n-3)$

See the pattern? It's $S(n) = 4nk + S(n-k)$. Again, since $S(1) = 1$, we plug in $n - 1$ for k and get $S(n) = 4n(n-1) + S(1) = 4n^2 - 4n + 1$. Therefore SelectionSort is (once again) $\Theta(n^2)$. Hooray!

1.4 Noticing Patterns

As we start to see more and more algorithms, it's important to recognize similarities, or patterns, in the way they are constructed. This can lead to insights into *algorithm design.*

What commonality do you see between the two sorting algorithms above? They can both be phrased as iterated two-step procedures:

- **Pick**: Pick an element in the unsorted part of the array
- **Place**: Insert that element into the sorted part of the array

InsertionSort has a trivial "pick" step (just take the next element, $O(1)$ time), and a complicated "place" step (insert into its position, $O(n)$ time). SelectionSort is the opposite: it has a complicated $O(n)$ "pick" (finding the minimum element) and a trivial $O(1)$ "place".

Another pattern we could put these algorithms into is that of inserting repeatedly into a priority queue and then repeatedly calling delete-min on that priority queue. Of course this also leads to the worst-case $O(n \log n)$ HeapSort algorithm that you learned in Data Structures class.

2 MergeSort

Another sorting algorithm you should be familiar with already is MergeSort, which is also a good example of an algorithm that would be rather painful to write iteratively:

MergeSort

```
def mergeSort(A):
    if len(A) <= 1:
        return A
    else:
        m = len(A) // 2
        B = A[0 : m]
        C = A[m : len(A)]
        mergeSort(B)
        mergeSort(C)
        A[:] = merge(B, C)
```

Once again, we have to specify the helper function Merge as well. Here the input will be two arrays, B and C.

Merge

```
def merge(B, C):
    A = []
    i, j = 0, 0
    while i < len(B) and j < len(C):
        if B[i] <= C[j]:
            A.append(B[i])
            i = i + 1
        else:
            A.append(C[j])
            j = j + 1
    while i < len(B):
        A.append(B[i])
        i = i + 1
    while j < len(C):
        A.append(C[j])
        j = j + 1
    return A
```

2.1 Divide and Conquer

Let's notice another pattern. MergeSort is the first example we have seen of a *divide and conquer* algorithm. This is a general paradigm for designing algorithms that works in three basic steps:

1. Break the problem into similar subproblems
2. Solve each of the subproblems recursively
3. Combine the results to solve the original problem.

In MergeSort, the first step is trivial: we break into subproblems just by dividing the array in half. The second step is the two recursive calls, and the third step is accomplished by the Merge subroutine, which combines the two sorted sublists back into one.

We'll see many more examples of this approach to algorithm design as we go along in the class. So add it to your toolbelt! When you approach a new problem, think about whether it can be solved by breaking into subproblems and using recursion.

2.2 Analyzing Merge

I'll leave it to you to show that these algorithms are both correct. Once we've done that, the first task is to figure out the worst-case cost of the Merge algorithm.

For this analysis, let m be len(B) and n be len(C).

We have three while loops, and they all have at most 4 primitive operations. Therefore the total cost is the sum of the number of iterations through each loop, times 4.

Unfortunately, this is a bit challenging to nail down precisely. We see that either i or j increases each time through the first loop, so the number of iterations there is at least $\min(m, n)$ and at most $m + n$. The number of iterations through the next two loops is at least 0 and at most m and n, respectively.

This tells us that the worst-case cost is $\Omega(\min(m, n))$ and $O(m + n)$. But since these aren't the same, we can't conclude any big-Θ bound.

So let's be more precise. As is often the case, the problem practically solves itself when we specify the right things to count. Say i_0 and j_0 are the values of i and j, respectively, after the first while loop. The number of iterations through each while loop then becomes:

- $i_0 + j_0$
- $m - i_0$
- $n - j_0$

Adding these up, the total worst-case cost is $4(m + n)$, which is $\Theta(m + n)$.

2.3 Analyzing MergeSort

Now it's time to analyze the recursive MergeSort function. Once again, we'll write a recurrence relation that just sums up the costs of all the steps, for each case. Here $T(n)$ is the worst-case number of primitive operations required to execute MergeSort on a size-n array.

- $T(n) = 1$ when $n \leq 1$
- $T(n) = 5 + T(m) + T(n - m) + 4(m + n - m)$ when $n \geq 2$

(Notice for the second case we used the derived cost of Merge from above.)

The recursive case simplifies somewhat to $T(n) = 5 + 4n + T(m) + T(n - m)$, but this is still unsatisfactory. First, since we know $5 + 4n \in \Theta(n)$, let's just write that part as n.

Second, we'd like to use the fact that m is very close to $n/2$, and combine the last two terms into one. To do this, we'll use the inequality $m = \lfloor n/2 \rfloor \leq \frac{n+1}{2}$, and $n - m \leq \frac{n+1}{2}$ as well. Now it's important to recognize that, since we're simplifying with inequalities, **we are now working on a big-O upper bound, not a big-*Theta* bound**.

Using these simplifications, let's try our old trick of expanding this out until we see a pattern:

- $T(n) < n + 2T(\frac{n+1}{2})$
- $T(n) < n + 2(\frac{n+1}{2} + 2T(\frac{n+3}{4})) = 2n + 1 + 4T(\frac{n+3}{4})$
- $T(n) < 2n + 1 + 4(\frac{n+3}{4} + 2T(\frac{n+7}{8})) = 3n + 4 + 8T(\frac{n+7}{8})$
- $T(n) < 3n + 4 + 8(\frac{n+7}{8} + 2T(\frac{n+15}{16})) = 4n + 11 + 16T(\frac{n+15}{16})$

OK, this time the pattern is a little more difficult to spot. Hopefully we recognize the linear term looks like in after i recursive calls. The recursive term is a little more complicated, but not too unfamiliar; it looks like $2^i T(\frac{n+2^i-1}{2^i})$ after i recursive steps. That's a little too messy, so let's instead use the fact that it's less than $2^i T(\frac{n}{2^i}+1)$.

The tricky part is the sequence of constants $0, 1, 4, 11, \ldots$. There are a couple of ways to go with this. One is to recognize that this is always less than a corresponding power of 2, in particular, 2^i, after i recursive steps. For those of us with weaker powers of insight, there is the almighty Online Encyclopedia of Integer Sequences (OEIS). Type the sequence above into this search engine and it'll tell you (along with a ton of other information) that the general formula is $2^i - i - 1$.

Putting all this together, we have, after i recursive steps, $T(n) < in + 2^i + 2^i T(\frac{n}{2^i}+1)$. It took some tricky insights to get here! But now our job to solve this recurrence is pretty mechanical. We know the base case is when the argument to $T(n)$ satisfies $n < 2$. Plugging in the formula above to this inequality will give us an *upper bound* on how many recursive steps there are until we reach a base case:

$\frac{n}{2^i} + 1 < 2$, means that $n < 2^i$, and therefore $i > \lg n$.

Now we'll set $i = \lg n + 1$, which definitely satisfies that inequality, and plug into the formula for T:

$T(n) < (\lg n + 1)n + 2^{\lg n+1} + 2^{\lg n+1} T(1) = n \lg n + 5n$

Therefore $T(n)$ is $O(n \log n)$. Thus concludes our first challenging bit of formal analysis. Take a moment to reflect.

So far we have only shown an upper bound on the worst-case cost of MergeSort. To show a matching lower bound, we could look again at a worst-case family of examples. I would just pick $n = 2^k$, i.e., when the size of the array is exactly a power of two. Then the recurrence looks like $T(n) = n + 2T(\frac{n}{2})$, with no ugly floors or complicated fractions, which is much easier to solve to get the matching $\Omega(n \log n)$ lower bound.

2.4 Space Complexity of MergeSort

We know that MergeSort is $O(n \log n)$ time, but besides processor usage another important computational resource is *memory*. In fact, for some applications, the space that the algorithm uses might be even more important than the time!

The basic rules and processes we have been using for formal analysis will still apply when we are measuring space instead of time. But the measurement itself, at the beginning, is a little bit different. The key peculiarity is that time is always *additive*, but space isn't necessarily.

To measure space, we are concerned only about the maximum amount of memory used by our algorithm at any one time. For the MergeSort algorithm, this makes a really big difference compared to time, because we can *reuse the same space for both recursive calls*. Then, observing that the Merge algorithm allocates exactly n units of memory, the total amount of extra memory allocated in a call to MergeSort, for an input of size n, is given by

- $S(n) = 0$ if $n = 1$
- $S(n) = n + S(\frac{n}{2})$ if $n \geq 2$

See the crucial difference? Because we can use the same memory for both recursive calls, there is no coefficient of "2" in front of the recursive term. Now let's write out some terms and try to solve:

$S(n) = n + S(\frac{n}{2}) = n + \frac{n}{2} + S(\frac{n}{4}) = n + \frac{n}{2} + \frac{n}{4} + S(\frac{n}{8})$

See the pattern? After i recursive calls, $S(n) = \sum_{j=0}^{i-1} \frac{n}{2^i} + S(\frac{n}{2^i})$. Just like before, the base case is reached when $i = \lg n$, and the final cost in terms of memory usage is

$S(n) = \sum_{j=0}^{\lg n - 1} \frac{n}{2^i}$.

Oh no! It's another summation! We already saw how to do a summation of an *arithmetic series*, where the difference between consecutive is a constant amount. For *geometric series*, the quotients between consecutive terms is a constant amount.

In the example here, the sum looks like

$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 8 + 4 + 2$

Take any two adjacent terms, divide them, and you'll get either 2 or $\frac{1}{2}$. That's what it means for this to be a geometric series.

You can read online or in your book all about geometric series. The important thing to remember is the universal formula:

> **Geometric Series** For any real number $r > 0$ and integer $n \geq 0$,
> $\sum_{i=0}^{n} r^i = (r^{n+1} - 1)/(r - 1)$

Using this formula in the summation above, we conclude that $S(n) = 2n - 2$, which is $\Theta(n)$. So the amount of extra memory required by MergeSort is only linear even though the running time is $\Theta(n \log n)$. Interesting!

3 Lower Bound for Sorting

We have seen a few algorithms for sorting: some have worst-case $\Theta(n^2)$ running time and some have worst-case $\Theta(n \log n)$. A natural question to ask is whether this is the best possible, or if there is some faster sorting algorithm that we just haven't found yet.

If the answer is the first one, how could we know this? How could we be *absolutely sure* that no faster algorithm exists? In general, proving something is impossible is very difficult (but not impossible!).

The trick we usually use to prove lower bounds on the difficulty of problems is to define very carefully a *model of computation*, and only consider possible algorithms that follow the rules of our chosen model. This is more restrictive of course than allowing any algorithm whatsoever, but it allows us to draw more powerful conclusions. The hard part is coming up with a model that is restrictive enough so that we can prove something interesting, but general enough so that whatever we prove has wide applicability.

3.1 Comparison Model

Our lower bound for sorting will use the *comparison model*. In this model, the input comes in an array of elements, and the ONLY things any algorithm is allowed to do with those elements are:

- Move them around (swap, copy, etc.)
- Compare two of them (less-than, greater-than, etc.)

Any sorting algorithm which only accesses the input elements in these ways is called a *comparison-based sort*. As it turns out, *every* algorithm for sorting we have looked at so far is comparison-based. But this would also be a good time to stop and think of what kinds of operations this model restricts...

3.2 Permutations

A *permutation* of n elements is just an ordering of them. So if I asked a class of 20 students to line up, the ordering of that line-up would define a permutation of the set of 20 students. How many such orderings are there?

If you were paying attention in Discrete then you know that there are $20 \times 19 \times 18 \times \cdots \times 2 \times 1$ such orderings. This is called the *factorial* function, and because it's so exciting we use an exclamation mark to denote it. So the product above is 20!.

You should also know that this function grows rather quickly. 20 students can line up in 20! different ways, which is a whopping 2432902008176640000 different ways. (Allow the enormity of this number, compared to 20, to sink in. If there were just 60 students, the number of ways in which they could line up is more than the number of atoms in the entire universe. How many ways could the whole Brigade line up single-file?)

How does this relate to sorting? Well, in the comparison model at least, all that really matters about the input is what *ordering* the input array is in. For example, if the input array is [3 1 2 0] or [231 18 19 −40], any comparison-based sort will behave exactly the same, because all the comparisons between elements will be exactly the same.

The consequence is that, in the comparison model, there are really only $n!$ different inputs of size n for sorting. This is still a huge number, but it's not infinite. Isolating which inputs are really different from the viewpoint of the algorithm is crucial to the lower bound proof that follows.

3.3 Facts about logs

This is a good time to take a break and review some facts about logarithms that you should already know. For any positive numbers a, b, c, the following hold true:

- $\log_b b = 1$
- $\log_b ac = \log_b a + \log_b c$
- $\log_b a^c = c \log_b a$
- $\log_b a = (\log_c a)/(\log_c b)$
- $a^{\log_b c} = c^{\log_b a}$
- $(a^b)^c = (a^c)^b = a^{bc}$

3.4 The Proof

To prove the lower bound on any comparison-based sort, we will assume the input has n elements and they are all distinct. Making this kind of restriction is *totally fine* for proving a lower bound, because any fully correct algorithm would have to handle at least these inputs.

There are a few key ideas of this lower bound proof. Make sure you understand each step.

1. **A correct algorithm must take different actions for each of the possible input permutations.** For each of the $n!$ possible orderings of an n-element array, the swapping, copying, and so on that the algorithm performs must be different. If an algorithm took the same actions for two different permutations, then at least one of them must end up in the wrong order at the end.
2. **The choice of actions is determined only by comparisons.** The comparison model dictates that the only *information* the algorithm can get about its input (other than the length) is by comparing different elements in the array.
3. **Each comparison has two outcomes.** This comes from the fact that we assumed all the array elements are distinct. So comparing any two elements in the input either produces "less than" or "greater than".
4. **An algorithm that performs c comparisons can only take 2^c different actions.** This comes from the previous two points. If there were only one comparison, there would only be two possible different actions the algorithm could take. With 2 comparisons, there would be up to 4 possible actions for the algorithm. This generalizes to 2^c possible actions for c comparisons.
5. **The algorithm must perform at least $\lg n!$ comparisons.** From the first point and the previous one, we can conclude that we must have $2^c \geq n!$. Solving for c, the number of comparisons, gives $c \geq \lg n!$.

Now it's just a little math. We know that the factorial function gives really big numbers, but we also know that the logarithm makes big numbers get a lot smaller. So how big is $\lg n!$?

In this case, we are only interested in a lower bound. Remember that $n!$ is the product of all the integers up to n. At least half the integers in this product are greater than or equal to $n/2$. Therefore $n! \geq (n/2)^{n/2}$.

Now let's take the logarithm of this and use the facts we know from above.

$\lg n! > \lg(n/2)^{n/2} = \frac{n}{2} \lg \frac{n}{2}$

Therefore $\lg n! \in \Omega(n \log n)$. Finally, since each comparison will be at least 1 primitive operation in the algorithm, the total number of primitive operations in ANY comparison-based sort is also bounded below by $\Omega(n \log n)$.

3.5 Consequences

Wow. We've proven that our $O(n \log n)$ algorithms for sorting are somehow the best possible. We say that such algorithms are *asymptotically optimal*, at least in the comparison model, because the big-O bound of the worst-case performance can't possibly be improved.

However, let's remember the restriction we had to make to get this result: the comparison model. This actually gives us a hint as to how to "break" this lower bound: Any algorithm that wants to do better than $\Theta(n \log n)$ in the worst case will have to do something more than just compare array elements for less-than or greater-than. Is it possible? Wait a few weeks and we'll find out.