Lecture Notes for CSCI 311: Algorithms

Set 2-Example Algorithm

Professor Talmage
Based on *Introduction to Algorithms* by Cormen, Leiserson, Rivest, Stein

January 19, 2024

---

1 Algorithms

Exercise:

Try to give a **formal** definition of an algorithm.

Definition 1. An *algorithm* is a finite, well-defined sequence of computation steps that solves a problem.

Let us break down the pieces of that definition:

- Problem: A *problem* is a relation between input values from some domain I and "correct" output values from some range O. That is, $P \subseteq I \times O$. A pair $(i, o) \in P$ is a *problem instance*. You can think of a problem as a function, mapping inputs to outputs, though it actually maps an input to a set of acceptable outputs.
- Well-defined: At each point in the computation, it is clear what step to take next.
- Solves: Computes the problem function–given an input, gives an output specified by the problem as correct.
- Finite: This means that an algorithm must generate its output and terminate. No running forever.
- Computation Steps: We will not go into the formal definition in this class, but can get a pretty good approximation.

  Exercise:

  This definition actually leaves out lots of things that we think of as algorithms. Try to come up with counterexamples for each part, where something that definitely seems to be an algorithm breaks the rules we have set forth.

  – We want a notion of computational steps that is independent of the programming language or hardware we are using, or else comparing algorithms will be well-nigh impossible.

  Exercise:

  Think of some somehow "basic" steps that are common across programming paradigms that you think we should count as a single step.

Exercise:

Is this a valid algorithm for the sorting problem? Why or why not?

1: on input x:
2: **return** sorted version of x

- We will generally consider the following to each be a single computation step:
  - Arithmetic
  - Assignment
  - Conditionals
  - Control statements (`if`, `while`, etc.)
  - Any command that doesn't have to work with more than a constant number of values
- When we talk more about performance, the computation step, as we define here, is our fundamental unit of time. We will see then that the differences between actual time cost of different steps (such as addition vs. multiplication) are insignificant for our needs.

1.1 Example

Consider an algorithm you have seen before: `InsertionSort`. As an example of how we will work in this class, we will express the problem and algorithmic idea and analyze it.

The first thing we need to do is express the problem formally.

Exercise:

Give a formal statement of the sorting problem.

Problem 1 (Sorting)**.** The Sorting problem maps collections of elements to a particular type of permutation:

- **Input:** n elements $x_1, \ldots, x_n$.
- **Output:** Same n elements, relabeled as $b_1, \ldots, b_n$, where $b_1 \leq b_2 \leq \cdots \leq b_n$.

Now, we need to express our algorithm. To do this, we will use *pseudocode*, which is an abstract style of writing computation steps. The idea is to keep the precision of writing computer code, but get rid of any nonessential implementation details to retain only the core ideas of our solution.

Exercise:

Discuss with a partner the idea for `InsertionSort`. Try to explain as precisely and concisely, in English, what the algorithm does.

One of the best and worst aspects of pseudocode is the fact that you can give pseudocode at different levels of abstractions. For example, we can express `InsertionSort` as follows:

1. For each element in the list,
2. insert that element, in sorted order, into a sorted version of the list

This gives the idea, and a computer scientist could likely figure out how to implement this, but it leaves a lot of details open, some of which can have significant effects on the resulting code. For example, this seems to imply that we would be inserting into a separate, sorted version of the list, while we may care that we can do this sort in-place, using less memory. It is also not entirely clear how to insert in sorted

order, as that is not a single step. So, while it is good to start you algorithm development and presentation with a very high-level description of the idea like this, final pseudocode should be a bit more precise.

Algorithm 1 High-level pseudocode to Insertion Sort a list $X = [x_1, \ldots, x_n]$

```
1: function InsertionSort(X)
2:     for currElem in X do
3:         while currElem is smaller than the element immediately before it do
4:             Swap currElem and the element before it
5:         end while
6:     end for
7:     return X
8: end function
```

This is a fairly typical abstraction level of pseudocode in research papers. It still leaves some things, such as swapping two elements, unspecified, but these are smaller steps that are less ambiguous and allow less variation in the performance of the final algorithm. For example, swapping two elements is constant time, while inserting in sorted order is not. The weakness of this intermediate level of abstraction is that it can be harder to argue precisely about the behavior of the algorithm. As we will discuss later, if the differences between possible implementations are constant factors, then this level of detail is probably sufficient. For illustration, though, we will give one more, extra-precise version of the pseudocode and evaluate that version.

Consider the following pseudocode precisely expressing the `InsertionSort` algorithm.

Algorithm 2 Pseudocode to Insertion Sort a list $X = [x_1, \ldots, x_n]$.

```
1: function InsertionSort(X)
2:     for j = 2 to n do                          ▷ Use python notation (:, indent)
3:         key = X[j]
4:         i = j − 1
5:         while i > 0 and X[i] > key do          ▷ Use C/java notation (parens,braces)
6:             X[i + 1] = X[i]
7:             i − −
8:         end while
9:         X[i + i] = key
10:    end for
11:    return X
12: end function
```

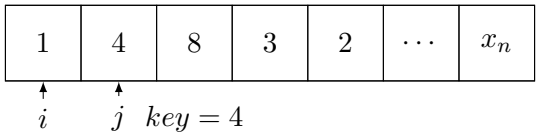
Note that:

- I mixed loop styles deliberately to show that different styles work. Do not do this.
- Pseudocode exists to clearly communicate an algorithmic idea, and consistency is essential to clarity.
- You can use either of the styles shown here, or that provided in the LaTeX package we will look at tomorrow, but you need to be consistent.

Exercise:

Before we analyze this algorithm, we need to understand what it is doing. Work together to reconcile the instructions here with your intuitive understanding. Draw what is happening in one or two iterations of the for loop.

We will draw the algorithm's behavior through a few steps. It is always good practice to solidify an intuitive understanding of an algorithm, often by visualizing its actions, or it will be difficult to reason properly about what it is doing.



Consider the variables in the algorithm:

- j tracks the sorted prefix.

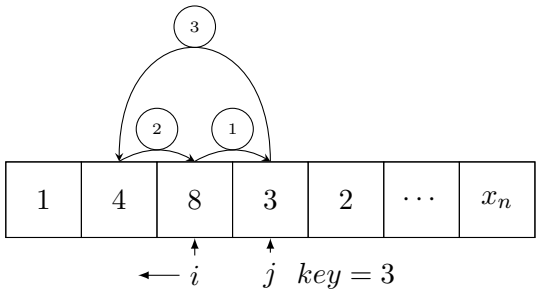
  Exercise:

  Why start at 2?

- i searches backwards to find the correct location to insert the next element.
- key stores the next element we need to insert.

As the algorithm runs,

- When $j = 2, 3$ and $key = 4, 8$, the keys are already in sorted order, so nothing interesting happens.
- When $j = 4$ and $key = 3$, we will walk i to the left, shifting first 8, then 4, to the right to make space. We then insert 3 between 1 and 4, which is the correct sorted order.
- Similarly, when $key = 2$, we will slide 8, 4, and 3 to the right to make space, and insert 2 in the correct location.



Exercise:

What is our end condition? How do we know we are done? Relate the code and the picture.

- This continues until j reaches the end of the array ($j = n$), when we insert the last key and return the (now sorted) list.

There are two main properties of algorithms on which we will focus: Correctness and Efficiency.

2 Correctness

Why do we need to talk about algorithmic correctness? Is it not obvious that we are only interested in correct solutions to problems? If an algorithm is incorrect, then it does not solve the problem, so is arguably not even an algorithm by our definition. The key here is not just having a correct algorithm, but *knowing* and *showing* that an algorithm is correct. That is, we can present claimed solutions all day long, but if we cannot verify and demonstrate that they are correct, we are wasting our time.

Exercise:

How do you determine whether an algorithm is correct? What have you done in past CS classes? How confident are you that your programs were correct?

- An algorithm is only correct if it gives the proper output for *every possible input*.
- We cannot typically test every possible input (most input spaces are infinite), so we need some way to argue generally about an algorithm's behavior. This is a *correctness proof*. Every algorithm needs one. **In this class, if you ever write an algorithm, you need to argue why it is correct (unless I specifically tell you otherwise).**
- *Formal methods* is the field of complete evaluation of algorithms. Typically, the resultant proofs are computer-generated and unreadable by humans, since they deal with many, many cases and explicitly tracing through the algorithm's logic.
- More commonly, we use *informal reasoning*, or hand-written arguments with a lot of base assumptions. This is what we will do in this class.
  - Informal reasoning is not cutting corners or waving your hands. You still need a complete argument. It just means you do not have to handle all of the low-level details, such as details of OS, programming language, compiler, etc.
  - You need to address all cases, but can rely more heavily on the reader's knowledge of fundamentals like loop behavior.

2.1 Running Example

Let us show that `InsertionSort` is correct. The key idea we will use is that it is always true that all elements before index j are sorted. Thus, when $j > n$, the entire list is sorted. We here give a non-inductive proof. We will discuss induction shortly, which could be a more intuitive proof for this particular algorithm.

Theorem 1. *`InsertionSort` solves the sorting problem.*

Aside: We could say "correctly solves", but that is somewhat redundant, as "solves" implies correctness.

Proof. Assume in contradiction that for some input array X, `InsertionSort`(X) returns an array $B = [b_1, \ldots, b_n]$ which is **not** sorted in increasing order. Then there must be some element in B which is larger than a later element in B.

Exercise:

Convince your neighbor that this out-of-order pair is logically equivalent to not being sorted.

Aside: Since we need to show that `InsertionSort` works on every possible input, we can equivalently show that there is no input on which it fails. To do this, suppose there is an input where it fails, then show that that is an impossibility. This is known as *proof by contradiction*, and is often the easiest way to deal with an infinite number of cases.

Aside: While it may seem minor, the reduction of "unsorted" to "there is an out of order pair" is one of the most important steps in the entire proof. This type of simplification is both essential and difficult to teach. Expect to spend a lot of time looking for this type of core insight to take a complex claim and reduce it something tangible and specific with which you can work.

Let b_{large} be the first element of B which is greater than a later element and b_{small} the first element after b_{large} which is smaller than b_{large}. So the array B looks something like $[b_1, b_2, \ldots, b_{large}, \ldots, b_{small}, \ldots]$. Note that $small > large$. We will first show that b_{large} and b_{small} must be adjacent in B, then conclude that that is also impossible.

Claim 1. $small = large + 1$.

Proof. Suppose not. Then for every $large < mid < small$, $b_{mid} > b_{large} > b_{small}$, by our definition of $large$ and $small$. There are two ways this could have happened:

1. Each b_{mid} was before b_{small} when b_{small} was key. In this case, the algorithm would have swapped b_{small} with $b_{small-1}$, then with $b_{small-2}$, and so on for each b_{mid}, leaving b_{small} next to b_{large}. That is, leaving $small = large + 1$. Contradiction.
2. Some b_{mid} was key after b_{small}, then the algorithm inserted it before b_{small}. But the algorithm only inserts keys left of larger elements, and $b_{mid} > b_{small}$, so this would not happen.

□

> **Aside:** A formal proof would need to go into the exact logic in the while loop condition to argue that the algorithm only inserts keys left of larger elements. An informal proof can abstract this out and assume the reader understands how a while loop works.

By the same argument, if b_{large} was ahead of b_{small} in the array when b_{small} was key, then the algorithm would have swapped b_{large} to the right of b_{small}, contradicting our assumption of their locations. If b_{large} was key after b_{small}, then the algorithm would not have swapped b_{large} to before b_{small}, so we again contradict our assumption. Thus, it is impossible for a larger element to precede a smaller one in B, and `InsertionSort` is correct. □

3 Efficiency

Exercise:

What does it mean for an algorithm to be efficient? What are you measuring? Is there a cutoff between efficient and inefficient?

Algorithms are only useful if they can actually run on real hardware and complete in a reasonable amount of time.

> **Aside:** This is a bit of an oversimplification, as there are very interesting algorithmic ideas which are not practical but from which we can learn useful things.

We can measure the efficiency or cost of an algorithm in several dimensions:

- Time: This is our primary concern
- Space: Important, as it is limited in a real computer, but it is easier to buy space than time
- Other resources: These tend to vary by field (in my research area, message size, number of messages, number of synchronization events, etc.)

We use the same techniques to describe cost in each of these dimensions. For this course, we will almost exclusively talk about time complexity, but the tools you learn will be useful for analyzing other types of cost, as well. Our primary goal is to be able to compare two algorithms for the same problem and determine which is better.

Exercise:

Suppose that we have two algorithms, A_1 and A_2, which solve the same problem. If we run A_1 on computer C_1 and A_2 on computer C_2, which will finish first? Suppose we know that A_1 is slow and A_2 is fast, while C_1 is fast and C_2 is slow. Now, can we say which will finish first?
Now assume the following, where n is the size of the input:

- A_1 completes in $2n^2$ steps.
- A_2 completes in $50n \log_2 n$ steps.
- C_1 performs 10^9 steps/second.
- C_2 performs 10^7 steps/second.

Which finishes first?

Despite this question being vague, this is exactly the kind of question we want to be able to answer. For this example, A_1 on C_1 will finish first, for n up to about 40000. Beyond that, A_2 on C_2 will finish first, despite C_2 being *two orders of magnitude* slower than C_1. Intuitively, the speed of the computers does not really matter, since that is fixed, and the time for the algorithms increases with input size. What we are considering here is the *growth rate* of the algorithms' runtime functions. That is, as the input grows, how does the time needed to solve the problem grow?

3.1 Running Example

To determine the running time of `InsertionSort`, all we need to do is count the number of steps it takes. Remember that we need to do this in terms of the input size, so first define the runtime of `InsertionSort` on an input of length n by the function $T(n)$.

Aside: This is a seemingly trivial step, but you always need to do this correctly. Many students are tripped up by not properly defining their runtime function and input size, which leads to meaningless answers. For example, if I ask for the runtime of a function $f(x)$, which calls $g(y)$, and your answer just says $T(n) = n^2$, is that the runtime of f or g? How does n relate to x or y?

Exercise:

Return to the pseudocode and annotate the number of steps we take in each line, then compile those to get a function for the entire runtime.

Algorithm 3 Pseudocode to Insertion Sort a list $X = [x_1, \ldots, x_n]$.

```
1:  function InsertionSort(X)                                   ▷ 1 step
2:      for j = 2 to n do                                       ▷ 2 steps (increment, compare), repeat body n − 1 times
3:          key = X[j]                                          ▷ 2 steps (index, assign)
4:          i = j − 1                                           ▷ 2 steps (subtract, assign)
5:          while i > 0 and X[i] > key do                       ▷ 3 steps (2 comparisons, and operator) repeat body ? times
6:              X[i + 1] = X[i]                                 ▷ 4 steps (addition, 2 indices, assignment)
7:              i − −                                           ▷ 2 steps (subtraction, assignment)
8:          end while                                           ▷ 1 step
9:          X[i + i] = key                                      ▷ 3 steps (addition, index, assignment)
10:     end for                                                 ▷ 1 step
11:     return X                                                ▷ 1 step
12: end function                                                ▷ 1 step
```

Adding these up, we get

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

First, note that you may have slightly different counts for each line. That is okay! As we will discuss in more detail next week, different constants do not change the growth rate of a function enough for us to care. Second, the seemingly extra $+3 + 1$ after the while loop and $+2 + 1$ after the for loop are because there is actually one extra check for each, in which the loop condition fails and we exit the loop.

Exercise:

What should the question mark be?

- Changes in each iteration of the `for` loop!
- At most, it is n each time, but that is pretty pessimistic.
- Really, at most $j - 1$ each time.

Exercise:

What input forces the while loop to run $j - 1$ times for each j? What would be the best-case input, and how many times would the while loop run for each iteration of the for loop on that input?

To reduce $T(n)$, we can sum over the iterations of the for loop:

$$
\begin{aligned}
T(n) &= 1 + \sum_{j=2}^{n}(2+2+2+3+(j-1)(3+4+2+1)+3+1)+2+1+1+1 \\
&= 1 + \sum_{j=2}^{n}(13+(j-1)(10))+5 \\
&= 6 + 13(n-1) + 10\sum_{j=2}^{n}(j-1) \\
&= 13n - 7 + 10\sum_{k=1}^{n-1} k \\
&= 13n - 7 + 10\frac{(n-1)n}{2} \\
&= 13n - 7 + 5n^2 - 5n \\
&= 5n^2 + 8n - 7
\end{aligned}
$$

We did that arithmetic in eye-watering detail, and in the future we will be able to bundle up constants much more easily, but I want you to understand the relation between the exact steps in the pseudocode and our final runtime functions.

Exercise:

What would happen if we had an "average" case, where each element had to move past half of the elements before it?

- Each iteration of the for loop would add $(j-1)/2$ iterations of the while loop. Completing the sum would still give a quadratic runtime function.

Exercise:

What would a best-case input look like, and how long would it take?

- Sorted input would have each while loop run zero times, as the $X[i] > key$ check would always fail. This would yield runtime linear in n.