SI 335, Unit 1: Design, Analysis, Implementation

Daniel S. Roche (roche@usna.edu)

Spring 2013

1 The course

By now, you've seen a lot of algorithms, for everything from concatenating linked lists to finding the shortest path in a graph. But how in the world does anyone come up with these algorithms? And why — what's the point?

To answer this second question, a history lesson is useful. The following is excerpted from the report *Designing a Digital Future* commissioned by President Obama and co-authored by such dignitaries as the president of Yale and the CEO of Google.

> **Progress in Algorithms Beats Moore's Law**
>
> Everyone knows Moore's Law — a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years.
>
> Fewer people appreciate the extraordinary innovation that is needed to translate increased transistor density into improved system performance. This effort requires new approaches to integrated circuit design, and new supporting design tools, that allow the design of integrated circuits with hundreds of millions or even billions of transistors, compared to the tens of thousands that were the norm 30 years ago. It requires new processor architectures that take advantage of these transistors, and new system architectures that take advantage of these processors. It requires new approaches for the system software, programming languages, and applications that run on top of this hardware. All of this is the work of computer scientists and computer engineers.
>
> Even more remarkable — and even less widely understood — is that in many areas, *performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed.*
>
> The algorithms that we use today for speech recognition, for natural language translation, for chess playing, for logistics planning, have evolved remarkably in the past decade. It's difficult to quantify the improvement, though, because it is as much in the realm of quality as of execution time.
>
> In the field of numerical algorithms, however, the improvement can be quantified. Here is just one example, provided by Professor Martin Grötschel of Konrad-Zuse-Zentrum für Informationstechnik Berlin. Grötschel, an expert in optimization, observes that a benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later — in 2003 — this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, whereas a factor of roughly 43,000 was due to improvements in algorithms! Grötschel also cites an algorithmic improvement of roughly 30,000 for mixed integer programming between 1991 and 2008.
>
> The design and analysis of algorithms, and the study of the inherent computational complexity of problems, are fundamental subfields of computer science.

So the big motivation for studying algorithms is... efficiency! The reason algorithmic thinking can be so powerful is that it is a form of *abstraction.* An algorithm is an abstraction of a computer program, and by using this abstraction we can get at the core, high-level problems that make a huge difference in practice.

(Unfortunately, this abstract thinking is also what can make this course difficult. So let me take the first of many opportunities to exhort you to *give yourself lots of time for the assignments in this class.*)

If I asked you to write a program to list all the possible schedules for a student given the list of sections and the courses they wanted to take, could you do it? Could you write a program to give schedules to all students at the Academy (or the maximum number possible) that would run in at most a day or two? Could anyone do it? These are the kind of issues we'll be thinking about: solving difficult problems, solving them fast, and figuring out when problems simply cannot be solved fast.

2 Problem, Algorithm, Program

The first thing we should probably figure out is exactly what an algorithm is. Because algorithms are all about *problem solving*, we have to start with what a problem is.

Formally, a *problem* is a collection of input-output pairs that specifies the desired behavior of an algorithm. For example, the sorting problem for lists of numbers contains pairs of lists like

- [20, 3, 14, 7], [3, 7, 14, 20]
- [13, 18], [13, 18]
- [5, 4, 3, 2, 1], [1, 2, 3, 4, 5]

(and infinitely many more). We can also think of a problem as a mathematical function, which specifies a single image (output) for each preimage (input) in its domain.

(Valid inputs for a problem are also called problem *instances* in the textbook and elsewhere.)

An *algorithm* is a specific way to actually compute the function defined by some problem. Specifically, we say that an algorithm is a solution to a problem if, for every input in the problem's domain, the algorithm produces the correct output in a finite number of steps. So for example InsertionSort and MergeSort are two different algorithms that are both solutions to the sorting problem.

Notice that the definition of a correct algorithm doesn't say anything about what the algorithm should do if the input is *not* in the problem's domain. In such cases we say the algorithm's behavior is "unspecified" — meaning that it can do anything at all. In summary, for this class (only!) you don't have to worry about producing nice error messages.

Now an algorithm is a high-level description about how to compute something — it doesn't depend on any particular machine or programming language. To get an actual program we need an *implementation* of the algorithm. If you were asked in Data Structures to write a Java program to implement MergeSort, each of you would have produced a slightly different program, different implementations of the same algorithm. Even if all the programs correctly implement the same algorithm, some might be better (faster, smaller, etc.) than others.

So the situation looks like this: For any problem there are many possible algorithms, and for any algorithm there are many possible programs. So in some sense this class is focused on the "middle man" - the stage of programming that sits between the problem specification and the actual code.

Disclaimer: Because the focus in studying algorithms is on *practical usefulnes*, almost all of the definitions above will get stretched or flat-out broken in various contexts. For example, we'll see algorithms where the input is *not* entirely known to the algorithm but given one piece at a time, and we'll even discuss algorithms that might never halt in some cases, but yet we still consider to be correct (and even efficient)! This is just to say, rules are meant to be broken, even when we make up the rules ourselves. But first we have to learn the rules...

3 Three foci of this course

There are three major components to the study of algorithms, and in this course we will concentrate on all three of them:

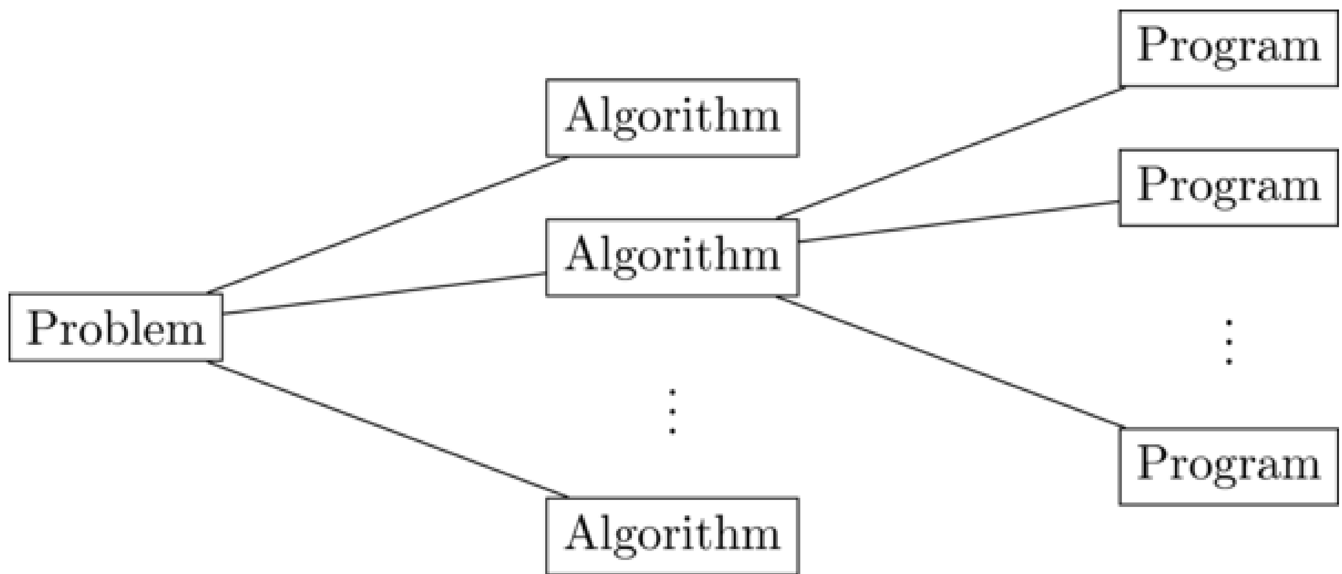

Figure 1: One-to-many relationships

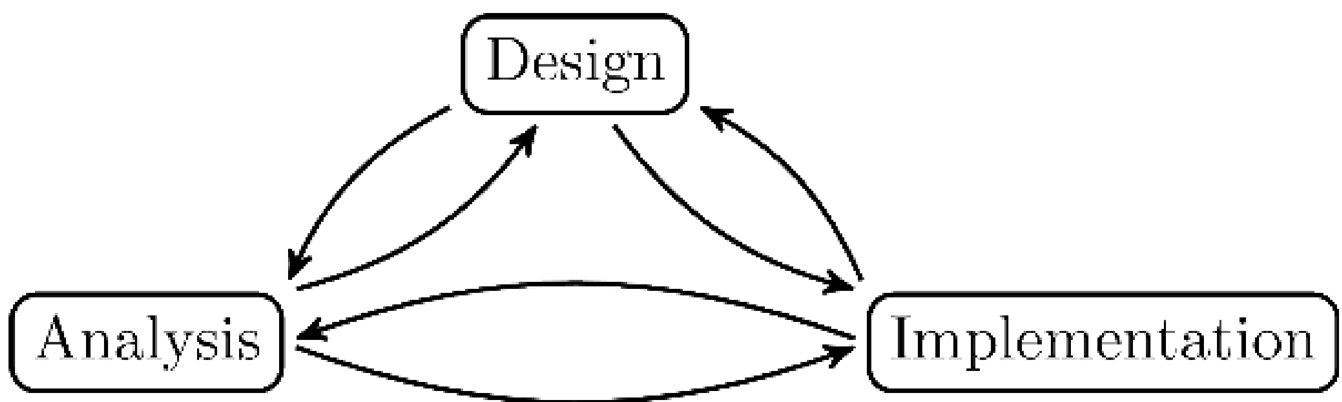

Figure 2: Components of Algorithms

- **Design**: How to come up with efficient algorithms for all sorts of problems
- **Analysis**: What it means for an algorithm to be “efficient”, and how to compare two different algorithms for the same problem.
- **Implementation**: Faithfully translating a given algorithm to an actual, usable, fast program.

Your intro CS classes mostly covered implementation (“Here is an algorithm, now write me a C++ program for it”). In Data Structures, you started to see some analysis, comparing and judging the performance of different algorithms. One thing that will be completely new in this class is a discussion of *design strategies* for algorithms — a few techniques, and a lot of practice, aimed at coming up with new algorithms for new problems.

We always have to think about all three of the aspects above, because they are all inter-related. It is easy to see how design affects analysis and implementation because this is what you are used to: given an algorithm, analyze its running time and write a program that implements it. But in this class we’ll also see other relationships. Having a deeper understanding of analysis often shows where the weaknesses are in existing algorithms and allows us to design better ones. Even more surprising, there are cases where the practical implementation results seem to contradict formal analysis, which occasionally leads to new ways of thinking about how to do the analysis in the first place! We will in fact see examples like this in this course.

4 Case Study: Array Search

Let’s look at design, implementation, and analysis for a simple problem:

Problem: Sorted array search

Input:

- A, sorted array of integers
- x, number to search for

Output:

- An index k such that $A[k] = x$, or NOT_FOUND

You have already seen this problem in previous classes, although you might not recognize it at first. What algorithms can you think of to solve it?

I can think of at least three. First, the obvious one:

Algorithm: linearSearch

Input: (A, x), an instance of the *Sorted Array Search* problem

```
def linearSearch(A, x):
    i = 0
    while i < len(A) and A[i] < x:
        i = i + 1
    if i < len(A) and A[i] == x: return i
    else: return 'NOT FOUND'
```

You should recognize this one too:

Algorithm: binarySearch

Input: (A, x), an instance of the *Sorted Array Search* problem

```
def binarySearch(A, x, left=0, right=len(A)-1):
    while left < right:
        middle = floor((left + right) / 2)
        if x <= A[middle]:
            right = middle
        elif x > A[middle]:
            left = middle+1
    if A[left] == x: return left
    else: return 'NOT FOUND'
```

Finally, for fun, here's one you might not have seen before:

Algorithm: gallopSearch

Input: (A, x), an instance of the *Sorted Array Search* problem

```
def gallopSearch(A, x):
    i = 1
    while i < len(A) and A[i] <= x:
        i = i * 2
    left = floor(i / 2)
    right = min(i, len(A)) - 1
    return binarySearch(A, x, left, right)
```

Now, what language are these programs written in? Do you recognize it? Probably not. The reason is, I haven't used any particular programming language. These are written in what's called *pseudocode*, which just means it looks like real code, but it doesn't actually correspond to any language. This is just a convenient way to describe algorithms in a *high-level, abstract* way. The important thing is just whether it's clear exactly what the algorithm should do.

5 Analyzing Correctness

The first thing to do, when we encounter a new algorithm, is to convince ourselves that it is *correct*, which recall means that it produces the correct output for all valid inputs in a finite number of steps.

These algorithms have all been written iteratively — that is, using loops. When analyzing algorithms with loops, the trick we will usually use is called a *loop invariant*. This is a property which is preserved at every iteration through the loop. There are always three steps to showing a loop invariant:

1. **Initialization**: The invariant is true at the beginning of the first time through the loop.
2. **Maintenance**: If the invariant is true at the beginning of one iteration, it's also true at the beginning of the next iteration.
3. **Termination**: After the loop exits, the invariant PLUS the loop termination condition tells us something useful.

Let's start with linearSearch, since it's the easiest. The first loop invariant we might think of is "$A[i] \leq x$". But wait! That's not necessarily going to be true if x is not actually in the array A. So we'll use the slightly more sophisticated invariant "If A contains x, then $A[i] \leq x$".

This is true at the beginning because the array is sorted, so $A[0]$ is the smallest thing in it (*initialization*, step 1). If the condition is true at the beginning of one iteration, then it must be true at the beginning of the next iteration. This is because, since there was another iteration, it must have been true that $A[i]$ was strictly less than x, so increasing i by one must still have $A[i] \leq x$ (*maintenance*, step 2).

Now what about termination? First, if x is not in A, then the condition of the **if** statement can't possibly be true, so NOT_FOUND will be returned, like we want. If x is in A, then we know two things at the end of the loop. From the loop invariant, $A[i] \leq x$, and since the loop actually finished, the loop condition must be false and $A[i] \geq x$. Therefore $A[i] = x$, and i is returned, as required.

Wow! That was a lot of work just to show the correctness of such a simple algorithm! Don't worry, we won't have to be so pedantic all the time. But this kind of careful reasoning is exactly what we'll rely on this class, to solve little and big problems, and to be *confident* in the accuracy of our conclusions.

Following the same steps, proving the correctness of the other two algorithms is also straightforward. Can you think of good loop invariants? Here's what I'd pick for the loops in the other two algorithms.

- binarySearch: If A contains x, then $A[left] \leq x \leq A[right]$.
- gallopSearch: If A contains x, then $A[\text{floor}(i/2)] \leq x$.

I'll leave it to you to carry out the three steps (initialization, mainentance, termination) for these loop invariants, and show that all the algorithms are correct.

6 Implementation

What does it take to implement these algorithms? Notice that the "code" above is not actual code in any particular language. We have at least the following choices in turning the high-level descriptions above into actual programs:

- What programming language to use
- What precise language constructs to use (For example, should the list be an array or a linked list? Should we actually call the "length" function on the list every time, or save it in a variable?)
- What compiler to use, and what compiler options to compile with.
- What machine/architecture to run on

All of these choices will result in slightly different programs. Nevertheless, in an *attempt* to make a fair comparison, all three algorithms have been programmed in C++ (searches.cc).

So now we just have to run these programs to find out which one is best, right? Well, before we can do that, we have to at least define what we mean by "best", and how we might measure it. Fastest (time)? Smallest (space)? Fewest cache misses? Most elegant??? For now, and in most cases throughout the course, we'll try to find the *fastest* algorithm. But remember that this isn't the only important measurement.

OK then, here are some time mesurements (in nanoseconds of CPU time):

Input	x	Result	linear	binary	gallop
[6 7 8]	4	NOT_FOUND	5	5	7
[27 50 62 78 ... 180]	62	2	6	7	12
[3 6 23 27 ... 990]	500	NOT_FOUND	76	14	25
[7 11 14 17 ... 99997]	19	4	8	31	15
[14 17 28 58 ... 999992]	966	99	128	53	27
[0 2 2 3 ... 9998]	9999	NOT_FOUND	12108	35	59

So, which do you think is the fastest? Hard to say! Clearly there are some examples where each algorithm looks best. So let's do some analysis and try to make some sense out of our raw timing data.

The first thing we need is a **measure of difficulty**. It's not very useful to say "algorithm A took 31 nanoseconds on some example". How hard was that example? The measure of difficulty should be a numerical value, for each possible input or *instance* of the problem, that captures how hard that instance should be to solve.

The most common measure of difficulty, and the one we'll use for now, is the size of the problem. We almost always call this number n and for this problem n will be the size of the array A. This helps a lot in processing the data; now we can run a whole bunch of examples and plot the running times as a function of n.

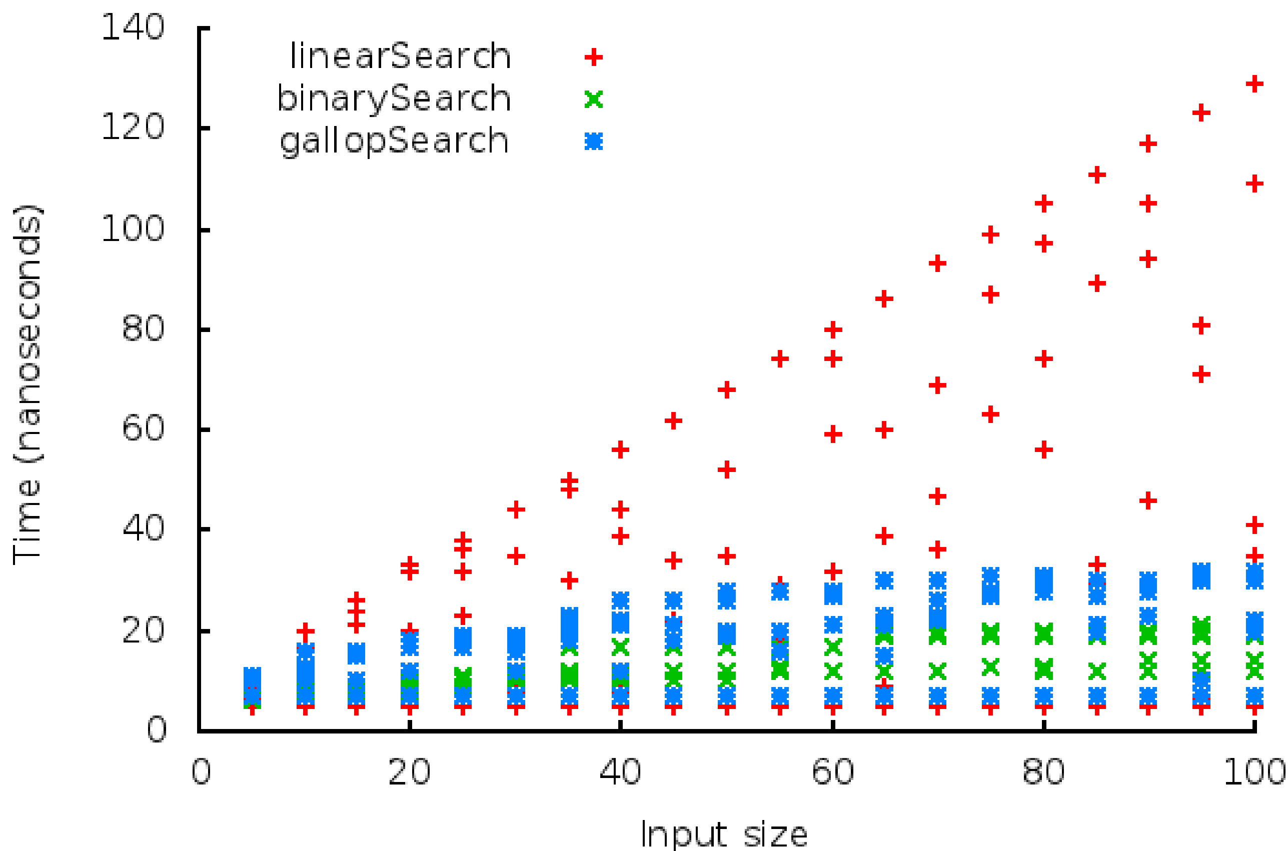

Figure 3: Search times plot

Okay, great! now we're starting to get a clearer picture of how these three implementations compare. But we'd like to simplify it even further to a single mathematical *function* for each algortihm's running time as a function of the input size. The data we have now has multiple times for each input size, because the time varies based on the particular example chosen.

There are three ways that we'll see to boil the many example timings for each input size down to a single number:

- **Best-case**: Choose the best (smallest) time for each size
- **Worst-case**: Choose the worst (largest) time for each size
- **Average-case**: Choose the average of all the timings for each size

Of these, *the worst-case time is the usually the most significant.* When we're writing algorithms and programs, we want them to perform well all the time, and especially in the most difficult examples. If the "worst-case" performance is pretty good, then that means the algorithm or program is pretty good *all the time.* Of course, we will sometimes look at best-case and average-case analysis to paint a more complete picture of an algorithm's performance.

So here's a plot of the worst-case performance of each algorithm:

So from these pictures, we can probably conclude that the binarySearch program has the best *worst-case* running time.

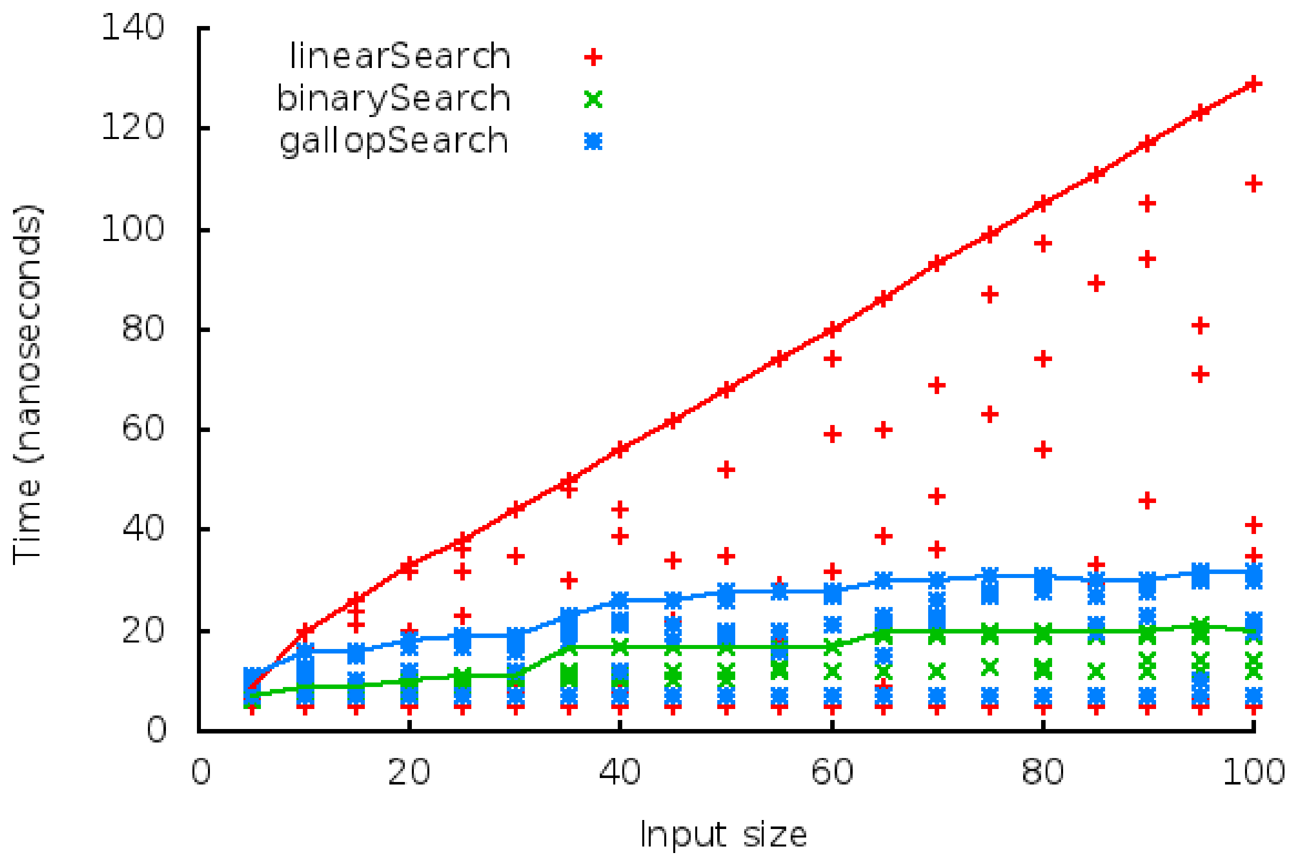

Figure 4: Worst-case of search algorithms

7 Analysis

The work we did above to compare the three implementations worked pretty well for this problem, but it has a few serious shortcomings:

- It depends on the machine. If we ran the same programs on a different machine or compiler, they might show very different results. How can we do a comparison that is *machine-independent*?
- It depends on the implementation. If you implemented these algorithms instead of me, or if you chose a different programming language, the programs would be different, and might have slightly different timings. How can we do a comparison that is *implementation-independent*?
- It depends on the examples chosen for each size. We graphed the worst-case of all the examples chosen, but how do we know that we really got the worst-case? It's impractical (sometimes impossible) to test every example for a given size, so how can we be certain to *always measure the absolute worst-case*?
- It depends on the sizes chosen. We tested the algorithms up to a few hundred elements in length. What if we come to an example where we want to search in an array of length 1,000 or even 1,000,000? How can we *measure the growth rate* as the sizes grow arbitrarily large?
- We might be able to compare algorithms a little, but can we describe *how much* better one algorithm is than another? Ideally, we'd like very simple mathematical functions like n^2 to describe the performance of algorithms we study.
- Implementing algorithms carefully and testing them on a variety of examples isn't always so cheap to do. When we get to more difficult problems and more sophisticated algorithms, this might be very costly in terms of time and money. We'd like a way to *directly analyze the algorithms themselves*, before we waste our time implementing a terrible algorithm.

The *formal analysis* of algorithms solves all of these problems in a rigorous way. But in order to do it, we'll have to make a number of simplifications. With each simplification, we get further away from the base reality of the experimental timings above, but gain a level of *abstraction* so that our analysis is more meaningful and more useful. This is the basic problem of all **mathematical modeling**, where we try to define complex phenomena (like running time) with a simple mathematical function.

8 First Simplification: Abstract Machine

The first question is, if we're not measuring the actual running time of an implementation, what are we counting? The standard approach is to define a machine (like you did all the time in Theory) and count the number of operations required for the algortihm on that machine. For this class, and in the book, the machine model is called a Random Access Machine (RAM).

But that's really an unnecessary level of detail for most of what we'll be doing in this class. Instead, we'll describe a vague notion of "primitive operation", and count the number of primitive operations in our algorithms. For our purposes, a *primitive operation* will be any command that can be accomplished in some small, *fixed* number of operations on any given modern computer architecture. Yes, this is vague, so our counts of primitive operations won't be exactly precise. More on that later...

Let's analyze the number of primitive operations in each of our algorithms above. In these cases, since our functions don't make any calls to any other functions, we can see that each line in the pseudo-code corresponds to a single primitive operation. So the operation counts are:

- linearSearch

  Lines 1, 3, and 4 are each executed exactly once every time, for 3 operations. The while loop consists of 2 operations, and the only question is how many times it runs. Based on the first loop condition, and since i is incremented by 1 every time around, there could be at most n iterations of the while loop.

But what about the second condition — will it make the loop end early? This question actually leads us to what the *worst-case input* is for this problem: when x is larger than every element in A. In this case, there really will be n iterations of the while loop, so the **total worst case cost** is $2n + 3$ operations.

- binarySearch

  The basic structure is similar to the last algorithm. We have 4 operations outside of the while loop, and each iteration through the while loop costs at most 5 operations. (It's 5 because only one of the if-else branches will get executed.)

  So once again the question is how many times the while loop will be executed *in the worst case.* This time the question is a little trickier though. We want to show how quickly progress is made towards the loop termination condition (when $left \geq right$). The trick here is to instead look at the value $right - left$. Initially, this value equals $n - 1$, and when it decreases down to 0, the loop will finish.

  So how quickly does $right - left$ decrease? Let's see what happens to it in a single iteration. There are two cases to consider (based on the if condition):

  - If the if condition is true, then we have $right$ assigned to $middle$, so the new value of $right - left$ will be $\left\lfloor \frac{left+right}{2} \right\rfloor - left \leq \frac{right-left}{2}$.
  - If the if condition is false, then $left$ gets assigned to $middle + 1$, so the new value of $right - left$ will be $right - \left\lfloor \frac{left+right}{2} \right\rfloor - 1 < right - \frac{left+right}{2} = \frac{right-left}{2}$.

  In both cases, the value of $right - left$ is reduced by at least a half. Since the intial value of $right - left$ is n, the value after k iterations will be at most $(n-1) * \frac{1}{2} * \cdots * \frac{1}{2} = \frac{n-1}{2^k}$. The loop ends as soon as this is less than 1. So we have the inequality $\frac{n-1}{2^k} < 1$, which solves to $k > \lg(n-1)$, or $k \geq \lg n$. (Note: lg just means the base-2 logarithm.)

  Combining this with what we know from above, the total worst-case cost is $4 + 5 \lg n$ operations.

- gallopSearch

  First, the easy part: lines 1, 4, and 5 are only executed once, for a total of 3 operations. But now there's *two* tricky things to worry about: the while loop, and the call to binarySearch. We'll deal with them separately.

  Let's examine the while loop first. Initially $i = 1$, and the loop terminates when $i \geq n$. As with linear search, we see that the worst case is when x is larger than every element in A, so the second condition of the loop is never true.

  So how many iterations are there? After k iterations, the value of i is $2 * 2 * \cdots * 2 = 2^k$. This gives the inequality $2^k \geq n$ as a condition for the loop's termination, which solves to $k \geq \lg n$. The total worst-case cost of the while loop is therefore $2 \lg n$ operations.

  Now what about the call to binarySearch? This is the first example we have seen of *analysis with components*, where one algorithm calls another one. This happens in almost every significant algorithm, so it's useful to be able to know how to handle it.

  In this case, we just use our analysis of binarySearch from above: the worst-case cost will be $4 + 5 \lg(right - left + 1)$ operations, since $right - left + 1$ is the size of the array that is passed into binarySearch. So what is the worst-case value of $right - left + 1$ in gallopSearch? You should be able to convince yourself that, in the worst case, the loop ends with $i = n$, and we would have $right - left + 1 = \frac{n}{2}$.

  Putting everything together, the total worst-case cost of gallopSearch is $3 + 2 \lg n + 4 + 5 \lg \frac{n}{2}$. Using a basic property of logarithms, $\lg \frac{n}{2} = \lg n - \lg 2 = \lg n - 1$. So this is $2 + 7 \lg n$.

Now after all that work we've boiled the worst-case running time of these three algortihms down to three functions: $2n + 3$, $4 + 5 \lg n$, and $2 + 7 \lg n$. But there's still two big problems:

First, these are *too exact.* Our notion of "primitive operation" is intentionally vague, so that the difference between any constant number of primitive operations is actually meaningless! Ideally, we'd like to *ignore constant factors* so that we don't get meaningless results.

Second, these functions are a huge simplification, but they still don't take us all the way towards our goal of *comparing algorithms.* Each function above will be smallest for some value or range of n. Somehow we'd like to investigate the *speed of growth* so that we can judge how the algorithms will compare for large values of n.

This leads to. . .

9 Second Simplification: Asymptotic Notation

Asymptotic notation is the final simplification we will make towards our ultimate goal of comparing the performance of algorithms. The basic idea is to *compare the growth rates* of functions.

The way this is done is with the three *relations* O, Ω, and Θ. Note that I said "relations", because that's what they are, even though we usually write them as if they're functions or sets. Yes, this is confusing, but unfortunately it's pretty standard so we have to get used to it.

9.1 Big-O Notation and Simplification Rules

We'll start with the big-O. Note that $T(n)$ is what we usually use to express the worst-case running time of a function with respect to the input size n.

> **Definition**: Big-O notation
>
> Given two functions $T(n)$ and $f(n)$, that always return positive numbers, $T(n) \in O(f(n))$ if and only if there exist constants $c, n_0 > 0$ such that, for all $n \geq n_0$, $T(n) \leq cf(n)$.

Take a few moments to digest that definition. Informally, it means that $T(n)$ does not grow any faster than $f(n)$. This is because, no matter how big n gets, $T(n)$ is always smaller than some constant multiple of $f(n)$.

Now there are a bunch of rules that we can use to make big-O simplifications. Actually, each of these is really a *theorem* based on the definition above. Could you prove them? In any case, they should become second nature to you.

> **Constant multiple rule**
>
> If $T(n) \in O(f(n))$ and $c > 0$, then $T(n) \in O(c * g(n))$.

> **Domination rule**
>
> If $T(n) \in O(f(n) + g(n))$, and $f(n) \in O(g(n))$, then $T(n) \in O(g(n))$.
>
> (In this case, we usually say that g "dominates" f.

> **Addition rule**
>
> If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then $T_1(n) + T_2(n) \in O(f(n) + g(n))$.

> **Multiplication rule**
>
> If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then $T_1(n) * T_2(n) \in O(f(n) * g(n))$.

> **Transitivity rule**
>
> If $T(n) \in O(f(n))$ and $f(n) \in O(g(n))$, then $T(n) \in O(g(n))$.

> **Trivial rules**
>
> For any positive-valued function f:
>
> - $1 \in O(f(n))$
> - $f(n) \in O(f(n))$

Using these rules, we can derive that for example $2n + 3 \in O(n)$: $3 \in O(1)$ from the constant multiple rule, and then $3 \in O(2n)$ from the first trivial rule and transitivity. So the domination rule tells us that $2n + 3 \in O(2n)$. One more application of the constant multiple rule says that the whole thing is $O(n)$.

9.2 Big-Ω and Big-Θ

If Big-O is like a $\leq$ operator, then Ω is like $\geq$ and Θ is like $=$. More formally:

> **Definition**: Big-Ω
> $T(n) \in \Omega(f(n))$ if and only if $f(n) \in O(T(n))$.

> **Definition**: Big-Θ
> $T_1(n) \in \Theta(T_2(n))$ if and only if both $T_1(n) \in O(T_2(n))$ and $T_2(n) \in O(T_1(n))$.

Conveniently, *every one of the rules above applies to these by replacing the big-O's with* Ω*s or* Θ*s*. So from the same reasoning as above we can say that $2n + 3 \in \Theta(n)$.

In all cases when analyzing algorithms, we prefer to get a big-Θ bound because that describes the function most exactly. Fortunately, this is possible with the three algorithms covered above:

- linearSearch is $\Theta(n)$ in the worst case
- binarySearch is $\Theta(\log n)$ in the worst case
- gallopSearch is $\Theta(\log n)$ in the worst case too!

What are the implications of all this? What does this mean? Well, it means we can definitely say linearSearch is asymptotically *slower* than the other two algorithms. For large enough values of n, in absolutely ANY implementation, it will eventually have a slower worst-case running time.

We also discover that the asymptotic worst-case cost of binarySearch and gallopSearch are *exactly the same*. Even though our experiments showed that binarySearch had a better worst-case cost, it was only by a constant factor, so we can't be sure (yet) which algorithm is really superior.

> **CAUTION**
> It is very common to confuse the two trios we have seen: worst/best/average case and big-O/big-Ω/big-Θ. These are not the same things! Convince yourself that we can have for example a big-Θ bound on the worst-case cost, or a big-O bound on the average cost of an algorithm. Worst/best/average case is about simplifying a cost to a single function, and asymptotic relations are about simplifying those functions further so that they can be compared.

10 A different difficulty measure

Remember the discussion of difficulty measure from above? So far we've been measuring performance in terms of n, the size of the input. And in our example of Sorted Array Search, the input size is just the length of the array A.

But sometimes we choose a different measure of difficulty to examine our algorithms from another perspective. In our running example, we might notice that there seem to be some cases where linearSearch and gallopSearch both do significantly better than binarySearch. In particular, this happens when the value of the search key x is very small relative to the values in the array A.

How can we turn this observation into a rigorous analysis? We'll consider an alternate measure of difficulty, m, defined as the least index such that $A[m] \geq x$. In particular if x is in A, then m will just be the index that the algorithm returns at the end.

Now we can go back and re-do the analysis in terms of this new measure of difficulty. We discover that the worst-case cost of linearSearch is $\Theta(m)$ and the worst-case cost of gallopSearch is $\Theta(\log m)$. Because $m \in O(n)$, these analyses actually imply the costs we derived earlier in terms of n.

The cost of binarySearch doesn't depend on m though; its cost is always $\Theta(\log n)$. This explains why, when m is very small, binarySearch doesn't perform as well as the others.

11 A different cost function

Let's break from the standard yet again. So far we have been analyzing the running time, which we measured precisely as the (imprecise) number of primitive operations.

Sometimes it's useful to measure something else. We might choose memory (the amount of space required by the algorithm) or cache misses if those seem to be bottlenecks in the performance. Alternatively, we sometimes count a very specific operation for various reasons.

We'll do the latter here and count the number of *comparisons* performed by each search algorithm. A comparison just means taking two elements from A (or one, along with the input x) and asking which one is larger. Here, it's every time we have an $=$ or $<$ or $\leq$. The exact counts of comparisons are as follows (we're back to using n as the difficulty measure):

- **linearSearch**: $n + 1$ comparisons in the worst case
- **binarySearch**: $\lg n + 1$ comparisons in the worst case

  Note: each iteration through the while loop only uses a single comparison involving *array elements*; comparisons of indices don't count here.
- **gallopSearch**: $2 \lg n$ comparisons in the worst case

Why is this interesting? Asymptotically, these are $\Theta(n)$, $\Theta(\log n)$, and $\Theta(\log n)$, just like before. So what have we learned?

Because the number of comparisons is a precise measure (unlike the number of primitive operations), this indicates that binarySearch really performs about half as many comparisons as gallopSearch in the worst case. Not coincidentally, this seems to be the closest correlation we have had with the actual experimental running time from the beginning.

12 Any Conclusions?

So which searching algorithm do you think is best? Your answer really ought to be "it depends". You should realize that there are a lot of different ways to measure the performance of algorithms, and some might be more meaningful than others depending on the problem at hand. For this problem, linearSearch is probably best if the problem size is very small, binarySearch is best if we are doing random searches, and gallopSearch will be better if we know that searches will usually be for relatively small values of x.

We've learned a lot from all our analysis, but the important thing to remember is that we don't always get clean answers like "Algorithm X is always better". In practice, we might actually use a combination of all three of the above algorithms into a so-called *hybrid* method. The best thing to do will depend on the specific problem at hand. But our analysis will help guide those decisions, and gives us good, reliable information about what the relative strengths and weaknesses of the algorithms are.

13 Summary

Good things come in threes, and *these all mean different things*:

- Design, analysis, and implementation
- Problem, algorithm, and program
- Best-case, worst-case, and average-case
- Big-O, Big-Ω, Big-Θ

Formal analysis is a methodology to understand the performance of algorithms independently of any implementation, and to compare algorithms. We have to decide *what we are measuring* (the cost function) as well as *what is the difficulty measure*.