

Mathematical Preliminaries

Logs

The log function is the inverse of an exponent. Thus, if we have

$a = b^c$, then it follows by definition that $\log_b c = a$.

Since a positive number to any exponent can never be negative, and only logs with positive bases (except for 1) are computed, it follows that you can never take the log of 0 or any negative value.

Take a look in your book on page 103 to review several rules that apply to logarithms and exponents.

$$\log_b a + \log_b c = \log_b ac$$

$$\log_b a - \log_b c = \log_b a/c$$

$$\log_b a^c = c \log_b a$$

$$\log_b a = \log_c a / \log_c b$$

$$b^{(\log_c a)} = a^{(\log_c b)}$$

$$b^a b^c = b^{a+c}$$

$$b^a / b^c = b^{a-c}$$

$$(b^a)^c = b^{ac}$$

Summations

By definition of a summation we have the following general form:

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + f(a+2) + \dots + f(b)$$

Here are a couple standard summations formulas we will use often:

$$\sum_{i=0}^n a^i = \frac{1-a^{n+1}}{1-a}$$

and

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Also, here is an example of a technique that works with some sums:

Find $1(2^0) + 2(2^1) + 3(2^2) + \dots + n(2^{n-1})$

Let $S = 1(2^0) + 2(2^1) + 3(2^2) + \dots + n(2^{n-1})$. Then

$$2S = \quad 1(2^1) + 2(2^2) + 3(2^3) + \dots + n(2^n)$$

Now, subtract the bottom equation from the top:

$$-S = 1(2^0) + (2^1) + (2^2) + \dots + (2^{n-1}) - n(2^n)$$

$$S = n(2^n) - ((2^0) + (2^1) + (2^2) + \dots + (2^{n-1}))$$

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

$$S = n(2^n) - (2^n - 1)/(2-1) = (n-1)2^n + 1$$

Proof Technique: By Example

If you have a statement of the form, "There exists a value of x which satisfies property P ," then all you need to do to prove it is find a single value of x for which the property P is satisfied. Consider this example:

There exists input such that Insertion Sort runs in $O(n)$ time for that particular input.

Consider the input $1,2,3,\dots,n$ for an input array of size n . Tracing through the code for insertion sort shows that the inner for loop in each instance runs only once. Thus, the algorithm runs in $O(n)$ for the input listed above.

Similarly, if you ever have a statement of the form, "For all values of x , the property P is satisfied," then all you would need to do to DISPROVE the statement is find a single value of x for which the statement is false. Consider the following example:

For all possible inputs, Merge Sort runs faster than Insertion Sort.

This statement is false. Once again, consider already sorted input. Merge Sort runs in $\theta(n \lg n)$ regardless of the input, but as we mentioned above, Insertion Sort on the already sorted input is faster, $O(n)$.

Proof Technique: Proof by Contradiction

If you are asked to prove an if then statement, one method is to prove the contrapositive of this statement. The method in which to do this is simply assume the opposite of the conclusion. Under this assumption, make logical deductions. Eventually, one of these logical deductions should contradict some type of well-known information, or one of the premises. Since this is impossible, the conclusion is that an incorrect step must have been taken in the proof. The only possible incorrect step was making the initial assumption. Thus, this assumption is faulty (so the opposite of it is true...) and that proves the assertion. Here's an example:

Prove the correctness of the binary search algorithm shown in class last time:

Clearly the algorithm never reports a false positive. It can ONLY say val is found if a particular array element equals val. Thus, we must show that it is impossible for the algorithm to NOT find a value that is actually in the sorted array.

Assume to the contrary that there exists an sorted array input A and a value val such that the algorithm does not find val in the array when it really does exist in the array. Let val be in array location x in the array. In order for the algorithm to not find val, the value of mid must never equal x during the running of the algorithm. In order for this to occur, at some point in the algorithm, low would have to be set to x+1 or more, OR high would have to be set to x-1 or less. In order for either of these things to occur, the appropriate if statement choices must be met. If low is set to x+1 or more, that means that a comparison was made with val and A[x+1] or some higher index, and val was greater than that value. But since we

know that $A[x] = \text{val}$, this contradicts the fact that the array was sorted. Similarly, if high is ever set to $x-1$ or less, we once again get a contradiction to the fact that the array is sorted.

Proof Technique: Induction

Based on past experience with this class, I've found that I don't have enough time to teach induction effectively and still cover all the algorithm and data structure material necessary for this course. But, in order for me to illustrate my example for loop invariants, I need induction. Plus, the concept of induction is a very important one. So, I will give you a very short synopsis of induction, so you can see the main idea and understand my loop invariant example.

Mathematical induction can be used to prove a particular statement about all positive integers. The basic idea of the proof method is as follows:

Consider trying to prove some statement to be true for all positive integers n . To do this, you must do the following three steps:

- 1) Show that the statement is true for $n=1$.**
- 2) Assume the statement is true for an arbitrary value of $n=k$.**
- 3) Prove the statement is true for $n=k+1$.**

The idea is that if you can always show that if a statement is true for one integer it is always true for the next one, and the statement is true for 1, then it must be true for 2. But if it's true for 2, it must also be true for 3, then 4, etc. Repeating the application of the rule (if the statement is true for k it is true for $k+1$) as many times as necessary proves the statement for any positive integer desired.

Proof Technique: Loop Invariant

If you want to prove some statement **S** about a loop, sometimes you can do so through proving that a loop invariant is true. Consider the following code:

```
int max = numbers[0];
for (int i=1; i<numbers.length; i++)
    if (numbers[i] > max)
        max = numbers[i];
```

Supposed we wanted to prove that **max** stored the largest value in the array **numbers** after this code segment was executed.

We can prove this through the use of a loop invariant. A loop invariant is a statement that is **ALWAYS** true about a loop at a particular point in its execution for **EACH** iteration. Consider the following loop invariant for the code above:

At end of a loop iteration (right before **i++** is executed), **max** stores the largest value from the set {**numbers[0]**, **numbers[1]**, ... **numbers[i]**}

Now, let's prove this loop invariant through induction on **i**:

Base case i=0: Before the loop begins, **max** stores **numbers[0]**, which is the maximum value from the set {**numbers[0]**}.

Assume for an arbitrary value **k**, of **i < numbers.length** that after the **k** th loop iteration, **max** stores the largest value from the set {**numbers[0]**, **numbers[1]**, ... , **numbers[k]**}

Now, we must prove, for $i=k+1$ that after the $k+1$ loop iteration, max stores the largest value from the set $\{\text{numbers}[0], \text{numbers}[1], \dots, \text{numbers}[k+1]\}$

Consider running the $k+1$ loop iteration. First, i gets incremented. Thus, now, $i=k+1$. Next, $\text{numbers}[k+1]$ is compared to max . If max is larger, we know that there is a value in the set $\{\text{numbers}[0], \dots, \text{numbers}[k]\}$ that is larger than $\text{numbers}[k+1]$ based on the inductive hypothesis. If this is the case, it necessarily follows that $\text{max}(\text{numbers}[0], \dots, \text{numbers}[k]) = \text{max}(\text{numbers}[0], \dots, \text{numbers}[k+1])$. Thus, when the if statement does not execute, max does correctly store the maximum value from the set $\{\text{numbers}[0], \dots, \text{numbers}[k+1]\}$.

The other possibility is that $\text{numbers}[k+1] > \text{max}$. By the inductive hypothesis, it follows that $\text{numbers}[k+1]$ is greater than each value in the set $\{\text{numbers}[0], \dots, \text{numbers}[k]\}$. Thus, it follows that $\text{max}(\text{numbers}[0], \dots, \text{numbers}[k+1]) = \text{numbers}[k+1]$. This is EXACTLY what max will get set to in the if statement. Thus, we can conclude in all cases, that max will store the maximum value in the set $\{\text{numbers}[0], \dots, \text{numbers}[k+1]\}$ after the $k+1$ loop iteration.

A Sample Algorithm

Sorted List Matching Problem – given two sorted lists of names, output the names common to both lists.

Perhaps the standard way to attack this problem is the following:

For each name on list #1, do the following:

- a) Search for the current name in list #2.
- b) If the name is found, output it.

If a list is unsorted, steps a and b may take $O(n)$ time. Can you tell me why?

BUT, we know that both lists are already sorted. Thus we can use a binary search in step a. From CS1, we learned that this takes $O(\log n)$ time, where n is the total number of names in the list. For the moment, if we assume that both lists are of equal size, then we can safely say that the size of list #2 is about $\frac{1}{2}$ the total input size, so technically, our search would take $O(\log n/2)$ time, where n is the **TOTAL SIZE** of our input to the problem. Using our log rules however, we find that $\log_2 n = (\log_2 n/2) + 1$. Thus, it's fairly safe to assume for large n that our running time is simply $O(\log_2 n)$.

Now, that is simply the running time for 1 loop iteration. But how many loop iterations are there? (Assume that there are $n/2$ names on each list, again, where n is the **TOTAL SIZE** of the input.) Under our assumption, there will be $n/2$ loop iterations, so our total running time would be $O(n \log_2 n)$. Why did I not divide the expression in the Big-O by 2?

A natural question becomes: Can we do better? The answer is yes. What is one piece of information we have that our first algorithm does NOT assume?

That list #1 is sorted. You'll notice that our previous algorithm will work regardless of the order of the names in list #1. But, we KNOW that this list is sorted also. Can we exploit this fact so that we don't have to do a full binary search for each name?

Consider how you'd probably do this task in real life...

| <u>List #1</u> | <u>List #2</u> |
|----------------|----------------|
| Adams | Boston |
| Bell | Davis |
| Davis | Duncan |
| Harding | Francis |
| Jenkins | Gamble |
| Lincoln | Harding |
| Simpson | Mason |
| Zoeller | Simpson |

You'd read that Adams and Boston are the first names on the list. Immediately you'd know that Adams wasn't a match, and neither would any name on the list #1 alphabetically before Boston. So, you'd read Bell and go on to Davis. At this point you'd deduce that Boston wasn't on the list either, so you'd read the next name on list #2 – voila!!! A match! You'd out put this name and simply repeat the same idea. In particular, what we see here is that you ONLY go forward on your list of names. And for every “step” so to speak, you will read a new name off one of the two lists. Here is a more formalized version of the algorithm:

- 1) Start two “markers”, one for each list, at the beginning of both lists.**

2) Repeat the following steps until one marker has reached the end of its list.

a) Compare the two names that the markers are pointing at.

b) If they are equal, output the name and advance BOTH markers one spot.

If they are NOT equal, simply advance the marker pointing to the name that comes earlier alphabetically one spot.

There are some slight mistakes in this algorithm, but the overall idea is here and you will “fix” the mistakes when you write a program to implement this algorithm for your homework.

Now, before we finish the lecture, the last thing we must do is analyze the running time of this algorithm. Here is what we see:

For each loop iteration, we advance at least one marker.

The maximum number of iterations then, would be the total number of names on both lists, which is n , using our previous interpretation.

For each iteration, we are doing a constant amount of work. (Essentially a comparison, and sometimes outputting a single name...)

Thus, our algorithm runs in $O(n)$ time – an improvement over our previous algorithm.

A final question one must ask is, can we solve this question in even less time? If yes, what is such an algorithm, if no, how can we prove it?

Our proof goes along these lines: In order to have an accurate list, we must read every name on one of the two lists. If we skip names on BOTH lists, we can NOT deduce whether we would have matches between those names or not. In order to simply “read” all the names on one list, we would take $O(n/2)$ time. But, in order notation, this is still $O(n)$, the running time of our second algorithm. Thus, we know we can not do better in terms of time, (within a constant factor), of our second algorithm.