

Foundations

Michael Barz

October 2, 2017

Note: \log (or \lg) denotes the binary logarithm (\log_2), not \log_{10} or \log_e .

1 Algorithms

This is an interesting read, but nothing really noteworthy.

2 Getting Started

2.1 Insertion Sort

Insertion sort will be our first algorithm, and it will solve the sorting problem:

Sorting Problem

Input: A sequence of n numbers (a_1, a_2, \dots, a_n) .

Output: A permutation (a'_1, \dots, a'_n) of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

The numbers sorted are called *keys*.

Insertion sort:

```
1 for j=2 to A.length
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1..j-1]
4     i = j-1
5     while i > 0 and A[i] > key
6         A[i+1] = A[i]
7         i = i-1
8     A[i+1] = key
```

Insertion sort works by sorting subsections of the list.

First, it sorts the first 2 elements, then the first 3, ...

It does this by taking element j of the list. Then, it compares it $A[j-1], A[j-2], \dots$ until it finds an element smaller than $A[j]$. Once it does that,

it just puts $A[j]$ in that element's spot. While it's doing this, if an element is larger than $A[j]$, it moves it up one in the list.

Loop Invariant

A **loop invariant** is something which is true at the start of every iteration of a loop. Formally, it satisfies two properties:

1. **Initialization:** It is true at the start of the loop's first iteration.
2. **Maintenance:** If it is true at the start of an iteration of the loop, it is true at the start of the next iteration.

Informally, it also has a third property: **termination**. This means that, when the loop terminates, the invariant gives us a useful property which helps prove the algorithm correct.

2.1.1 Exercises

Exercise 2.1-3. Consider the **searching problem**:

Searching Problem

Input: A sequence of n numbers $A = (a_1, \dots, a_n)$ and a value v .

Output: The smallest index i such that $v = A[i]$ or the special value -1 if v does not appear in A .

Write pseudocode for *linear search*, which scans through the sequence, looking for v . Using a loop invariant, prove that your algorithm is correct.

Solution. I claim the algorithm

```
1 for i=1 to A.length
2     if A[i] == v
3         return i
4 return -1
```

Now, we will prove it works by using the following loop invariant: At the start of each iteration, the sub-sequence (a_1, \dots, a_{i-1}) will not contain v .

To prove that the suggested loop invariant holds, note that at the start of the loop's iteration, the sub-sequence described will be the empty sub-sequence, which trivially does not contain v .

To prove the claimed invariant has maintenance, note that, if it is true at the start of an iteration, then (a_1, \dots, a_{i-1}) will not contain v . If a_i is v , then the algorithm will terminate. However, if the loop runs again, then $a_i \neq v$, and thus $(a_1, \dots, a_{i-1}, a_i)$ does not contain v .

Thus, the loop will either terminate the algorithm by returning the index, or the loop will terminate. If the loop terminates, we will have that (a_1, \dots, a_n)

will not contain v (by the invariant, if it got to the iteration where $i = n$, then (a_1, \dots, a_{n-1}) will not contain v , and if the loop went through $i = n$ without returning anything, then we don't have $a_n = v$), and thus we return -1 , as desired.

If the loop does terminate, then it returns the smallest i such that $a_i = v$. To see this, assume that the loop halts at iteration i and returns i . Then, (a_1, \dots, a_{i-1}) will not contain v by the loop invariant, and thus there will be no smaller index k such that $a_k = v$.

Exercise 2.1-4. Consider the binary addition problem:

Binary Addition Problem

Input: Two n -element sequences A and B . Each element of A and B will be either 0 or 1, and A and B will represent two binary integers.

Output: The sum of the two integers that A and B represent, represented similarly as an $n + 1$ -element sequence C .

Find (with proof) an algorithm which solves this problem.

Solution. Binary addition is fairly easy, and the only big issue is carrying. In pseudocode, I'm going to assume that C starts out as a sequence of $n + 1$ zeroes, as I don't know how I initialize objects with this language.

```

1 for i=1 to A.length
2     C[i] = C[i] + A[i] + B[i]
3     if C[i] == 2
4         C[i+1] = 1
5         C[i] = 0
6     if C[i] == 3
7         C[i+1] = 1
8         C[i] = 1
9 return C

```

Note: Numbers are backwards—that is, 110 would be $[0, 1, 1]$, not $[1, 1, 0]$. It's easier for me to implement and prove it works using that, and the only difference is to humans (which you can change by reversing the sequence before outputting/after inputting).

Now, a proof that it works.

Let A_k denote the number represented by the first k digits of A . Similarly define B_k and C_k .

First, we establish a loop invariant: Before the i^{th} iteration of this loop, C represents the sum of the numbers represented by the first $i - 1$ digits of A and B .

For initialization, note that before the first iteration, the first 0 digits of A and B aren't anything, and thus their sum is 0—the empty sum—and C represents 0.

Now, proving maintenance. If the property holds on iteration i , then we will prove it holds on iteration $i + 1$.

At the start of iteration i , we have that $C_i = A_{i-1} + B_{i-1}$. Also note that

$$\begin{aligned} A_i &= A_{i-1} + 2^{i-1}A[i], \\ B_i &= B_{i-1} + 2^{i-1}B[i]. \end{aligned}$$

Thus,

$$\begin{aligned} A_i + B_i &= A_{i-1} + B_{i-1} + 2^{i-1}(A[i] + B[i]) \\ &= C_i + 2^{i-1}(A[i] + B[i]) \\ &= C_{i-1} + 2^{i-1}(A[i] + B[i] + C[i]). \end{aligned}$$

Now, line 2 will set $C[i]$ to $A[i] + B[i] + C[i]$.

If $C[i]$ is 0 or 1, no worries. Then, we just leave $C[i + 1]$ as 0, and then, as

$$A_i + B_i = C_{i-1} + 2^{i-1}C[i],$$

the value of $C[i]$ is exactly what the i^{th} digit of the sum $A_i + B_i$ should be. Thus, C_{i+1} equals $A_i + B_i$, as desired.

If $C[i] = 2$, then there's a carry—we have

$$A_i + B_i = C_{i-1} + 2^i,$$

and thus need to place 10 in front of C_{i-1} 's binary representation (equivalent to adding 0 and then 1 to our representation) to get $A_i + B_i$. This is accomplished by the if statement on line 3.

Similarly, if $C[i] = 3$, we need to place 11 in front of C_{i-1} , which we do with the if statement on line 6.

As $C[i]$ cannot exceed 3, we've covered every case. Thus, maintenance holds.

Anyways, using our loop invariant, at the end of the loop we have that C contains the sum of $A_n + B_n$, which is what we want.

2.2 Algorithm Analysis

Let's look at the insertion sort from before. Except this time, let's analyze how long it takes to run.

By 'cost,' we mean the length of time it takes to execute that line; by 'times,' we mean how many times that line will execute.

There is a while loop in this code—we will denote by t_j the number of times that while loop executes for the given j value. For example, if it executes once for $j = 2$, then we'd say $t_2 = 1$. (This notation is a little different than the books—specifically, they define t_j to be one more than this, but I prefer it.)

We will also denote by c_i the amount of time it takes for line i to execute.

Now, let's look back at that insertion sort:

```

1 for j=2 to A.length
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1..j-1]
4     i = j-1
5     while i > 0 and A[i] > key
6         A[i+1] = A[i]
7         i = i-1
8     A[i+1] = key

```

Line 1 will run n times—it will run for $2, 3, \dots, n$, and then run for $n+1$, at which point it will stop.

Lines 2 through 4 will all run $n-1$ times each—once for each iteration of the loop. Line 8 will also run $n-1$ times.

Lines 5 through 7 are a little trickier. Line 5 will execute $\sum_{j=2}^n (t_j + 1)$ times, and lines 6 and 7 will each run $\sum_{j=2}^n t_j$ times.

Now, the total time cost, $T(n)$, of our algorithm is:

$$T(n) = nc_1 + (n-1)(c_2 + c_4 + c_8) + \left(\sum_{j=2}^n (t_j + 1)\right)c_6 + \left(\sum_{j=2}^n t_j\right)(c_5 + c_7).$$

Now, the best possible scenario for insertion sort happens when the list given to us is already sorted. Here, we have $t_j = 0$ for all relevant j , and thus the running time becomes

$$\begin{aligned}
 T(n) &= nc_1 + (n-1)(c_2 + c_4 + c_8) + \left(\sum_{j=2}^n (1)\right)c_6 + \left(\sum_{j=2}^n 0\right)(c_5 + c_7) \\
 &= n(c_1 + c_2 + c_4 + c_8 + c_6) - (c_2 + c_4 + c_8 + c_6) \\
 &= an + b,
 \end{aligned}$$

for some constants a and b . Thus, in the best case, $T(n)$ grows linearly.

In the worst case, when the array is in reverse order, we have $t_j = j-1$, as we need to put the j^{th} element at the beginning of the list, and we look one at a time. Thus, we have

$$\begin{aligned}
 T(n) &= n(c_1 + c_2 + c_4 + c_8) - (c_2 + c_4 + c_8) + (n(n+1)/2 - 1)c_6 + (n(n-1)/2)(c_5 + c_7) \\
 &= an^2 + bn + c,
 \end{aligned}$$

for some constants a, b, c . Thus, in the worst case, $T(n)$ grows quadratically.

2.2.1 Exercises

Exercise 2.2-2. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element and exchange it with $A[2]$, and so on. Write

pseudocode for this algorithm, known as a *selection sort*. Give the best and worst case running times (using Θ notation).

Solution. First, we describe it's implementation (given some sequence A):

```

1 for i=1 to A.length-1
2     smallest = i
3     for j=i+1 to A.length
4         if A[j] < A[smallest]
5             smallest = j
6     if smallest != i // don't switch unless we need to
7         temp = A[smallest]
8         A[smallest] = A[i]
9         A[i] = temp

```

Note that, with this implementation, the best and worst case running times are both $\Theta(n^2)$. This is because the only place where we can save time is skipping stuff in that if statement, which can only save linear time, whereas the inner for loop has a quadratic running time.

Exercise 2.2-3. What are the average, best, and worse case running times of the linear search from earlier?

Solution. The best case, when $A[1] = v$, gives us a constant running time. The worst case, when $A[n] = v$, gives us linear running time.

If the item occurs once in the sequence, then we expect it to be at position

$$\frac{1 + 2 + \cdots + n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2},$$

and thus the running time is linear.

2.3 Merge Sort

Merge sort is a divide and conquer solution to the sorting problem. It has three steps:

1. Divide the n -element sequence into two $n/2$ -element sequence.
2. Sort the sub-sequences using merge sort.
3. Combine the sorted sub-sequences to make a sorted sequences.

We'll define `merge(A, p, q, r)` to take a sequence of numbers A such that $A[p..q]$ and $A[q+1..r]$ are sorted, and then merges them to form a single sorted sub-sequence, which will be stored in $A[p..r]$. Our algorithm for `merge` takes $\Theta(n)$ time, where $n = r - p + 1$ is the size of the sub-sequence.

```

1 n_1 = q-p+1
2 n_2 = r-q
3 let L[1..n_1 + 1] and R[1..n_2+1] be new arrays

```

```

4
5 for i=1 to n_1
6     L[i] = A[i+p-1]
7 for i=1 to n_2
8     R[i] = A[i+q]
9 L[n_1+1] = \infty
10 R[n_2+1] = \infty
11
12 i = 1
13 j = 1
14 for k = p to r
15     if L[i] \leq R[j]
16         A[k] = L[i]
17         i = i+1
18     else
19         A[k] = R[j]
20         j = j+1

```

The proof that this works is kinda boring, so I'll skip it.

Now, merge sort!

`merge-sort(A, p, r)` accepts a sequence A , an two indices p and r , and sorts $A[p..r]$.

```

1 if p < r
2     q = [(p+r)/2]
3     merge-sort(A, p, q)
4     merge-sort(A, q+1, r)
5     merge(A, p, q, r)

```

To sort the entire sequence A , we call `merge-sort(A, 1, A.length)`.

Also, $\lfloor x \rfloor$ denotes the floor of x .

2.4 Running Time of Merge Sort

Let's compute $T(n)$, the worst case running time of merge sort on a set of n numbers. Merge sort on $n = 1$ takes constant time, otherwise it takes $2T(n/2)$ time to sort the two subsequences, and then some linear amount of time to merge. Thus,

$$T(n) = \begin{cases} c & n = 1 \\ 2T(\lfloor n/2 \rfloor) + \Theta(n) & n > 1 \end{cases}$$

Rigorously solving this requires actually defining Θ notation, but intuitively it's $\Theta(n \lg n)$.

2.4.1 Exercises

Exercise 2.3-2. Rewrite the `merge` procedure so that it doesn't use infinity.

Solution. See `merge_sort.py`, the python implementation of the algorithm.

Exercise 2.3-3. Use mathematical induction to show that when n is an exact power of 2, the solution to the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Solution. If $n = 2$, then $T(n) = 2 = 2 \lg 2$ trivially. Now, the inductive step.

Assume that $T(2^k) = k2^k$. Then,

$$T(2^{k+1}) = 2T(2^k) + 2^{k+1} = 2k2^k + 2(2^k) = 2^k(2k + 2) = 2^{k+1}(k + 1),$$

as desired.

Exercise 2.3-4. Recursively, we can define insertion sort to sort $A[1..n]$ by sorting $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this algorithm.

Solution. Let $T(n)$ be the running time. Then,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$

where c is some constant.

Exercise 2.3-5. Consider the searching problem. If A is sorted, we can use the binary search algorithm, which checks the midpoint and then halves. Implement it, prove it works, then argue that the worst case running time is $\Theta(\lg n)$.

Solution. I implement the `binary_search(A, v, n, m)` function. A is a sequence, v is the value, and n, m are the start and end of the search interval. To search, call `binary_search(A, v, 1, A.length)`.

```

1  if n > m
2      return -1
3  else if A[n] == v
4      return n
5  else if n == m
6      return -1
7
8  x = ceil((n+m) / 2)
9
10 if A[x] == v
11     return x

```



```

12 else if A[x] > v
13     return binary_search(A, v, n, x-1)
14 else
15     return binary_search(A, v, x+1, m)

```

Now, we will prove it works. To prove it works, note that if it is called, then either v is nowhere in the list, or it is in $A[n..m]$. This can be proven by induction easily. For the base case, when we call it on $n = 1$ and $m = A.length - 1$, it's trivially true.

Now, assume it was called on n and m . Then, we check the midpoint of the interval. If we don't find it by checking the midpoint x , then it's either on $A[n..x-1]$ or $A[x+1..m]$ or not in A . If it's on $A[x] > v$, then it must be on $A[n..x-1]$ if it's in A by our list. Otherwise, it must be on $A[x+1..m]$ if it's on our list. Thus, at the start of each method call (assuming we started with $n = 1$ and $m = A.length - 1$) the invariant remains true.

Now, the size of the interval $A[n..m]$ will continually decrease. Eventually, if we have not found v earlier, we will get to an interval containing only one point. In this case, we explicitly check if $A[n] = v$, and proceed accordingly.

Anyways, now that we know it works, let's find the time complexity.

The method takes some constant time, say c , to run on a single iteration. Say the first 6 lines take c' time to run, where c' is constant. Thus, we have

$$T(n) = \begin{cases} c' & \text{if } n=1 \\ T(n/2) + c & \end{cases}$$

which is $\Theta(\lg n)$, as it will take $\lg n$ iterations to go stop this recursion, and other than that we only add constants.

Exercise 2.3-7. Describe a $\Theta(n \lg n)$ algorithm that, given a set S of n integers and another integer x , determines whether or not there exist two elements in S whose sum is exactly x .

Solution. We can do this. Consider the algorithm `sum(A, x)`

1. Merge sort the list.
2. Iterate over the first $n-1$ elements of S , and binary search to see if $x-i \in S$ for each $i \in S$. Use binary search for this.

Merge sort is $\Theta(n \lg n)$ and binary search is $\Theta(\lg n)$. As the binary search will be repeated n times, the overall time complexity is $\Theta(n \lg n)$ for the second step, and thus the total algorithm has a time complexity of $\Theta(n \lg n)$.