FIT2004: Supplementary notes for lecture 2

Analysis of Algorithms: Part I

DANIEL ANDERSON [1]

When we write programs, we usually like to test them to give us confidence that they are correct. But testing a program can only ever demonstrate that it is wrong (if we find a case that breaks it). To prove that a program always works, we need to be more mathematical. The two major focuses of algorithm analysis are proving that an algorithm is **correct**, and determining the amount of **resources** (time and space) used by the algorithm.

Summary: Analysis of Algorithms : Part I

In this lecture, we cover:

- What it means to analyse an algorithm
- Correctness and termination of an algorithm
- How to prove rigorously that an algorithm is correct
- Analysing the time complexity of an algorithm
- All of the above demonstrated on binary search

Recommended Resources: Analysis of Algorithms : Part I

- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Math/`
- `http://www.csse.monash.edu.au/courseware/cse2304/2006/03logic.shtml`
- CLRS, Introduction to Algorithms, Pages 17 - 19, Section 2.1

Program Verification

Program verification or correctness hinges on two main principles. We must on one hand prove that our algorithm, if it terminates always produces the correct result. On the other, we must prove that it does indeed always terminate.

We will explore these issues by analysing one of the most well-known fundamental search algorithms, binary search.

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

Algorithm: Binary Search

```
1:  function BINARY_SEARCH(array[1..N], key)
2:      ▷ Precondition: array must be sorted
3:      Set lo = 1 and hi = N + 1
4:
5:      while lo < hi - 1 do
6:          Set mid = floor((lo + hi) / 2)
7:          if key ≥ array[mid] then
8:              lo = mid
9:          else
10:             hi = mid
11:         end if
12:     end while
13:
14:     if array[lo] = key then
15:         key found in position lo
16:     else
17:         key not found
18:     end if
19: end function
```

Programmers, even experienced ones frequently write incorrect binary searches. Commonly they are "off-by-one" errors in which the final index reached by the search is one away from where it should have been. What if line 7 had read `key > array[mid]`? This would be a mistake, and not an easy one to find given how small of a difference it is. So how then do we reason that this algorithm is in fact the correct one?

We reason the correctness by establishing an **invariant** in the algorithm. The "key" invariant for binary search is the following:

Invariant: Binary Search

If `key` $\geq$ `array[1]`, then at each iteration:

1. `array[lo]` $\leq$ `key`
2. `array[hi]` $>$ `key`. (Assume that `array[N+1]` $= \infty$)

Taken together, these two conditions imply that `array[lo]` $\leq$ `key` $<$ `array[hi]`, combined with the sortedness of `array`, we deduce that `key` (if it exists) lies within the range `[lo..hi)`.

We can now re-interpret the binary search algorithm such that every action it takes is simply aiming to maintain these invariants. This observation can lead us to prove that the algorithm is indeed correct and that it terminates.

Arguing the correctness of binary search

Case 1: `key` $\geq$ `array[1]`

We must first argue that the invariants established above are correct when the algorithm begins. Once established, we subsequently argue that the invariants are maintained throughout the algorithm. When we first begin, we initialise `lo = 1, hi = N + 1`, so:

1. If `key` $\geq$ `array[1]`, since `lo = 1`, `array[lo]` $\leq$ `key`.
2. If we take `array[N+1]` $= \infty$, then since `hi = N + 1`, `array[hi]` $>$ `key`.

Therefore the invariant is true at the beginning of the binary search algorith,. As we perform iterations of the main loop of the binary search, what happens to this invariant? At each step, we compute `mid` $= \lfloor$`(lo + hi)`$/2 \rfloor$ and compare `array[mid]` to `key`. The conditional statement in the loop then enforces the invariant.

1. If `key` $\geq$ `array[mid]`, then after setting `lo = mid`, we will still have `array[lo]` $\leq$ `key`
2. If `key` $<$ `array[mid]`, then after setting `hi = mid`, we will still have `array[hi]` $>$ `key`.

Hence the invariant holds throughout the iterations of the loop. Therefore, if the loop terminates, it is true that `array[lo]` $\leq$ `key` $<$ `array[hi]`. Since the condition for the loop to terminate is that `lo` $\geq$ `hi - 1`, combined with the previous inequality, it must be true that `lo = hi - 1`. Therefore if `key` exists in `array` at all, it must exist at position `lo`, and hence the binary search algorithm correctly identifies whether or not `key` is an element of `array`.

Case 2: `key` $<$ `array[1]`

In this case since `array` is sorted, `key` can not be an element of `array`. Therefore provided that the algorithm terminates, regardless of the value of `lo`, `array[lo]` $\neq$ `key` and hence the algorithm correctly identifies that `key` is not an element of `array`.

Arguing that binary search terminates

We have argued that binary search is correct **if it terminates**, but we still need to prove that the algorithm does not simply loop forever, else it is not correct in general. Let us examine the loop condition closely. The loop condition for binary search reads `lo < hi - 1`, hence while we are still looping, this inequality holds. We now recall that

$$\texttt{mid} = \left\lfloor \frac{\texttt{lo} + \texttt{hi}}{2} \right\rfloor.$$

Theorem: Finiteness of Binary Search

If `lo < hi - 1`, then

$$\texttt{lo} < \texttt{mid} < \texttt{hi}$$

Proof

Since `mid` is the floor of (`lo + hi`)/2, it is true that

$$\frac{\texttt{lo + hi}}{2} - 1 < \texttt{mid} \leq \frac{\texttt{lo + hi}}{2}.$$

Multiplying by two, we find

$$\texttt{lo + hi} - 2 < 2 \times \texttt{mid} \leq \texttt{lo + hi}.$$

We can now substitute the inequality `lo` $<$ `hi - 1` to obtain

$$2 \times \texttt{lo} - 1 < 2 \times \texttt{mid} \leq 2 \times \texttt{hi} - 1.$$

Adding one to the inequality then yields

$$2 \times \texttt{lo} < 2 \times \texttt{mid} + 1 \leq 2 \times \texttt{hi},$$

which simplifies to

$$2 \times \texttt{lo} < 2 \times \texttt{mid} < 2 \times \texttt{hi},$$

and hence

$$\texttt{lo} < \texttt{mid} < \texttt{hi},$$

as desired.

Since this is true, and we always set either `lo` or `hi` to be equal to `mid`, it is always the case that at every iteration, the interval `[lo..hi)` decreases in size by at least one. The interval `[lo..hi)` is finite in size, therefore after some number of iterations it is true that `lo` $\geq$ `hi - 1`. Therefore the loop must exit in a finite number of iterations and hence the binary search algorithm terminates in finite time.

Complexity analysis

We have proven that the binary search algorithm terminates, and that it terminates with the correct answer. The only thing left to do is to prove that it terminates **fast**. We can express the number of operations performed

by the binary search algorithm T_N as a function of the size N of the input data. The number of operations taken by the binary search algorithm can be expressed as

$$T_N = \begin{cases} T_{\frac{N}{2}} + a, & N > 1 \\ b, & N = 1 \end{cases}, \tag{1.1}$$

where a is some constant number of operations that we perform each step, and b is some constant number of operations required at the end of the algorithm.

Theorem: Time Complexity of Binary Search

The recurrence relation (1.1) has the explicit solution

$$T_N = a\log_2(N) + b$$

Proof

We will prove that $T_N = a\log_2(N) + b$ by induction on N.

Base case:

Suppose $N = 1$, then $a\log_2(N) + b = b$ which agrees with out definition of T_N.

Inductive step:

Suppose that for some N, it is the case that $T_{\frac{N}{2}} = a\log_2\left(\frac{N}{2}\right) + b$. It is then true that

$$\begin{aligned} T_N &= T_{\frac{N}{2}} + a \\ &= a\log_2\left(\frac{N}{2}\right) + b + a \\ &= a(\log_2(N) - \log_2(2)) + b + a \\ &= a\log_2(N) - a + b + a \\ &= a\log_2(N) + b \end{aligned}$$

as desired. Hence by induction on N, it is true that

$$T_N = a\log_2(N) + b.$$

Given this, we can conclude that the time complexity of binary search is $O(\log(N))$.

Solutions to common recurrence relations

When analysing the time and space complexity of our algorithms, we will frequently encounter some very similar recurrence relationships whose solutions we should be able to recall.

Linear Complexity

The solution of the following recurrence

$$T_N = \begin{cases} T_{N-1} + a, & N > 1 \\ b, & N = 1 \end{cases},$$

is given by

$$T_N = aN + b - a,$$

and can easily be proven by induction. (Try it as an exercise.)

Exponential Complexity

For the following recurrence relationship

$$T_N = \begin{cases} 2 \times T_{N-1} + a, & N > 0 \\ b, & N = 0 \end{cases},$$

the solution is given by

$$T_N = (a + b) \times 2^N - a.$$

Try to prove this by induction. We will see some more complicated examples of recurrence relations in the future, particularly when analysing divide-and-conquer algorithms such as the recursive sorting algorithms Quicksort and Merge Sort.