COMP 251 Study guide

Francis Piche

January 11, 2018

Contents

1 Preliminaries **3**

I Recursive Algorithms **3**

2 Divide + Conquer Algorithms **3**
2.1 MergeSort 4
2.2 Binary Search 5
2.3 Run Time of Divide + Conquer in General 5
2.4 Aside on Recurrences: Domain Transformation 6

1 Preliminaries

In this course an algorithm is considered **good** if it:

- Works
- Runs in polynomial time. Meaning it runs, in $O(n^k)$ time. Where n is (always) the size of the problem. (Number of elements in a list to be sorted etc.)
- Scales multiplicatively with computational power. (If your computer is twice as fast, the problem is solved at least twice as fast)

A bad algorithm is one that:

- Doesn't always work
- Runs in exponential time or greater. Meaning: $O(k^n)$ time.
- Does not scale well with computational power. (Your computer is twice as fast, but barely any performance boost).

Part I

Recursive Algorithms

I won't be going into detail on the specifics of things like how recursion works, MergeSort, BinarySearch, solving recurrences, Big O, etc. as it's considered prerequisite material. If you need some review, my COMP250 study guide is still publicly available.

2 Divide + Conquer Algorithms

Examples:

- MergeSort
- BinarySearch

2.1 MergeSort

The MergeSort algorithm involves splitting a list of n elements in half, sorting each half recursively, and merging the sorted lists back into one. It takes time $T(\frac{n}{2})$ to sort the list of half size, and time $O(n)$ to merge the list back together. So the recurrence relation for MergeSort is given by:

$$T(n) = 2T(\frac{n}{2}) + cn$$

where c is some constant.

Theorem 1. MergeSort runs in time $O(nlog(n))$.

Proof. Add **dummy numbers** (extra "padding" to the list), until n is a power of two. $n = 2^k$. We can do this because $O()$ gives an **upper bound**, and adding numbers will make our solution take longer than the real one. Doing this will make solving the recurrence easier.

Unwinding the formula:

$$T(n) = 2(2(T(\frac{n}{4}) + c\frac{n}{2}) + cn$$

$$= 2^2(T(\frac{n}{4}) + 2cn)$$

$$= 2^3(T(\frac{n}{8}) + 3cn)$$

$$= 2^4(T(\frac{n}{16}) + 4cn)$$

Notice we have a pattern emerging.

$$= 2^k(T(1)) + kcn$$

Recall $2^k = n$, so $k = log_2(n)$ and $T(1) = 1$ so:

$$= n + nlog_2(n)$$

Which is $O(nlogn)$. □

2.2 Binary Search

Binary search involves splitting your sorted list into two, and searching that half. So our recurrence is given by:

$$T(n) = T(\frac{n}{2}) + c$$

where c represents the constant work (comparisons, setting new bounds etc.)

Theorem 2. Binary Search is $O(log_2(n))$.

Proof. Again we add dummy numbers so that n is a power of two. $n = 2^k$

We begin with our recurrence:

$$T(n) = T(\frac{n}{2}) + c$$

$$= T(\frac{n}{4}) + c + c$$

$$= T(\frac{n}{8}) + c + c + c$$

$$= T(\frac{n}{2^k}) + kc$$

$$= T(1) + log_2(n)$$

since $k = log_2(n)$ which is $O(log_2(n)$. $\square$

2.3 Run Time of Divide + Conquer in General

Divide and Conquer is a technique of solving problems that involves taking one large problem of size n , and breaking it down into a smaller problems of size $\frac{n}{b}$, and solving those problems recursively. They are then combined to produce a solution in time poly-time: $O(n^d)$.

So the run-time of a divide and conquer algorithm is:

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

In the case of MergeSort, $a = 2$, $b = 2$, $d = 1$.

In the case of BinarySearch, $a = 1$, $b = 2$, $d = 0$.

2.4 Aside on Recurrences: Domain Transformation

Note that the recurrence for MergeSort is really:

$$T'(n) \leq T'(\lfloor n/2 \rfloor) + T'(\lceil n/2 \rceil) + cn$$

Which we simplified by adding dummy entries. However, we can also say this: Note that this is an informal approximation, since it's really:

$$T'(n) \leq 2T'(\frac{n}{2} + 1) + cn$$

But the +1 doesn't fit with our previous method.

We'll use **domain transformation** to solve this, starting with:

$$T(n) = T'(n + 2)$$

$$\leq T'(\frac{n+2}{2} + 1) + c(n + 2)$$

plugging in our expression from above

$$\leq T'(\frac{n+2}{2} + 1) + c'(n)$$

absorbing the +2 into c.

$$= T'(\frac{n}{2} + 2) + c'(n)$$

simplifying the fraction.

$$= T(\frac{n}{2}) + c'n$$

from our domain transformation at the beginning. Solving this the usual way, we get:

$$T(n) = O(nlog(n))$$

But again from our domain transformation:

$$T(n) = T'(n + 2)$$

, so

$$T'(n) = T(n - 2) = O(nlog(n))$$

So we've shown that $T'(n)$ has the same upper bound as $T(n)$.