COMP 251 Study guide

Francis Piche

January 16, 2018

Contents

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 $O(nlogn)$. $\square$

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 $T(n)$.

3 Master Theorem

Theorem 3. If $T(n) = aT(n/b) + O(n^d)$ for constants $a > 0$, $b > 1$, $d \geq 0$, then:

$$\begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d log(n)) & \text{if } a = b^d \\ O(n^{log_b(a)}) & \text{if } a > b^d \end{cases}$$

These cases are just a few that occur often in practice when dealing with divide + conquer algorithms.

Proof. First we'll need two things. One is the geometric series, and the other is a law of logarithms. Professor Vetta proved them in class, and honestly I doubt you'd be asked to prove them on an exam, but it's good proof practice to go through them so I'll do it here.

$$\sum_{k=0}^{l} x^k = \frac{1 - x^{l+1}}{1 - x}$$

Proof:

Starting with:

$$(1 - x) \sum_{k=0}^{l} x^k$$

We can expand it out:

$$= \sum_{k=0}^{l} x^k - \sum_{k=0}^{l} x^{k+1}$$

Simplifying the sigma notation:

$$= \sum_{k=0}^{l} x^k - \sum_{k=1}^{l+1} x^k$$

All terms will cancel except:

$$= x^0 - x^{l+1} = 1 - x^{l+1}$$

Divide through by $1 - x$

$$= \frac{1 - x^{l+1}}{1 - x}$$

Our second fact to derive is this law of logs:

$$x^{log_b(y)} = y^{log_b(x)}$$

Using the power rule of logarithms:

$$log_b(x)log_b(y) = log_b(y^{log_b(x)})$$

similarly,

$$log_b(x)log_b(y) = log_b(x^{log_b(y)})$$

so,

$$log_b(x^{log_b(y)}) = log_b(y^{log_b(x)})$$

Now we're ready for the proof.

Assume n is a power of b, and split up the problem into all it's chunks.

$$T(n) = n^d + a(\frac{n}{b})^d + a^2(\frac{n}{b^2})^d + ... + a^l(\frac{n}{b^l})^d$$

(this is just if you'd "unwound" the whole recursion down to it's simplest form like we did in the MergeSort/Binary Search proofs.)

Each term is the amount of work it will take at each level of the recursion.

Notice you can factor out:

$$= n^d(1 + a(\frac{1}{b})^d + a^2(\frac{1}{b^2})^d + ... + a^l(\frac{1}{b^l})^d$$

$$= n^d(1 + (\frac{a}{b})^d + (\frac{a}{b^d})^2 + ... + (\frac{a}{b^d})^l$$

That looks like a geometric series! So let's look at the cases:

Case 1: $a < b^d$

Applying the geometric series formula:

$$= n^d \sum_{k=0}^{l} (\frac{a}{b^d})$$

$$= n^d \frac{1 - (\frac{a}{b^d})^{l+1}}{1 - \frac{a}{b^d}}$$

we can remove the $\frac{a}{b^d}^{l+1}$ term with this inequality:

$$\leq n^d \frac{1}{1 - \frac{a}{b^d}}$$

which is $O(n^d)$.

Case 2: $a = b^d$

Since $\frac{a}{b^d} = 1$:

$$= n^d(1 + 1 + 1 + ... + 1)$$

There are $l + 1$ terms, but we said n was a power of b, ($n = b^l$) so, $l = log_b(n)$,thus:

$$= n^d(log_b(n) + 1)$$

which is $O(n^d log_b(n))$

Case 3: $a > b^d$

Again from geometric series, and multiplying through by -1:

$$n^d \frac{(\frac{a}{b^d})^{l+1} - 1}{\frac{a}{b^d} - 1}$$

Again this inequality holds:

$$\leq n^d \frac{(\frac{a}{b^d})^{l+1}}{\frac{a}{b^d} - 1}$$

Which is $O(n^d(\frac{a}{b^d})^l)$ which we can simplify:

$$(\frac{n}{b^l})^d a^l$$

but $n = b^l$, so:

$$= (1)a^l$$
$$= a^{log_b(n)}$$

now by our second fact:

$$= n^{log_b(a)}$$

which is $O(n^{log_b(a)})$

$\square$

It's **much** more important to understand the proof than it is to memorize the theorem.

3.1 Tree Method to Prove Master Theorem

A more intuitive way to think of the proof is with a *Recursion Tree*.

The root node of the tree has label n, and each node has a children (except the leaves). a is called the *branching factor*. Each child is labelled $\frac{n}{b^d}$ where d is the depth. The labels represent the size of the sub problems.

The number of nodes at each level is a^d.

Case 1 is when the root level "dominates" all other levels, so the running time is just $O(f(n))$ where $f(n)$ is the amount of work at the root level.

Case 2 is when all levels are roughly the same weight. So the total running time is just $O(f(n)l)$ where l is the number of levels.

Case 3 is when the leaves dominate, so the running time is $O(a^l)$ since the leaves each take time $O(1)$, and there are a^l of them.

4 Multiplication

4.1 Grade School Multiplication

This takes n^2 multiplications when you multiply two n-digit numbers. so the runtime is $\Omega(n^2)$

4.2 Russian Peasant Multiplication

Super weird looking algorithm but it works!

```
Mult(x,y){
   if x = 1 then output y
   if x is odd then output y + Mult(floor(x/2),2y)
   if x is even then output Mult(x/2, 2y)
}
```

This actually comes from if you take the binary representation of x: say $x = 46_{10}$ then $x = 101110_2$. The bits that are 1's will have the y added step, and the zero bits will just have the doubling step. Weird right?

Notice that this means the number of steps is just the number of bits in x. The number of digits in the result will be at most $2n$, so if we need to then add these, we add at most n numbers of $2n$ digits so takes time $O(n^2)$

4.3 Divide + Conquer Multiplication

Notice that a number x can be written as:

$$x = x_n x_{n-1} ... x_{\frac{n}{2}+1} x \frac{n}{2} ... x_2 x_1$$

where the x_i are the digits.

Then we have:

$$x = 10^{\frac{n}{2}} x_L + x_R$$

where n is the number of digits, x_L is the first $\frac{n}{2}$ digits, and x_R is the last $\frac{n}{2}$

So by expanding:

$$xy = (10^n x_L y_R + 10^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R$$

Notice that this now involves four products of $\frac{n}{2}$ digit numbers. So the recursion is:

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

We have $a = 4$,$b = 2$,$d = 1$, which is case 3 of the master theorem.

Which means the running time is:

$$O(n^{log_2(4)})$$

which simplifies to:

$$O(n^2)$$

Thanks to Gauss, we can actually use this fact:

$$x_L y_R + x_R y_L = x_R y_R + x_L y_L - (x_R - x_L)(y_R - y_L)$$

which is actually only 3 unique products. (adding is cheap)

So our new running time is:

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

which is case 3 of the master theorem, so

$$O(n^l og_2(3))$$

$$= O(n^{1.59})$$

4.4 Fast Fourier Transforms

These are $O(nlog(n))$ for multiplying n-bit numbers. They'll be studied more in-depth at the end of the course (time-permitting).

4.5 Multiplying Matrices

There are n multiplications to calculate each entry of the result matrix, and there are n^2 entries, so $O(n^3)$

Using divide + conquer, divide into 4 sub-matrices:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ x_{d1} & x_{d2} & x_{d3} & \dots & x_{dn} \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

So if we let:

$$x = \begin{bmatrix} A & B \\ C & D \end{bmatrix} y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

then:

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

So multiplying involves eight products with $\frac{n}{2}$x$\frac{n}{2}$ and the recurrence is:

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

which is Case 3 of the master theorem, so runtime is $O(n^{log_2 8})$ which is $O(n^3)$, no improvement.

There actually is a trick to do better.

Claim:

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

is the same as:

$$\begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

where:

$$S_1 = (B - D)(G + H)$$

$$S_2 = (A + D)(E + H)$$

$$S_3 = (A - C)(E + F)$$
$$S_4 = (A + B)H$$
$$S_5 = A(F - H)$$
$$S_6 = D(G - E)$$
$$S_7 = (C + D)E$$

which is only 7 products! (The additions are negligible)

So we have:

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

Which is Case 3 of the master theorem, so $O(n^log_2(7))$which is $O(n^{2.81})$

4.6 Fast Exponentiation

Method of taking exponents in a fast way, since doing:

$$x * x * x * x ... * x$$

is super slow.

```
FastExt(x,n){
if n=1 output x
else
   if n is even output FastExp(x, floor(n/2))^2)
   if n is odd output FastExp(x, floor(n/2))^2)*x
}
```

So our recurrence looks like:

$$T(n) = T(floor(\frac{n}{2}) + O(1)$$

(since we're halving the problem, and doing some constant work at each step)

This is Case 2 of the Master Theorem, so the runtime is $O(log_2 n)$