CS 3510: Design & Analysis of Algorithms (Section C)

Exam 1 Study Guide

Abrahim Ladha

1 Recommended Problems:

Here are some recommended problems to do in the textbook:

Link To DPV

- Big-O: 0.1
- Modular Arithmetic: 1.11, 1.20, 1.24, 1.25
- Fermats Little Theorem: 1.42, 1.43, 1.44, and 1.45
- Divide-and-Conquer: 2.1, 2.4, 2.5(a - e), 2.14, 2.16, 2.17, 2.19, 2.23, 2.31(b), and 2.32

Note: Some of these problems are difficult, perhaps more difficult than the exam. They will provide good practice for the exam!

2 Topic List:

2.1 Big-O

Before delving deep into designing algorithms to solve problems, we need a way to characterize and analyze algorithms in relation to other. We focus on the *runtime* of algorithms, and so compare algorithms based on how they grow in relation to input size n. We use three operators to compare functions: $\mathcal{O}, \Omega, \Theta$. Let's review each one:

Let $f(n)$ and $g(n)$ be functions from positive integers $\mathbb{Z}^+$ to positive reals $\mathbb{R}^+$.

1. **We say $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.**
2. **We say $f = \Omega(g)$ if there is a constant $c > 0$ such that $c \cdot f(n) \geq g(n)$. In other words, $g = O(f)$.**
3. **We say $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.**

In more simple terms, $f = O(g)$ means that g grows at least as fast as f, while $f = \Theta(g)$ means that f and g grow at the same rate. It's important to note that we look at growth rates after some large constant $K > 0$, so that we are not affected by how a function behaves at small inputs (as that has high variation).

Some of the most complicated functions to classify in relative to each other involve *log* statements. There are a couple equivalencies regarding log functions that can help a lot to simplify these functions into expressions which are more easily classifiable:

$$\log_b(n^c) = c(\log_b n) \ (1)$$

$$a^{\log_a n} = n \ (2)$$

$$\log_b(xy) = \log_b x + \log_b y \ (3)$$

$$\log_b(\frac{x}{y}) = \log_b x - \log_b y \ (4)$$

$$a^{\log_b n} = n^{\log_b(a)} \ (5)$$

$$\log_b a = \frac{\log_c a}{\log_c b} \ (6)$$

Equation 5 is one of the most useful tools when dealing with logs in the exponent. For example, consider the function

$$f(x) = 32^{\log_2(x)}$$

$$g(x) = x^5$$

Upon inspection of the two functions in this form, it's not entirely clear whether f or g grows faster. However, if we apply (5), we can simplify $f(x)$ as follows:

$$f(x) = 32^{\log_2(x)} = x^{\log_2(32)} = x^5$$

Now we can see that $f(x) = \Theta(g(x))$.

> **Note:** Unless otherwise specified, you can assume that the *log* functions use base 2. However, it technically does not matter, as $\log_b(x) = \Theta(\log_c(x))$ for any valid b and c. See if you can prove this using log rules!

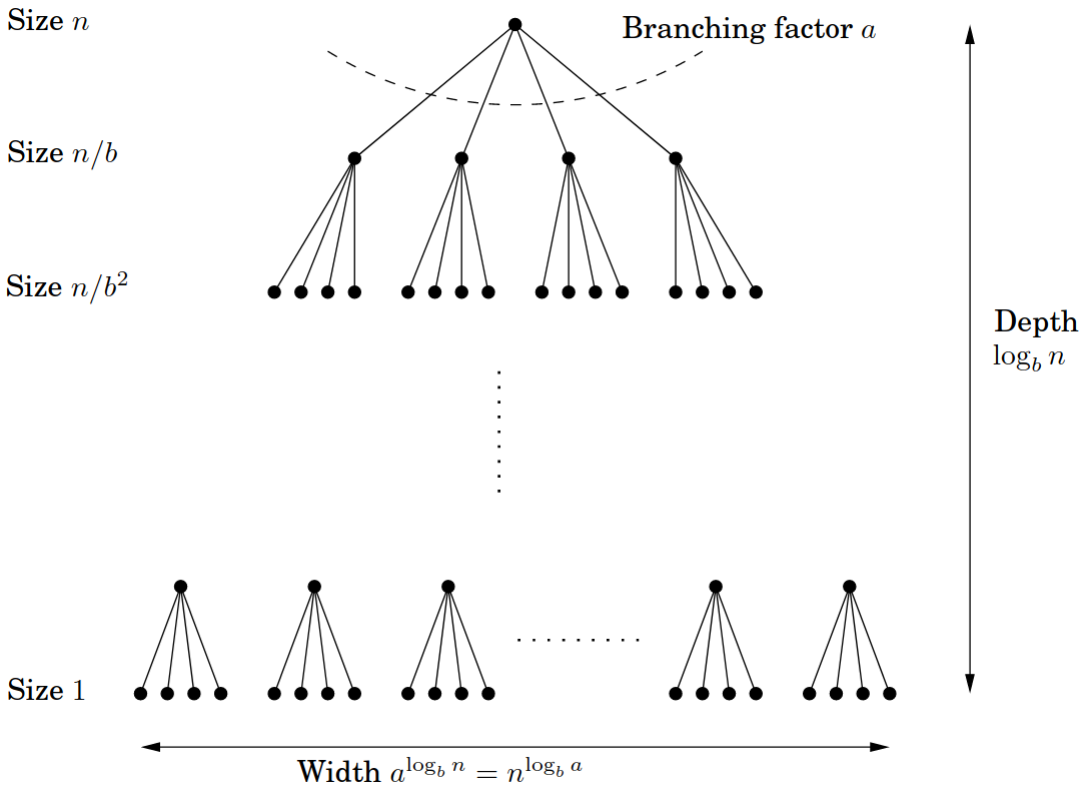
There are some classes of functions we can easily compare that you should keep in mind for the exam. We can list them in increasing order of growth rate:

$$\text{Constant} = \mathcal{O}(\text{Logarithmic}) = \mathcal{O}(\text{Polynomial}) = \mathcal{O}(\text{Exponential}) = \mathcal{O}(\text{Factorial})$$

In the homework, you encountered some functions that didn't exactly fit these classes, such as $f(n) = (\log n)^4$. In this cases, you either want to simplify them using log rules above, or if that's not possible, try to reason between which classes the function should reside. In the case of $f(n)$, it must be faster than general logarithmic functions, but must be smaller than polynomial time (think of the case where $g(n) = n^4$).

2.2 Divide-and-Conquer

In a Divide-and-Conquer algorithm, we break down the original problem into smaller, non-overlapping problems which are easier to solve. The complexity of this kind of algorithm is determined by three factors: the branching factor a, the depth $\log_b n$, and the complexity of combining answers to previous subproblems, which takes time $O(n^d)$.



As we can see in the figure above, a determines how many subproblems are created per split, and b represents the size of these subproblems with respect to the size of the original problem. We aim to analyze divide-and-conquer strategies by comparing the amount of work done to combine answers, which is n^d, to the amount of work done by recursively subdividing the problem into smaller ones, which is $n^{\log_b a}$.

Once we have the recurrence relation for a Divide-and-Conquer algorithm, we can use the Master Theorem to directly determine its Big-O runtime.

Master Theorem: Given a recurrence relation in the form

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

where $a > 0$, $b > 1$, $d \geq 0$, we obtain:

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

Now, let's review some of the canonical divide-and-conquer algorithms and analyze them using the Master Theorem!

2.2.1 Binary Search

One of the most common divide-and-conquer algorithms is **Binary Search**. To review, given as an input an array A containing values $[a_1, a_2, a_3, \ldots, a_n]$, and a target value k, we wish to find if k is in A. We define the algorithm as follows:

1. Compare k with $a_{n/2}$.
2. If $k < a_{n/2}$, recurse on the subarray $[a_1, a_2, \ldots, a_{n/2-1}]$.
3. Otherwise, recurse on the subarray $[a_{n/2}, \ldots, a_n]$.
4. Repeat until we find k, or until the subarray is empty.

In this algorithm, we divide our input of size n into one subproblem of size $n/2$ at each level. The work outside of the subproblems is just constant; we calculate $n/2$ and perform a constant number of calculations. Therefore we get the recurrence

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

We have $a = 1, b = 2, d = 0$. We have $d = \log_b a$, so the final running time is $O(n^0 \log n) = O(\log n)$.

2.2.2 Merge Sort

Now let's review **Merge Sort**. As a review, Merge Sort is an algorithm that seeks to sort an array of size n; it splits the array into two smaller, non-overlapping subarrays of size $n/2$ in a single iteration/recursive call. This continues until we have segments that consist of only a single element, at which point the algorithm can compare these individual elements in linear time, joining smaller, sorted subarrays until it produces a fully sorted array of length n. Here are the individual steps:

1. Divide input array $A[1, \ldots, n]$ into left half A_L and right half A_R.
2. Recursively call Merge Sort on A_L and A_R.
3. Merge left half and right half by checking elements index-by-index and replacing the original array A with the correct order. This is an $O(n)$ operation.

Based on the above explanation, we can derive the following recurrence relation:

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

We can identify that a (number of subproblems per split) here is 2, b (division factor of subproblems) is 2 as well, and d is 1, given that the recombination that occurs is in $\mathrm{O}(n^1) = \mathrm{O}(n)$ or linear time.

Using this information, we can reference the Master Theorem equation above and determine the time complexity. Does your answer agree with what you know about MergeSort? Try a similar analysis with another popular Divide-and-Conquer algorithm to test your understanding.

2.2.3 Multiplication

We can also use divide-and-conquer for integer multiplication. Let's say we have two n-bit integers x and y, and we wish to calculate xy. The normal iterative approach would be $O(n^2)$, as we multiply bit-by-bit. Let's define two D-and-C algorithms (second one improving the iterative approach) to do this instead.

First Try: $O(n^2)$

First, we can split x into left and right halves based on the properties of binary integers:

$$x = 2^{n/2}x_L + x_R$$

We can do the same with y:

$$y = 2^{n/2}y_L + y_R$$

Now, we can express xy as the following:

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n(x_Ly_L) + 2^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R$$

Now we have subproblems we can use: multiplying $x_Ly_L, x_Ly_R, x_Ry_L, x_Ry_R$. We now obtain the following algorithm:

```
Mult (x, y):
    n = len(x)
    if n == 1:
        return x[0] * y[0]
    # Splitting x & y into left and right halves
    x_L = x[0 : n/2]
    x_R = x[n/2 : n]
    y_L = y[0 : n/2]
    y_R = y[n/2 : n]

    # Splitting x & y into left and right halves
    A = Mult(x_L, y_L)
    B = Mult(x_L, y_R)
    C = Mult(x_R, y_L)
    D = Mult(x_R, y_R)

    return 2^n A + 2^(n/2) (B+C) + D
```

This running time of this algorithm is $T(n) = 4T(n/2) + O(n) \approx O(n^2)$ using Master's Theorem. But, we can do better.

Fast Multiplication:

To optimize this, we look at the expansion of xy:

$$2^n(x_Ly_L) + 2^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R$$

We can actually express

$$(x_Ly_R + x_Ry_L) = (x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R$$

As a result, we don't have to compute x_Ly_R or x_Ry_L! Instead, we just have to compute $(x_L + x_R)(y_L + y_R)$. This reduces the number of recursive calls at each level down to 3, rather than 4:

```
FastMult (x, y):
    n = len(x)
    if n == 1:
        return x[0] * y[0]

    x_L = x[0 : n/2]
    x_R = x[n/2 : n]
    y_L = y[0 : n/2]
    y_R = y[n/2 : n]

    A = FastMult(x_L, y_L)
    B = FastMult(x_L, y_R)
    C = FastMult(x_L + x_R, y_L + y_R)

    return 2^n A + 2^(n/2) (C − A − B) + B
```

This running time of this algorithm is $T(n) = 3T(n/2) + O(n) \approx O(n^{log_2 3}) \approx O(n^{1.59})$ using Master's Theorem.

Answering Divide-and-Conquer Questions on the Exam:

When given a question on the exam that asks you to come up with an algorithm to a problem, you should always include the following things:

1. **Steps of your algorithm.** Please explain this in English; **no credit** will be given for pseudocode or actual code. Imagine you are explaining the steps of an algorithm to someone who doesn't know the solution. If you choose to run an algorithm covered in lecture like Merge Sort, or FFT, you do not have to explain how it works you can just state and use the algorithm. However, in the runtime analysis section, please re-state the recurrence.
2. **Proof of correctness.** This is *not a formal proof*; rather, it is just a couple sentences where you explain why your algorithm works in all cases. An example where you would *not* receive credit for this is when you simply state an algorithm like Fast Multiplication but don't explain why the formula $2^n(x_L y_L) + 2^{n/2}((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R$ works. Don't be too worried about writing a lot for this; as long as us graders can tell you know your algorithm is correct by a couple sentences in your explanation, you're good!
3. **Runtime Analysis.** You are required to build a recurrence for your algorithm and use Master Theorem unless otherwise specified. Note that you will not be able to access the Master Theorem formula during the exam, so be sure to memorize it.

2.3 Modular Arithmetic

We define x ***modulo*** N to be the remainder of the division $\frac{x}{N}$, where x and N are integers. In other words, if $x = qN + r$ where $r \leq 0$, $x \bmod N = r$.

This allows us to define a **congruence** between integers given modulus N. x is congruent to y if $x \mod N = y \mod N$. This means that x and y can differ by exactly a multiple of N, or that N divides $(x - y)$. We denote this congruence as follows:

$$x \equiv y \pmod{N}$$

For example, $129 \equiv 65 \equiv 1 \pmod{64}$. We can also have negative numbers: $23 \equiv -1 \pmod{24}$. Essentially, once we define a modulus N, we limit the range of numbers to the set $\{0, 1, \ldots, N-1\}$ and wrap around numbers once going outside the range.

Let's briefly review an example of an algorithm that utilizes modular arithmetic: finding the greatest common divisor between two numbers. This is known as the **Euclidean Algorithm**:

```
Euclid(a, b):
    if b == 0:
        return a
    return Euclid(b, a mod b)
```

Euclid's Algorithm works based on the rule that if x and y are positive integers such that $x \geq y$, then $\gcd(x, y) = \gcd(x \bmod y, y)$.

Now let's review some additional operations that carry on to modular arithmetic. First, we can define the **modular multiplicative inverse** for an integer a modulo N. Suppose we have the following congruence:

$$a \cdot n \equiv 1 \pmod{N}$$

n is the multiplicative inverse of a, and a is the multiplicative inverse of n. However, this works differently than the multiplicative inverse in regular arithmetic; there are numbers which do not have such inverses! Consider $2n \equiv 1 \pmod{6}$. There is actually no solution to this congruence, which by definition implies that a^{-1} does not exist. Therefore there are numbers which are not invertible with a given modulus N. As it turns out, only a number a that is *relatively prime* to N has an inverse modulo N, meaning that $\gcd(a, N) = 1$. This is the **modular division theorem**.

We can also define modular exponentiation:

$$a^n \bmod N$$

is the remainder of the exponentiation a^n when dividing by N. As we lead up to the next section on RSA, we can try to find a method to evaluate large exponents when we use a prime modulus.

Fermat's Little Theorem states that given a prime number p,

$$a^{p-1} \equiv 1 \pmod{p}$$

> **Note:** We can also express Fermat's Little Theorem as
>
> $$a^p \equiv a \pmod{p}$$
>
> We can derive this from the original congruence: $a^p = a^{p-1} \cdot a \equiv 1 \cdot a \equiv a \pmod{p}$

This allows us to evaluate really large exponents in modular arithmetic by trying to reduce the exponent to $p-1$, as long as p is prime. Let's see a couple examples:

1. Find $3^{31} \bmod 7$. We have by Fermat that $3^6 \bmod 7 \equiv 1$, so $3^{31} = (3^5)^6 3^1 \equiv 1^6 3^1 \equiv \boxed{3} \pmod 7$.
2. Find $128^{129} \bmod 17$. We have by Fermat $(128^{16})^8(128) \equiv 128 \pmod{17}$. We have $128 = 17(7)+9 \equiv \boxed{9} \pmod{17}$.
3. Find $29^{25} \bmod 11$. We have by Fermat that $29^{25} = (29^{10})^2(29)^5 \equiv 29^5 \pmod{11}$. As $29 \equiv 7 \pmod{11}$, we have $29^5 \equiv 7^5 \pmod{11}$. We could technically now calculate the final result, but to make the expression even more simple we can use the fact that $7 \equiv -4 \pmod{11}$! We then get

$$7^5 \equiv 7 \cdot (-4)^4 \equiv 7 \cdot 256 \equiv 7 \cdot 3 \equiv 21 \equiv \boxed{10} \pmod{11}$$

In general, try to simplify the exponent to the form $mx + b$, where $m = p - 1$. Then, since a^{mx} will reduce to 1, you just have to simplify a^b. If the base a is too large, simplify it by calculating $a \bmod p$ and then replace the base with the result.

2.4 RSA

Modular Exponentiation:
The recursive function used to achieve modular exponentation is as follows:

```
def modexp(x, y, N):
  if y = 0, return 1
  z = modexp(x, y / 2, N)
  if y is even return z^2 (mod N)
  else         return z^2 * x (mod N)
```

The most straightforward example for this technique is for the expression: $x^y \bmod (n)$
There are only two scenarios which we have to address to prove correctness in this case:

- If y is even:

$$\begin{aligned} z &\equiv x^{y/2} \pmod{N} \\ z^2 &\equiv x^{y/2}x^{y/2} \pmod{N} \\ z^2 &\equiv x^y \pmod{N} \end{aligned}$$

- If y is odd:

$$\begin{aligned} z &\equiv x^{(y-1)/2)} \pmod{N} \\ z^2 &\equiv x(x^{(y-1)/2)})^2 \pmod{N} \\ z^2 &\equiv x^{(y-1)/2+(y-1)/2+1} \pmod{N} \\ z^2 &\equiv x^{y+1} \equiv x(x^y) \pmod{N} \end{aligned}$$

How would you approach a runtime analysis of this algorithm? How many recursive calls are there? How much computation is happening at each step?

We can see that there are N recursive calls happening, each performing a multiplication of N bits by N bits. We can conclude that the runtime of this function is $O(N^3)$.

Primality:
We typically use Fermat's Little Theorem to test for primality; keep in mind that this theorem is not absolute in its certainty.

It's important to recognize that a lot of exceptions to this rule have been identified, including the (infinite set of) Carmichael numbers.

However, if we design an algorithm to randomly choose a, we can conclude that a number is likely composite if the equality does not hold. Any value a that holds this equality for a known composite p is known as a Fermat's Liar, while a value of a that does not hold this value would be a part of Fermat's witnesses.

Totients:
$\phi(n)$ = Amount of numbers relatively prime to N which are less than N.
Remember: For a prime p, $\phi(p) = p - 1$.
Otherwise: For two primes p, q, $\phi(pq) = (p-1)(q-1)$ (this does not work for p^2)

$a^{\phi(N)} \equiv 1 \pmod{N}$. Remember Fermat's Little Theorem; can you see where this comes from?

The Actual RSA Algorithm:
With all the information you've learned so far, we can now discuss important facets of the RSA Algorithm itself.

Procedure:

- B generates large primes p, q
- B then computes $N = pq$
- B computes e, a number relatively prime to $(p-1)(q-1)$
- B computes d $\equiv e^{-1}(mod(p-1)(q-1))$
- $pk = (N, e)$, $sk = d$
- B broadcasts pk
- A computes and sends m^e (mod N)
- B computes $m \equiv (m^e)^d$ (mod N)

We can understand this computation mathematically in the following manner:

$$\begin{aligned} ed &\equiv 1 \pmod{p-1)(q-1)} \\ ed &= k(p-1)(q-1)+1 \\ (m^e)^d &\equiv m^{ed} \pmod{N} \\ (m^e)^d &\equiv m^{k(p-1)(q-1)+1} \pmod{N} \\ (m^e)^d &\equiv m^{k\phi(N)+1} \pmod{N} \\ (m^e)^d &\equiv m(1)^k \pmod{N} \\ (m^e)^d &\equiv m \pmod{N} \end{aligned}$$