CS 3510 C: Design & Analysis of Algorithms 08/31/2023

Practice Exam 1

Abrahim Ladha

- This is the CS 3510 practice exam for Exam 1. This does **not** approximate the difficulty or length of the actual exam; it serves as just a big question bank for you to practice!
- Topics include: Big-O, Master Theorem, Divide and Conquer, Arithmetic, Cryptography.
- Note that this assignment does not need to be submitted, but we highly recommend going through it to prepare for the exam!

Big-O

1.) Explain why or why not the following claims are true (no need for any formal proofs):

1. If $a \in \mathbb{Z}_+$, then
   $a^{n+1} = \mathcal{O}(a^n)$

   Solution: This is true. It could be rewritten as $a \cdot a^n$ which is $O(a^n)$.

2. If $a \in \mathbb{Z}_+$, then
   $a^{2n} = \mathcal{O}(a^n)$

   Solution: This is not true by algebraic laws.

3. If $f(n) = \Omega(p(n))$ and $f(n) = \mathcal{O}(q(n))$, then
   for all $n \in \mathbb{R}$, $p(n) \leq q(n)$

   Solution: This is not true by definitions of the asymptotic descriptors.

4. If $a, b \in \mathbb{Z}_+$ and $a > b$, then
   $n^a - n^b = O(n^{a-b})$

   Solution: This is not true by algebraic rules.

5. If $a, b \in \mathbb{Z}_+$ and $a > b$, then
   $n^b - n^a = O(n^a)$

   Solution: This is true.

6. If $a \in \mathbb{Z}_+$, then
   $\mathcal{O}(n^a) - \mathcal{O}(n^a) = 0$

   Solution: This is not true. You could choose two functions that are $\mathcal{O}(n^a)$ which could cancel out, but you could just as easily choose something like n and 0 which would show its not.

2.) For each of the following, describe the relationship between $f(n)$ and $g(n)$ using Big-O notation.

1. $f(n) = n^3 + 2n, \quad g(n) = 12n^2 + 24\sqrt{n}$
2. $f(n) = (\log n)^2, \quad g(n) = \log n + \sqrt{n}$
3. $f(n) = 2\log_5(3^n), \quad g(n) = n + 4n^{0.2}$
4. $f(n) = 8^{\log_7 n}, \quad g(n) = n \log n$

Solution:

1. $g(n) = O(f(n))$. $f(n)$ has an n^3 term, which grows strictly faster than both the n^2 and $n^{0.5}$ terms in $g(n)$.

2. $f(n) = O(g(n))$. $f(n)$ is a power of $\log n$, which grows slower than any power of n, and $g(n)$ has an $n^{0.5}$ term.

3. $f(n) = O(g(n))$ and $g(n) = O(f(n))$, this requires a bit of manipulation: by log properties, $f(n) = 2 \cdot n \cdot \log_5(3)$, which is a constant times n, and in $g(n)$ the $n^{0.2}$ term is dominated by n, so these both grow like $O(n)$.

4. $g(n) = O(f(n))$. By log properties, $f(n) = (7^{\log_7 8})^{\log_7 n} = n^{\log_7 8}$, and since $8 > 7$, we know $\log_7 8 > 1$. Since $g(n) = n \log n$, it grows faster than n, but slower than n^c for any $c > 1$.

3.) Show that, c is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is

$$g(n) = \begin{cases} \Theta(1) & \text{if } c < 1. \\ \Theta(n) & \text{if } c = 1. \\ \Theta(c^n) & \text{if } c > 1. \end{cases}$$

***Hint**: Use geometric series.*

Solution: We use the geometric series formula $g(n) = (1 - c^{n+1})/(1 - c)$.

If $c < 1$ (recall c is assumed to be positive) then c^{n+1} becomes arbitrarily small as $n \to \infty$, so the series converges to $1/(1 - c)$ which is a constant and therefore $\Theta(1)$.

If $c = 1$, then $g(n) = 1 + 1 + \cdots + 1 = n$ which is trivially $\Theta(n)$.

If $c > 1$, write $g(n) = (1/(c-1))c^{n+1} - 1/(c-1) = (c/(c-1))c^n - 1/(c-1)$ and observe this is a constant times c^n and then subtracting a constant, so $g(n)$ is $\Theta(c^n)$.

Master Theorem

4.) Give the big-$\mathcal{O}$ runtime of the following recurrence relations.

1. $T(n) = 2T(n/4) + \mathcal{O}(n)$

   Solution:

   $$a = 2, b = 4, d = 1$$
   $$1 > \log_4 2$$
   $$\mathcal{O}(n)$$

2. $T(n) = 2T(n/4) + \mathcal{O}(1)$

   Solution:

   $$a = 2, b = 4, d = 0$$
   $$0 < \log_4 2$$
   $$\mathcal{O}(\sqrt{n})$$

3. $T(n) = 2T(n/4) + \mathcal{O}(\sqrt{n})$

   Solution:

   $$a = 2, b = 4, d = \frac{1}{2}$$
   $$\frac{1}{2} = \log_4 2$$
   $$\mathcal{O}(\sqrt{n} \log n)$$

4. $T(n) = 3T(n/3) + \mathcal{O}(1)$

   Solution:

   $$a = 3, b = 3, d = 0$$
   $$0 < \log_3 3$$
   $$\mathcal{O}(n)$$

5. $T(n) = 4T(n/3) + \mathcal{O}(n^2)$

Solution:

$$a = 4, b = 3, d = 2$$
$$2 > \log_3 4$$
$$\mathcal{O}(n^2)$$

6. $T(n) = 5T(2n/3) + \mathcal{O}(n^2)$

Solution:

$$a = 5, b = 3/2, d = 2$$
$$2 < \log_{3/2} 5$$
$$\mathcal{O}(n^{\log_{3/2} 5})$$

Divide & Conquer

5.) Given an ordered array, A, of size n with unique integers, and two boundary numbers, l and u, design a Divide & Conquer algorithm to determine the number of integers within A that lie between l and u, both inclusive.

Solution:

Algorithm:

Firstly, we identify the index of the smallest number within the given range and likewise, the largest number.

For a subproblem with one element, it's possible to validate if it's between l and u in $O(1)$. If so, we return its index from A. Otherwise, we indicate it doesn't belong to the range.

For arrays larger than one element, we inspect the middle value. If it's within our range, we search for the smallest number on the left. If we find an index leftwards, that represents the index of our smallest number in the larger array. Otherwise, it's in the middle.

Conversely, when seeking the largest number, if the central element is within the range, we solve the rightward subproblem. If an index emerges, it's the largest number's index. If not, the center holds the index.

If the middle number is less than l, we refer to the right subproblem's results. If it's larger than u, the left is referred to.

Finally, since this is an inclusive range, the cardinality is the difference in indices plus 1.

Correctness:

This solution is effective because it leverages the binary search technique to accurately identify the lower and upper bounds. With the array being sorted and all values being distinct, binary search assures that whenever a bound is determined, it's the precise representation of that limit within the stipulated range. Each decision, made by comparing the midpoint value with the range's boundaries, ensures consistent and accurate narrowing down to the correct position.

Runtime:

Every procedure adheres to $T(n) = T\left(\frac{n}{2}\right) + O(1)$. Applying the Master Theorem with parameters $a = 1, b = 2$, and $d = 0$, we can infer $T(n) = O(\log n)$.

Note that $a = 1$ because the searches for the smallest and largest numbers each result in one subproblem, with each subproblem being of size $b = \frac{n}{2}$. Verifying the central value in each iteration has a time complexity of $O(1)$.

6.) Let x and y be two n-digit binary numbers, where n is a power of 3. Let x_L, x_M and x_R consist of the first third, middle third, and final third of the digits of x, so that $x = 2^{\frac{2n}{3}}x_L + 2^{\frac{n}{3}}x_M + x_R$, and define y_L, y_M, and y_R analogously.

(a.) Express xy in terms of $x_L, x_M, x_R, y_L, y_M, y_R$. Simplify your answer.

Solution: We have that

$$xy = (2^{2n/3}x_L + 2^{n/3}x_M + x_R) * (2^{2n/3}y_L + 2^{n/3}y_M + y_R)$$

$$= 2^{\frac{4n}{3}}x_Ly_L + 2^n x_Ly_M + 2^{\frac{2n}{3}}x_Ly_R + 2^n x_My_L + 2^{\frac{2n}{3}}x_My_M + 2^{\frac{n}{3}}x_My_R + 2^{\frac{2n}{3}}x_Ry_L + 2^{\frac{n}{3}}x_Ry_M + x_Ry_R$$

$$= 2^{\frac{4n}{3}}x_Ly_L + 2^n(x_Ly_M + x_My_L) + 2^{\frac{2n}{3}}(x_Ly_R + x_My_M + x_Ry_L) + 2^{\frac{n}{3}}(x_My_R + x_Ry_M) + x_Ry_R$$

(b.) Give a recursive algorithm that calculates the above expression with a recurrence relation of $T(n) = 6T(n/3) + O(n)$ and calculate its runtime.

Solution: We could create 9 subproblems to calculate $x_Ly_L, x_Ly_M, x_My_L, \ldots$ and then plug into the equation from part (a), but we can do better using algebraic maniupla-tion.

Consider subproblems $A = x_Ly_L$, $B = x_My_M$, and $C = x_Ry_R$. We make additional subproblems D, E, F such that

$$D = (x_L + x_M)(y_L + y_M) = x_Ly_L + x_Ly_M + x_My_L + x_My_M$$

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

$$F = (x_M + x_R)(y_M + y_R) = x_My_M + x_My_R + x_Ry_M + x_Ry_R$$

Our original expression from part (a) now becomes

$$xy = 2^{\frac{4n}{3}}A + 2^n(D - A - B) + 2^{\frac{2n}{3}}(B + E - A - C) + 2^{\frac{n}{3}}(F - B - C) + C$$

So $A - F$ are our subproblems. It takes $\mathcal{O}(n)$ non-recurisve work (bit partitioning and polynomial reconstruction). This combines to give the target runtime recurrence relation of $T(n) = 6T(n/3) + \mathcal{O}(n)$. Using Master's Theorem, this gives us a runtime of $\mathcal{O}(n^{\log_3 6})$.

7.) You've become an explorer and you've come across some mountains and valleys you need to cross. Unluckily for you, you can't climb mountains yet, so you need to find a valley. You're given a list representing a 2d height map like the one shown below.

An index, i, in the list, A, is considered a valley if $A[i] \leq A[i-1] \wedge A[i] \leq A[i+1]$ is true. Give a quick solution for finding valleys. Now assume, $A[1] > A[2]$ and $A[n-1] < A[n]$. Give a complete solution to find a valley under these conditions.

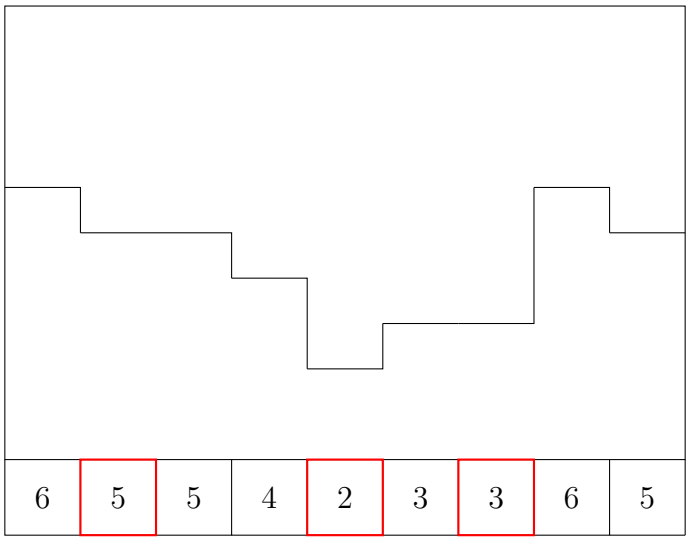


Figure 1: This figure displays an example of what a list of numbers would be represented as; the possible valleys are highlighted in red.

Solution:

You could pretty easily check for valleys by scanning the array. This would take $\mathcal{O}(n)$ time. However the conditions specified above imply that one valley exists. so we can use a divide and conquer solution to achieve a faster runtime.

Algorithm: We take our array, compute the middle element, and check if that element is a valley. If it is we return, otherwise we check the angle of the slope and if the slope is going up, we recurse on the first half of the array otherwise we recurse on the second half. When we find a valley, we return its index.

Correctness: This is correct because we know that at some point, the array's change will need to change it's direction going from decreasing to increasing. At this point, we will find a valley. The algorithm works by shrinking the size of the array until we find a valley which we know must exist.

Runtime: We do a constant amount of work at each step, and we partition the array in two at each step, recursing down one of the partitions. Our recurrence relation is $T(n) = T(n/2)+O(1)$. Our constants are $a = 1, b = 2, d = 0$, meaning we use the $O(n^d \log n)$ branch of the master theorem because $d = \log_b a$. This becomes $\mathcal{O}(n^0 \log n) = \mathcal{O}(\log n)$.

8.) Guess the Path

There exists an $n \times n$ grid in which you can only move down and to the right. Your goal is to escape the grid by starting at the top left corner $(1, 1)$ and moving to the bottom right corner (n, n). However, there is only one correct path that will allow you to escape, and only the TAs know the answer. Your goal is to determine this path by querying the TAs.

(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 1)	(3, 2)	(3, 3)	(3, 4)
(4, 1)	(4, 2)	(4, 3)	(4, 4)

Figure 1: Correct path

Treat a query to the TAs as a blackbox algorithm that takes in a path, and returns the coordinates of tiles in your proposed path that are in the correct path. Assume that this takes $\mathcal{O}(1)$ time for simplicity. For example, the correct path in a 4×4 grid is shown in Figure 1. If you query the paths shown in Figures 2 and 3 (marked in bold and italic), the green tiles show coordinates returned by the TAs.

(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 1)	(3, 2)	(3, 3)	(3, 4)
(4, 1)	***(4, 2)***	***(4, 3)***	***(4, 4)***

(a) Figure 2

(1, 1)	***(1, 2)***	***(1, 3)***	(1, 4)
(2, 1)	(2, 2)	***(2, 3)***	(2, 4)
(3, 1)	(3, 2)	***(3, 3)***	***(3, 4)***
(4, 1)	(4, 2)	(4, 3)	***(4, 4)***

(b) Figure 3

(a.) Propose an algorithm to find the correct path that takes $\mathcal{O}(n^2)$ time.

Solution: Build paths by starting at each row, going all the way to the right, and down towards the goal. Building such a path takes $\mathcal{O}(n)$. If we build such a path for every row, we will visit each cell at least once, and so we will find all correct coordinates. This takes $\mathcal{O}(n^2)$ time, as our queries are assumed to take $\mathcal{O}(1)$ time.

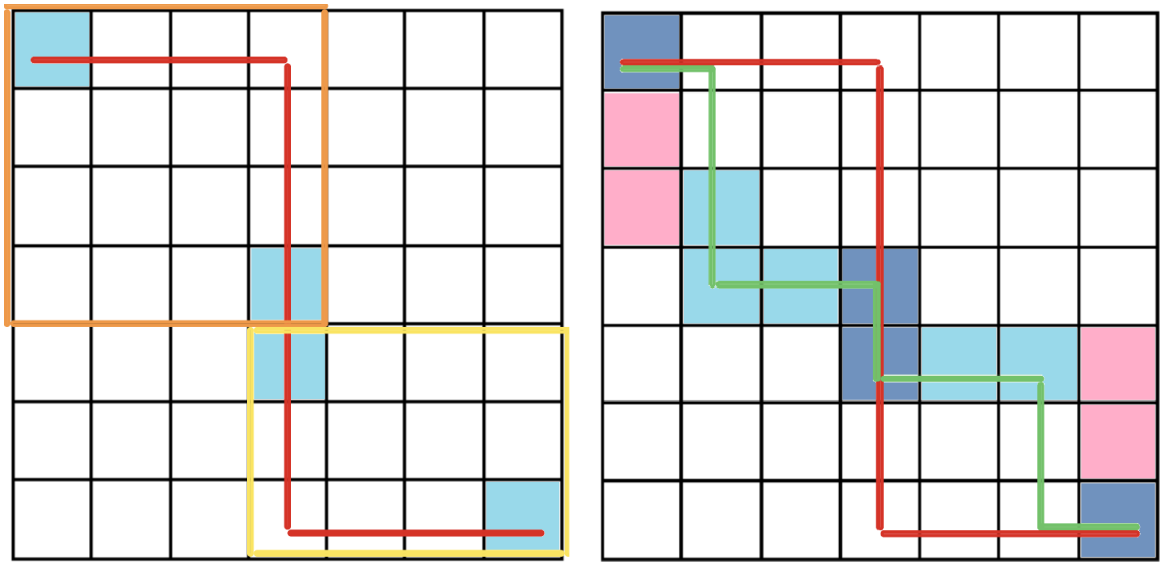
(b.) Now, design a Divide-and-Conquer algorithm to find the correct path that is faster than your approach in part (a).

Solution: **Key observation:** If we move through an entire row / column, we are **guaranteed** to intersect at least one point in the correct path.

If we move directly to the right to the midpoint of the first row, then move all the way down, and then to the right to the end, we will intersect the correct path at least once. Now that we know more of the path, we can subdivide the grid into two smaller grids.

For example, if our first query finds the intersection point (x, y), then we can subdivide the problem into two problems: find the path from $(1, 1)$ to (x, y), and find the path from (x, y) to (n, n). We repeat this process recursively. We can solve all sub-problems simultaneously in one query, by joining the sub-problem paths using the known correct tiles.

This approach can be seen below on a 7×7 grid. The red line shows the first query, and the light blue tiles show the TAs response. The orange and yellow boxes show the sub-problems we divide into. The green path shows the second query, After the second query, the pink tiles can be determined because they are the only moves left possible. They can also be found by performing another recursive step; both solutions work.



Our recurrence is $T(n) = 2T(n/2) + \mathcal{O}(n)$. The $\mathcal{O}(n)$ extra work comes from building the initial path that we send to the query, as well as combining subproblem paths to get our returned path. This results in a runtime of $\mathcal{O}(n \log n)$ which is faster than previous runtime.

(c.) What changes about your solutions to part a and b if the grid is an $m \times n$ rectangle? Give new runtimes of both algorithms.

Solution: For part (a), we may choose to iterate through rows or columns. For part (b), there are no changes; we still use paths that go through the midpoint of the matrix. Only the runtimes change - part (a)'s runtime becomes $\mathcal{O}(\max(mn, n^2, m^2))$, while part (b)'s recurrence becomes $T(n, m) = 2T(n/2, m/2) + \mathcal{O}(m + n)$ which yields runtime of $\mathcal{O}((m + n) \log(\max(m, n))$.

Modular Arithmetic & RSA

9.) Solve the following problems:

1. Find $2^{20} + 3^{30} + 4^{40} + 5^{50} + 6^{60} \bmod 7$.

   Solution: By Fermat's Little Theorem, $2^6 \equiv 3^6 \equiv 4^6 \equiv 5^6 \equiv 6^6 \equiv 1 \bmod 7$. As a result, $2^{20} + 3^{30} + 4^{40} + 5^{50} + 6^{60} \equiv 2^2 + 3^0 + 4^4 + 5^2 + 6^0 \equiv 4 + 1 + 4 + 4 + 1 \equiv 14 \equiv 0 \bmod 7$.

2. Solve the congruence $x^{103} \equiv 4 \bmod 11$.

   Solution: By Fermat's Little Theorem, $x^{10} \equiv 1 \bmod 11$. As a result we have $x^{103} \equiv x^3 \bmod 11$. Solving $x^3 \equiv 4 \bmod 11$, if we try all the values from $x = 1$ through $x = 10$, we find that $5^3 \equiv 4 \bmod 11$. $x \equiv 5 \bmod 11$.

10.) Suppose Tom wants to send Jerry a message using the RSA scheme.

1. Who should set up the RSA key?

2. Suppose he (the person generating the key) chooses as the two primes $p = 23$ and $q = 29$, and chooses as the encryption exponent $e = 3$. What number must he select for d, the decryption exponent? Show your work.

3. If the message being sent is $m = 15$, what is the encrypted message? Please show your work.

4. Let us say that when you generate your RSA key you pick p and q as 1024-bit primes, but by some luck when your algorithm randomly chooses the encryption exponent it gets a small value, $e = 3$. At first glance, it might seem helpful since it would require less computation for the sender to encrypt their messages. But there's actually a problem here, and you should modify your algorithm to ensure that e is suitably large. Can you think of why this is?

   Hint: Think about what happens when the message is really short, say a few bits. This problem can also be fixed by padding the message with extra bits, which is often how this is resolved in general.

Solution:

1. Jerry

2. We need to find the inverse of e mod $(p-1)(q-1)$. Though the standard method is to apply the Euclid Algorithm, here, since all the numbers are small, we can solve it by hand instead. It is easy to find that 3 divides $2 \times 22 \times 28 + 1$, so $d = 411$.

3. The encrypted message should be m^e mod N, which is 15^3 mod 667. The answer is 40.

4. When the message m is too small such that $m^e < pq$, then one can directly calculate the e-th root of the encrypted message. If e is large, then m^e is likely to be much greater than pq.