CSCE629 Analysis of Algorithms

Homework 2

Texas A&M U, Fall 2019 — *09/06/19*

Lecturer: Fang Song — *Due: 10am, 09/13/19*

Instructions.

- Typeset your submission by LaTeX, and submit in PDF format. Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct, and you will get significantly more partial credit if you clearly identify the gap(s) in your solution. You may opt for the "Ill take 15%" option (details in Syllabus).
- You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources** for each problem. Be ready to explain your solutions orally to a course staff if asked.
- For problems that require you to provide an algorithm, you must give a precise description of the algorithm, together with a proof of correctness and an analysis of its running time. You may use algorithms from class as subroutines. You may also use any facts that we proved in class or from the book.

This assignment contains 5 questions, 13 pages for the total of 70 points and 15 bonus points. A random subset of the problems will be graded.

1. (More asymptotics)
   (a) (5 points) Prove that for any positive numbers $a, b > 0$, we have $n^b = \omega(\log^a(n))$. There are several ways to approach this. One way is to use L'Hospital's rule for evaluating limits together with the following fact: $f(n) = \omega(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$.
   (b) (5 points) Prove or disprove: If $h(n) = \lceil n\log(n)\rceil$, then $n = \Theta(h(n)/\log h(n))$.

Solution:

(a) Given, $f(n) = n^b$ and $g(n) = \log^a(n)$. To prove $f(n) = \omega(g(n))$ we can use the limit evaluated at infinity for this fraction $f(n)/g(n)$. Let us recall the definition of the ω-notation. $\omega(g(n)) = \{f(n)$: for all positive constants $c > 0$, there exist a constant $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

For the given problem, $a, b > 0$ are positive constants then, (I am assuming log is natural logarithmic i.e., log to base e)

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n^b}{\log^a(n)} \\
&= \lim_{n\to\infty} \frac{bn^{b-1}}{(a\log^{a-1}(n))/n} \\
&= \lim_{n\to\infty} \frac{bn^b}{a\log^{a-1}(n)} \\
&= \lim_{n\to\infty} \frac{b^a n^b}{a!} \\
&= +\infty
\end{aligned}$$

Since the given problem initially is in indeterminate form (∞/∞) so we can apply L'Hospital's rule. After simplifying the terms we notice that we can apply L'Hospital's rule iteratively a times i.e., until the log term in the denominator vanishes. Then we end up with n^b term in numerator and as $n \to \infty$ limit will become infinity. Hence $n^b = \omega(\log^a(n))$ which also means $\log^a(n) = o(n^b)$. Thus any, positive polynomial function grows faster than any polylogarithmic function. This property is known as **transpose symmetry**.

(b) We can neglect the ceiling function, then we can write the problem as $n = \Theta(n \log n / \log(n \log n))$. To prove $n = \Theta(g(n))$ we need to prove $n = O(g(n))$ and $n = \Omega(g(n))$ (this comes from the theorem provided in the CLRS textbook).

If $n = O(n \log n / \log(n \log n))$ then there exist a constant c_1 and n_1 such that $n \le c_1(n \log n / \log(n \log n))$ for $n \ge n_1$. We can simplify further more

$$\begin{aligned}
n &\le c_1 \frac{n \log n}{\log(n \log n)} \\
\frac{\log(n \log n)}{\log n} &\le c_1 \\
\frac{\log n + \log \log n}{\log n} &\le c_1.
\end{aligned}$$

The above ratio for large values of n (i.e., $n \to \infty$) it will be close to 1. Hence we can choose $c_1 = 50$ and $n_1 = 2^{2^2}$.

If $n = \Omega(n \log n / \log(n \log n))$ then there exist a constant c_2 and n_2 such that $n \geq c_2(n \log n / \log(n \log n))$ for $n \geq n_2$. We can simplify further more

$$\begin{aligned} n &\geq c_2 \frac{n \log n}{\log(n \log n)} \\ \frac{\log(n \log n)}{\log n} &\geq c_2 \\ \frac{\log n + \log \log n}{\log n} &\geq c_2. \end{aligned}$$

The above ratio for large values of n (i.e., $n \to \infty$) it will be close to 1. Hence we can choose $c_2 = 1/50$ and $n_2 = 2^{2^2}$.

Since we proved there exist some constants c_1, c_2 and n_1, n_2 such that

$$\begin{aligned} n &= \Omega(n \log n / \log(n \log n)) \\ n &= O(n \log n / \log(n \log n)) \end{aligned}$$

From the theorem 3.1 from Chapter 3 of Introduction to Algorithms textbook $n = \Theta(n \log n / \log(n \log n))$ and since ceiling of $h(n) = \lceil n \log n \rceil \leq n \log n + 1$ for large values of n, we can neglect the 1 and bound it with $n \log n$. Hence, $n = \Theta(h(n)/\log(h(n)))$.

2. (Recurrence) Solve the following recurrences.
   (a) (5 points) $A(n) = 2A(n/4) + \sqrt{n}$
   (b) (5 points) $B(n) = 2B(n/4) + n$
   (c) (5 points) $C(n) = 3C(n/3) + n^2$
   (d) (5 points (bonus)) $D(n) = \sqrt{n}D(\sqrt{n}) + n$

Solution: We will use the Master Theorem to solve the recursion problems whenever possible.

(a) Given problem is in the following form $T(n) = aT(n/b) + f(n)$. Where $a = 2$, $b = 4$, $n^{\log_b a} = \sqrt{n}$, and $f(n) = \sqrt{n}$. Since $f(n) = \Theta(\sqrt{n})$, then $T(n) = \Theta(\sqrt{n} \log n)$.

(b) Given problem is in the following form $T(n) = aT(n/b) + f(n)$. Where $a = 2$, $b = 4$, $n^{\log_b a} = \sqrt{n}$, and $f(n) = n$. Since $f(n) = \Omega(n^{0.5+\epsilon})$ where $\epsilon \approx 0.5$ (we know this from the reflexive property that $f(n) = \Omega(f(n))$). For sufficiently large n we have that $af(n/b) = 2f(n/4) = n/2 \leq cf(n)$ for $c = 1/2$ Hence, $T(n) = \Theta(n)$.

(c) Given problem is in the following form $T(n) = aT(n/b) + f(n)$. Where $a = 3$, $b = 3$, $n^{\log_b a} = n$, and $f(n) = n^2$. Since $f(n) = \Omega(n^{1+\epsilon})$ where $\epsilon \approx 1$ (we know this from the reflexive property that $f(n) = \Omega(f(n))$). For sufficiently large n we have that $af(n/b) = 3f(n/3) = n^2/3 \leq cf(n)$ for $c = 1/3$ Hence, $T(n) = \Theta(n^2)$.

(d) We can't apply master method to this problem. One of the ways to solve this problem would be to expand the recursion iteratively and try to find some pattern in it i.e., we have to write the recursion tree. For this problem we will assume that for $n \leq 2$, $T(n)$ will be constant. If $n = 2$, then $T(2)$ can be assumed to be last term in recursion and it will be the leaf of the tree.

Claim: *For this recurrence rule we observe that for every level of the tree we have to do n amount of work.* Lets look into this by calculating the total work needs to be done at each level of the tree. At the root of the tree we have only one node and we have to do n amount of work and then the tree splits into $\sqrt{n}$ branches. At second level we have $\sqrt{n}$ nodes and for each node we have $\sqrt{n}$ work to do hence total work will be n and each node splits into $n^{1/4}$ branches. At third level we will have total $n^{1/4} * n^{1/2}$ branches. For each node we will have $n^{1/4}$ work, so total work will be $n^{1/4} * n^{1/2} * n^{1/4} = n$. For i^{th} level we have $n^{1-\frac{1}{2^i}}$ leaves and $n^{\frac{1}{2^i}}$ work per node hence total work will be n. For simplification of the problem lets assume that n is number raised to even power. So the recursion tree will stop at $T(2)$ i.e., when $n^{\frac{1}{2^i}} = 2 \implies i = \log\log n$. Hence the height of the tree will be $\log\log n$ and for each level we have n amount of work and total work will be $n \log\log n$. Hence the bound is $T(n) = O(n \log\log n)$.

Note: Generally when there is a division by a constant in the recurrence equation then recursion tree will have $\log n$ height and when using sqrt we get $\log\log n$ height. Algorithms involving $\sqrt{n}$ will have $\log\log n$ height and they grow very slowly vs input size.

3. (10 points) (Domain transformation) We analyzed the running time of Mergesort by the recurrence $T(n) = 2T(n/2) + O(n)$. The actual Mergesort recurrence is somewhat messier:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)\,.$$

We'll justify in this problem that ignoring the ceilings and floors in a recurrence is okay afterall using a technique called *domain transformation*.

First, because we are deriving an upper bound, we can safely overestimate $T(n)$, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n\,.$$

Second, we define a new function $S(n) = T(n + \alpha)$ for some α. **You are to complete the second step.** Show that you can find a nice α so that $S(n) \leq 2S(n/2) + O(n)$ does hold, and conclude from there that $T(n) = O(n \log n)$.

[Exercise (Do not turn in). Show how to remove floors by similar arguments.]

Solution: Our hope in doing domain transformation is that getting an exact solution with the messy recurrence equation is hard. But if we do domain transformation then we can produce a tight asymptotic solution. Given, that we are overestimating the time bound by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$\begin{aligned} T(n) &\leq 2T(\lceil n/2 \rceil) + n \\ &\leq 2T(n/2 + 1) + n \end{aligned}$$

Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \leq 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

$$\begin{aligned} S(n) \leq 2S(n/2) + O(n) &\quad\Longrightarrow\quad T(n+\alpha) \leq 2T(n/2 + \alpha) + O(n) \\ T(n) \leq 2T(n/2 + 1) + n &\quad\Longrightarrow\quad T(n+\alpha) \leq 2T((n+\alpha)/2 + 1) + n + \alpha \end{aligned}$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that α is 2. The Master Theorem now tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n)$$

Similarly we can remove floor by:

$$\begin{aligned} T(n) &\geq 2T(\lfloor n/2 \rfloor) + n \\ &\geq 2T(n/2 - 1) + n \end{aligned}$$

Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \geq 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

$$\begin{aligned} S(n) \geq 2S(n/2) + O(n) &\quad\Longrightarrow\quad T(n+\alpha) \geq 2T(n/2 + \alpha) + O(n) \\ T(n) \geq 2T(n/2 - 1) + n &\quad\Longrightarrow\quad T(n+\alpha) \geq 2T((n+\alpha)/2 - 1) + n + \alpha \end{aligned}$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 - 1$, which implies that α is -2. The Master Theorem now tells us that $S(n) = \Omega(n \log n)$, so

$$T(n) = S(n+2) = O((n+2)\log(n+2)) = \Omega(n \log n)$$

So, $T(n) = \Theta(n \log n)$. Domain transformations are useful for removing floors, ceilings, and lower order terms from the arguments of any recurrence that otherwise looks like it ought to fit either the Master Theorem or the recursion tree method.

4. (Quicksort) We were not precise about the running time of Quicksort in class (for a good reason). We will give some case studies in this problem (and appreciate the subtlety).
   (a) (10 points) Given an input array of n elements, suppose we are unlucky and the partitioning routine produces one subproblem with $n-1$ elements and one with 0 element. Write down the recurrence and solve it. Describe an input array that costs this amount of running time to get sorted by Quicksort.
   (b) (5 points) Now suppose that the partitioning always produces a 9-to-1 proportional split. Write down the recurrence for $T(n)$ and solve it.
   (c) (5 points) What is the running time of Quicksort when all elements of the input array have the same value?

Solution: Generally for any Divide-and-Conquer problems we write the recursion relation as follows

$$T(n) = \begin{cases} \Theta(1) & if n \le c \\ aT(n/b) + D(n) + C(n) & otherwise \end{cases}$$

$T(n)$ is the running time on a problem of size n. If the problem size is small enough, say $n \le c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. It takes time $T(n/b)$ to solve one subproblem of size $n = b$, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, then we end up with above recursion equation.

All Divide-and-Conquer problems algorithms have same boilerplate template i.e., we have (a) **Divide** the problem into a number of subproblems that are smaller instances of the same problem, (b) **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner, and (c) **Combine** the solutions to the subproblems into the solution for the original problem.

In Quicksort, no work is needed to combine the sorted array because sorting happens on the original array hence $C(n) = 0$ (no work in combining). Running time of Divide routine or the partition routine in Quicksort is $D(n) = \Theta(n)$, where n is the problem size. The for loop makes exactly n iterations, each of which takes at most constant time. The part outside the for loop takes at most constant time. Since n is the size of the subarray, partition routine takes at most time proportional to the size of the subarray it is called on. Running time of Conquer routine will depend how lucky we are in partitioning the array.

(a) **Worst-case partitioning** The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

$\Theta(n)$ can be written as n in the equation which give us $T(n) = T(n-1) + n$. If we write the recursion tree for this then we get a tree with one branch at each level and the work done at each will be $n, n-1, n-2, \ldots, 1$. Hence total work needed would be summation of the arithmetic series which will give us $\Theta(n^2/2 + n/2)$. This can be approximated to $T(n) = \Theta(n^2)$.

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ occurs when the input array is already completely sorted; a common situation in which insertion sort runs in $O(n)$ time. We can give more examples for worst-case scenarios for Quicksort, for example reverse sorted array, or all the elements are same in the array. Input array with worst-case running time example: [1, 2, 3, 4, 5, 6]. For this example Quicksort algorithm has to go through each element and ends up calling the sort function again and again; as the array is sorted every time it picks the pivot it will be minimum.

(b) **Balanced partitioning** Here we have unbalanced partitioning of 90:10 split. For this we can write the recurrence as $T(n) = T(9n/10) + T(n/10) + \Theta(n)$. To solve this recurrence problem we will use recursion trees. At root the tree will split in $9/10th$ and $1/10th$ parts of the original problem and this will keep on repeating for each node. Since, the $1/10th$ decreases faster it will reach to $T(1)$ faster than its counter part. This will happen when $n/(10^i) = 1 \implies i = \log_{10} n$. Also we observe that for each level with depth less than $\log_{10} n$ we will need to cn amount of work to be done (here c is a positive constant hidden in $\Theta(n)$). At the subtrees of the $9/10th$ part reach to $T(1)$ when $n/((10/9)^j) = 1 \implies j = \log_{10/9} n$. At for each level at depth between i and j we need to do at most cn work. Hence total work we need that has to done can be bounded by $O(n \log_{10/9} n)$. Since $\log_a n = \log_b n / \log_b a$, we can write $n \log_{10/9} n$ as $n \log n$ and neglect the term $\log_2 10/9$. Hence, total cost of Quicksort for this case is therefore $T(n) = O(n \log n)$.

Note: With a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(n \log n)$ time; asymptotically the same as if the split were right down the middle. Indeed, even a 99-to-1 split yields an $O(n \log n)$ running time. In fact, any split of *constant* proportionality yields a recursion tree of depth $\Theta(\log n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \log n)$ whenever the split has constant proportionality.

(c) If all the input values of the array are same then this comes under the case of the worst-partitioning as discussed in the above parts. Hence for array with all same values will have $\Theta(n^2)$ running time where n is the input array size. To reiterate the above discussion the traditional implementation of Quicksort which partitions the problem into two parts $<$ and $\geq$ will need more run time on identical inputs (or sorted arrays). While no swaps will necessarily occur, it will cause n recursive calls to be made and each of which need to make a comparison with the pivot and n-recursion elements i.e., $O(n^2)$ comparisons need to be made.

5. (Sumerians' multiplication algorithm) The clay tablets discovered in Sumer led some scholars to conjecture that ancient Sumerians performed multiplication by reduction to *squaring*, using an identity like

$$x \cdot y = (x^2 + y^2 - (x-y)^2)/2\,.$$

In this problem, we will investigate how to actually square large numbers.

(a) (7 points) Describe a variant of Karatsubas algorithm that squares any n-digit number in $O(n^{\log 3})$ time, by reducing to squaring three $\lceil n/2 \rceil$-digit numbers. (Karatsuba actually did this in 1960.)

(b) (8 points) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 6})$ time, by reducing to squaring six $\lceil n/3 \rceil$-digit numbers.

(c) (10 points (bonus)) Describe a recursive algorithm that squares any n-digit number in $O(n^{log_3 5})$ time, by reducing to squaring only five $(n/3 + O(1))$-digit numbers. [Hint: What is $(a+b+c)^2 + (a-b+c)^2$?]

Solution:

(a) We know that any number given base B can be written as

$$a = a_1 B^m + a_0,$$

Here we are dealing with binary integers and if the size of the given array is n then we can write

$$a = a_1 2^{n/2} + a_0,$$
$$a^2 = a_1^2 2^n + a_1 a_0 2^{\frac{n}{2}+1} + a_0^2.$$

Here a_1 and a_0 are $\lceil n/2 \rceil$ numbers. We can simplify $a_1 a_0$ using the identity given in the question to,

$$a_1 a_0 = \frac{a_1^2 + a_0^2 - (a_1 - a_0)^2}{2}.$$

If we assume $a_1 = x$, $a_0 = y$ and $a_1 - a_0 = z$, then

$$a^2 = x^2 2^n + (x^2 + y^2 - z^2) 2^{n/2} + y^2.$$

Hence we only need square of three numbers of size $\lceil n/2 \rceil$ to square a number of size n. Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to half i.e., $O(1)$. When we square a number of size $\lceil n/2 \rceil$ the size will be at most n hence at each step combining will take $O(n)$ time. At each step we divide the problem to 3 sub problems which are half the size of the original problem (neglecting the ceiling). Final equation will be

$$T(n) = 3T(n/2) + O(n)$$

Using master theorem we get $T(n) = O(n^{\log 3})$.

(b) Using the same logic mentioned we can write a number of size n as sum of three $\lceil n/3 \rceil$ as follows,

$$a = a_2 2^{2n/3} + a_1 2^{n/3} + a_0,$$

here a_2, a_1, a_0 are size of at most $\lceil n/3 \rceil$. After expand a^2 and simplifying we get,

$$a^2 = a_2^2 2^{4n/3} + a_1^2 2^{2n/3} + a_0^2 + a_1 a_2 2^{n+1} + a_1 a_0 2^{\frac{n}{3}+1} + a_2 a_0 2^{\frac{2n}{3}+1}.$$

If we assume $a_0 = x$, $a_1 = y$, $a_2 = z$, $a_1 - a_0 = p$, $a_2 - a_0 = q$, and $a_2 - a_1 = r$. Then using identity given in the question we can simplify $a_1 a_0$, $a_2 a_0$, and $a_1 a_2$ as

$$\begin{aligned} xy &= \frac{x^2 + y^2 - p^2}{2} \\ zx &= \frac{x^2 + z^2 - q^2}{2} \\ zy &= \frac{z^2 + y^2 - r^2}{2}. \end{aligned}$$

Now we can write a^2 as,

$$\begin{aligned} a^2 = z^2 2^{4n/3} + y^2 2^{2n/3} + x^2 + (z^2 + y^2 - r^2) 2^n \\ + (x^2 + y^2 - p^2) 2^{n/3} + (z^2 + x^2 - q^2) 2^{2n/3}. \end{aligned}$$

Hence we only need square of six numbers of size $\lceil n/3 \rceil$ to square a number of size n. Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to one thirds i.e., $O(1)$. When we square a number of size $\lceil n/3 \rceil$ the size will be at most n hence at each step combining will take $O(n)$ time. At each step we divide the problem to 6 sub problems which are $1/3^{rd}$ the size of the original problem (neglecting the ceiling). Final equation will be

$$T(n) = 6T(n/3) + O(n)$$

Using master theorem we get $T(n) = O(n^{\log_3 6})$.

(c) <u>**Toom - Cook multiplication:** Here we will use the technique which was introduced by Toom and Cook</u> in their algorithm for multiplication (source). Assume we have been given a digit of size n and it is in base 2 and we split it into three parts each of size $\lceil n/3 \rceil$ and the digits will be shifted according $\lfloor n/3 \rfloor$. Now we can write the number as follows,

$$a = a_2 2^{2n/3} + a_1 2^{n/3} + a_0.$$

If we square the given number it will be,

$$\begin{aligned} a^2 &= a_2^2 2^{4n/3} + a_1^2 2^{2n/3} + a_0^2 + a_1 a_2 2^{n+1} + a_1 a_0 2^{\frac{n}{3}+1} + a_2 a_0 2^{\frac{2n}{3}+1} \\ &= a_2^2 2^{4n/3} + (2a_1 a_2) 2^n + (a_1^2 + 2a_2 a_0) 2^{\frac{2n}{3}} + (2a_1 a_0) 2^{\frac{n}{3}} + a_0^2. \end{aligned}$$

Now we will write the given number as terms of a polynomial in x i.e.,

$$p(x) = a_2 x^2 + a_1 x + a_0$$

where if we substitute $x = 2^{n/3}$ we get the number back,

$$p(2^{n/3}) = a = a_2 2^{2n/3} + a_1 2^{n/3} + a_0.$$

We know from algebra that to compute the coefficients of a polynomial of degree d we need to $d+1$ points. Lets take small numbers and substitute them into the polynomial to get,

$$\begin{aligned}
p(0) &= a_0 \\
p(1) &= a_2 + a_1 + a_0 \\
p(-1) &= a_2 - a_1 + a_0 \\
p(2) &= 4a_2 + 2a_1 + a_0 \\
p(-2) &= 4a_2 - 2a_1 + a_0.
\end{aligned}$$

Let $r(x)$ be polynomial such that $r(x) = p(x) * p(x)$, then

$$\begin{aligned}
r(0) &= p(0)p(0) \\
&= a_0^2
\end{aligned}$$

$$\begin{aligned}
r(1) &= p(1)p(1) \\
&= (a_2 + a_1 + a_0)^2 \\
&= a_2^2 + a_1^2 + a_0^2 + 2a_1a_2 + 2a_2a_0 + 2a_1a_0.
\end{aligned}$$

$$\begin{aligned}
r(-1) &= p(-1)p(-1) \\
&= (a_2 - a_1 + a_0)^2 \\
&= a_2^2 + a_1^2 + a_0^2 - 2a_1a_2 + 2a_2a_0 - 2a_1a_0
\end{aligned}$$

$$\begin{aligned}
r(2) &= p(2)p(2) \\
&= (4a_2 + 2a_1 + a_0)^2 \\
&= 16a_2^2 + 4a_1^2 + a_0^2 + 16a_1a_2 + 2a_2a_0 + 4a_1a_0
\end{aligned}$$

$$\begin{aligned}
r(-2) &= p(-2)p(-2) \\
&= (4a_2 - 2a_1 + a_0)^2 \\
&= 16a_2^2 + 4a_1^2 + a_0^2 - 16a_1a_2 + 2a_2a_0 - 4a_1a_0.
\end{aligned}$$

If we assume that the coefficients of the polynomial $r(x)$ as r_0, r_1, r_2, r_3, r_4, then we can write the following relation

$$r(x) = r_4x^4 + r_3x^3 + r_2x^2 + r_1x + r_0$$

$$\begin{bmatrix} r(0) \\ r(1) \\ r(2) \\ r(3) \\ r(4) \end{bmatrix} = \begin{bmatrix} 0^4 & 0^3 & 0^2 & 0^1 & 0^0 \\ 1^4 & 1^3 & 1^2 & 1^1 & 1^0 \\ -1^4 & -1^3 & -1^2 & -1^1 & -1^0 \\ 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ -2^4 & -2^3 & -2^2 & -2^1 & -2^0 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix}.$$

We can invert the matrix and compute the coefficients of the polynomial which gives us

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} = \begin{bmatrix} 1/4 & -1/6 & -1/6 & 1/24 & 1/24 \\ 0 & -1/6 & 1/6 & 1/12 & -1/12 \\ -5/4 & 2/3 & 2/3 & -1/24 & -1/24 \\ 0 & 2/3 & -2/3 & -1/12 & 1/12 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r(0) \\ r(1) \\ r(2) \\ r(3) \\ r(4) \end{bmatrix}.$$

Now we can solve for the coefficients for the polynomial $r(x)$. Instead of expanding everything till the end we can assume that the five $O(n/3) + 1$ digit squares that we are looking for are $r(0), r(1), r(2), r(3), r(4)$, then

$$\begin{aligned} r_0 &= \frac{6r(0) - 4r(1) - 4r(2) + r(3) + r(4)}{24} \\ r_1 &= \frac{-2r(1) + 2r(2) + r(3) - r(4)}{12} \\ r_2 &= \frac{-30r(0) + 16r(1) + 16r(2) - r(3) - r(4)}{24} \\ r_3 &= \frac{8r(1) - 8r(2) - r(3) + r(4)}{12} \\ r_4 &= r(0). \end{aligned}$$

If we substitute $x = 2^{n/3}$ in $r(x)$ we get,

$$r(2^{n/3}) = r_4 2^{4n/3} + r_3 2^n + r_2 2^{2n/3} + r_1 2^{n/3} + r_0,$$

this is nothing but a^2. If we compare the equations of a^2 and $r(2^{n/3})$ we get,

$$\begin{aligned} r_4 &= a_2^2 \\ r_3 &= 2a_1a_2 \\ r_2 &= a_1^2 + 2a_2a_0 \\ r_1 &= 2a_1a_0 \\ r_0 &= a_0^2. \end{aligned}$$

Now we can write a^2 in terms of the five squares,

$$\begin{aligned}
a_2^2 &= r(0) \\
a_1 a_2 &= \frac{8r(1) - 8r(2) - r(3) + r(4)}{24} \\
a_1^2 + 2a_2 a_0 &= \frac{-30r(0) + 16r(1) + 16r(2) - r(3) - r(4)}{24} \\
a_1 a_0 &= \frac{-2r(1) + 2r(2) + r(3) - r(4)}{24} \\
a_0^2 &= \frac{6r(0) - 4r(1) - 4r(2) + r(3) + r(4)}{24}.
\end{aligned}$$

This technique of splitting terms and finding the coefficients of the squared number is very powerful because this can be used to decrease the computation of powers not only to 5 but even below. But linear run time is not achievable.

We have used to five $O(n/3) + 1$ squared terms $r(0), r(1), r(2), r(3), r(4)$ to compute the square of a number a of size n. Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to one thirds i.e., $O(1)$. When we square a number of size $\lceil n/3 \rceil$ the size will be at most n hence at each step combining will take $O(n)$ time. At each step we divide the problem to 5 sub problems which are $1/3^{rd}$ the size of the original problem (neglecting the ceiling). Final equation will be

$$T(n) = 5T(n/3) + O(n)$$

Using master theorem we get $T(n) = O(n^{\log_3 5})$.