CS 344 Homework One

Matthew McCaughan

Due February 13th, 2025

Problem 1: Academic Integrity

I affirm that I have not given or received any unauthorized help on this assignment, and that this work is my own.
- Matthew McCaughan

Problem 2: Sorting Faster Sometimes

Within the word-RAM model, reading or writing to a cell as well as most arithmetic operations, can be performed in constant time. This is important, as these extra tools can allow us to sort without the dependence on expensive comparisons between numbers. In order to sort this list of numbers that runs in time O(n), we have to circumvent comparison all together! Luckily, there exists methods to sort such lists without explicit comparison between elements in the list. A good contender for sorting this list in $O(n)$ would be Radixsort. With Radixsort, we're foregoing comparison queries and reading the literal digits in each numbers' place for the values, the values associated with these digits are placed into buckets with the associated number. This process is iterated through each place in the values in the array until completion. When this is completed, each bucket will have the values sorted and each successive bucket is sorted relative to eachother, merging the buckets as queues will form a sorted array. The runtime for Radixsort is $O(n * d)$, where d is the actual number of digits in the value with the most digits to check. For each number place until the largest, we perform linear work to place each value into a bucket. If $\max_i x_i \leq 1000n$ for a list $x_1, ..., x_n$, the approximate run time would be $O(n * (log(1000n) + 1))$ which, as a multiple of n, is bounded by $O(n)$.

Problem 3: Karatsuba into 3

Given x and y represented as:

$$x = x_2 \cdot 10^{2n/3} + x_1 \cdot 10^{n/3} + x_0$$

$$y = y_2 \cdot 10^{2n/3} + y_1 \cdot 10^{n/3} + y_0$$

We can expand the product to get a better idea of the naive approach to performing this computation:

$$x \cdot y = (x_2 \cdot 10^{2n/3} + x_1 \cdot 10^{n/3} + x_0) \cdot (y_2 \cdot 10^{2n/3} + y_1 \cdot 10^{n/3} + y_0)$$

$$x \cdot y = x_2y_2 \cdot 10^{4n/3} + (x_2y_1 + x_1y_2) \cdot 10^{3n/3} + (x_2y_0 + x_1y_1 + x_0y_2) \cdot 10^{2n/3}$$

$$+(x_1y_0 + x_0y_1) \cdot 10^{n/3} + x_0y_0$$

This naive approach requires computing NINE multiplications! This is not good for a faster algorithm. We want to beat Karatsuba, so lets use what has been defined in the problem. Given our generous hint, these defined z variables will be used as a function of each x component and each y component. Without being too tedious and for some brevity, w_1 and w_5 follow quite simply with z_1 and z_5 and describe small, identifiable terms at each end of the expanded $x * y$ product, using these bookends I can see that each z_i represents a component of the expanded x and y terms in the expanded product with decreasing powers of 10, we can construct this expanded product in terms of z.

$$x \cdot y = z_5 \cdot 10^{4n/3} + z_4 \cdot 10^{3n/3} + z_3 \cdot 10^{2n/3} + z_2 \cdot 10^{n/3} + z_1$$

We have five z terms, and each z term is made up of the addition or subtract of w terms, and each w term requires only 1 multiplication, so overall we reduce the number of multiplications by 4, leaving with only 5 multiplications to perform with this method of breaking up the terms. We've reduced the problem from 9 to 5 calls, and each breaking up the problem by a factor of 3, and for each call, we're performing $O(n)$ work. We can form our recurrence formula based on this information:

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

Like original Karatsuba, we can evaluate the performance of this improvement using the master theorem with $a = 5, b = 3, k = 1; r = 5/3^1$. Since $r > 1$, the master theorem evaluates this recurrence to:

$$T(n) = O(n^{log3(5)}) \approx O(n^{1.47)}$$

This beats the original Karatsuba runtime of $O(n^{1.585})$. Our improved Karatsuba has a runtime of $O(n^{1.47})$.

Problem 4: Polynomials

(a)

We must construct an algorithm for this problem that runs in time $O(n^{log_2 3})$. We can work backwards from our recurrence relation based on the master theorem. A reasonable recurrence that would result in this runtime could be:

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

Based on the similarity to the Karatsuba recurrence, and given that polynomials are of degree $n = 2^k$ for some positive integer k (also important for Karatsuba), we can similarly break these to polynomials up like we do for Karatsuba. Like how we split terms x and y into two components for regular Karatsuba, we are splitting each polynomial P_i into P_{iL}, which will contain the lower terms below degree $x^{n/2}$ and P_{iR}, which will contain the higher terms from degree $x^{n/2}$. We can rewrite polynomial $P_i(x)$ into:

$$P_i(x) = P_{iL}(x) + (x^{x/2}) * P_{iR}$$

Our original multiplication $P_1(x) \cdot P_2(x)$ can be rewritten as:

$$P_L = P_{1L}(x)P_{2L}(x)$$

$$P_R = P_{1R}(x)P_{2R}(x)$$

We can define our product of the polynomial components as P_m:

$$P_M = (P_{1L}(x) + P_{1R}(x)) \cdot (P_{2L}(x) + P_{2R}(x))$$

Expanding this:

$$P_M = P_{1L}(x)P_{2L}(x) + P_{1L}(x)P_{2R}(x) + P_{1R}(x)P_{2L}(x) + P_{1R}(x)P_{2R}(x)$$

Since we already have $P_L = P_{1L}(x)P_{2L}(x)$ and $P_R = P_{1R}(x)P_{2R}(x)$, we can compute the missing terms like how we do for original Karatsuba:

$$P_M - P_L - P_R = P_{1L}(x)P_{2R}(x) + P_{1R}(x)P_{2L}(x)$$

Putting it all together:

$$P(x) = P_L + x^{n/2}(P_M - P_L - P_R) + x^n P_R$$

Doing this manipulation of our input, we reduce our recursive multiplication calls from 4 to 3, and reduce each problem by a factor of 2 each recursive call, and perform linear work. We can form a recurrence relation based on these components:

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

Utilizing the master theorem, with $a = 3, b = 2, k = 1; r = 3/2^1$, the recurrence relation computes to:

$$T(n) = O(n^{log2(3)}) \approx O(n^{1.585})$$

Strategically breaking up our polynomials based on Karatsuba multiplication allows us to calculate their product in time $O(n^{1.585})$.

(b)

I will only explain the process, rather than explicitly write it all out (potentially to the detriment of some of the bonus). Using a similar method to problem three and breaking Karatsuba into 3 parts, we can use the same process to break each polynomial into three components based on degrees divisible by 3 with the lower, middle, and higher degree components. Following the process in problem 3, we reduce the problem from 9 to 5 recursive multiplication calls, and each breaking up the problem by a factor of 3, and for each call, we're performing $O(n)$ work. Using the master theorem again, we'll get a recurrence relation of:

$$T(n) = O(n^{log3(5)}) \approx O(n^{1.47)})$$

This runtime beats the two part Karatsuba, with runtime $O(n^{1.47)}) < O(n^{1.585})$.

Problem 5: Save my hard-drive

(a)

An algorithm is possible to identify the block containing the special bit by identifying the parity of each sequence of either 1s or 0s as we move through the list. Starting at index 0, we increment a counter for every of the same number (1 or 0) we see. When the value at the next index does not equal the value at the current index, we will check if the incremented counter modulo 2 is 0.

```
**SpecialBit(List L)**
incrementer $c = 0$
**if** value at current index == value at next index **then**
    $c \leftarrow c + 1$
    increment index
**else**
    **if** value at current index != value at next index **then**
        **if** c mod 2 == 0 **then**
            continue
            **if** c mod 2 != 0 **then**
                return index
            **end if** we reach index n-1
```

Using this outline for an algorithm, the special bit will be identified by the end of iterating through the list. Since we will only be interating through at most the length of the list, and at each index, we are doing constant work, this algorithm will run for at most $o(n)$.

(b)

Such an algorithm could not be found to rune within $o(n)$ for all inputs. Assuming there exists some algorithm that can complete this, for every index, there must be a comparison between the value at an index in the repeated pattern and the value at an index in the array, and there also must be a comparison to compare where in the repeated sequence of values we are in, to make sure we are following the pattern. This will require at least 2 comparisons per item in the array, and $2n$ exceeds the bound imposed by $o(n)$, so an algorithm could not perform this task strictly within $o(n)$ time for all inputs.

Problem 6: Tweaking Median of Medians

(a)

Running the median of medians algorithm with blocks of size 3 results in a recurrence formula of $T(n) = T(n/3) + T(2n/3) + O(n)$. The pivot, when calculated, allows us to remove $(1/2 * n/3) * 2 = n/3$ of the values, resulting in a subproblem of $T(2n/3)$ Because although we recursively split the problem into sub-problems of $T(n/3)$ and $T(2n/3)$, this results in linear work for each recursive layer we access, and a reduction of the problem size by a maximum of 3. Resulting in an asymptotic run time of $O(n * log_3 n)$.

(b)

Running the median of medians algorithm with blocks of size 7 results in a recurrence formula of $T(n) = T(n/7) + T(2n/3) + O(n)$. The pivot, when calculated, allows us to remove $(1/2*n/7)*4 = 4n/14$ of the values, resulting in a subproblem of $T(10n/14)$ Because although we recursively split the problem into sub-problems of $T(n/7)$ and $T(10n/14)$, this results in close to linear work for each recursive layer we access, and a reduction of the problem size by a maximum of 7. This results in an asymptotic run time of $O(n * log_7 n)$.