Gabriel Arellano
Advanced Algorithms – Fall 2014
Programming project part III - Report

COMPLETING PARTS I AND II

In part I, I implemented an algorithm for performing Discrete Fourier Transformation (DFT) which involved taking in coefficients for a polynomial of a length equal to some power of 2. Using this list of coefficients, DFTs create point values for the provided polynomial. An inverse function was also implemented, as well as a function for multiplying polynomials. I ran several tests, on input from size 2^2 to 2^{10}, observing the execution time of the DFT function for each input size. From these values (and Excels handy data extrapolation features), I found the following formula for determining the running time of my DFT algorithm based on n as a power of 2 (so $n = \log_2(\text{input size})$):

$$an^2 + bn + c = .0509n^2 - .4771n + .9356$$

In part II, I implemented the recursive Fast Fourier Transformation (FFT) as it is defined in our class textbook, Introduction to Algorithms (pg 911). I also had to implement the recursive version of the inverse FFT. This part of the project only required me to implement the functions, without doing any timing experiments.

The remainder of this report contains information on the three major tasks for part III of this project and a conclusion. The three main tasks I identified were:

1. Determining the time for an individual recursive FFT call without including the time during recursive calls (so only the time spent within the current function's scope, and not nested calls).
2. Calculating an optimal threshold for switching between my DFT and FFT functions, based on optimizing the running time of the recursive function.
3. Experimentally verify that the optimal threshold from the second task is correct.

TASK 1 – DETERMINING THE RUNNING TIME OF THE RECURSIVE FFT

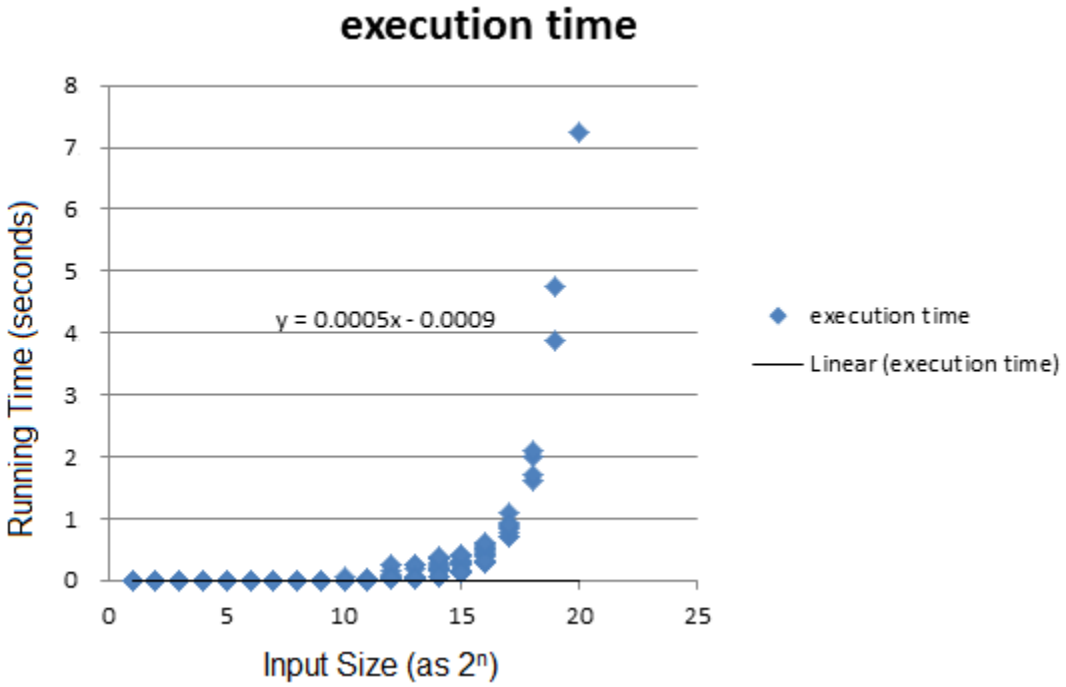
To complete this task, I modified my FFT method to save five local variables for each execution of the recursive FFT function. The recursive FFT function involves some locally scoped work, some recursive calls, followed by more locally scoped work. The five variables saved were (1) the start time of the function, (2) the time right before calling the recursive calls in the middle of the function, (3) the time after returning from recursive calls, (4) the time at the end of the recursive FFT function (i.e. before returning), and (5) the input size (as an integer that is equivalent to the log of the input size). These 5 variables were stored to a dictionary which is eventually flushed/printed to stdout in human-readable form and a file (as comma-separated values (CSV)).

After collecting the data, I then used the CSV file I created to load an Excel spreadsheet and extrapolate the value for the running time of the algorithm without recursions. The recursive FFT function I created has a running time described by the following recurrence equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

From the data, collected, I was to calculate the function implied asymptotically by $\theta(n)$. Using the data in Excel (graph printed below), I calculated this function to be:

$$.0005n - .0009 \in \theta(n)$$



Please note that the black line along the x-axis from x = 0 to 20 is the trend line fitted to the data. I'm not sure why Excel chose to render the line that way, and messing with available options yielded no visual changes.

TASK 2 – CALCULATING AN OPTIMAL THRESHOLD

Using the data collected from the first task, I could now theoretically determine the optimal threshold for switching between my DFT and FFT functions. To do this, I used the observation that the optimal threshold would be the point where the recursive algorithm only has to switch to the DFT algorithm once (i.e. the input would be divided into two halves, and the recursive call to solve each half would only use the DFT to solve it, requiring no further nested calls. Thus, in the recurrence equation above, $T\left(\frac{n}{2}\right)$ becomes $a\left(\frac{n}{2}\right)^2 + b\left(\frac{n}{2}\right) + c$. The threshold exists where this instance of the recurrence equation is equal to the timing equation found in part I of the project. Thus we arrive at the following:

$$2T\left(\frac{n}{2}\right) + \theta(n) = an^2 + bn + c$$

$$2\left(a\left(\frac{n}{2}\right)^2 + b\left(\frac{n}{2}\right) + c\right) + \theta(n) = an^2 + bn + c$$

$$\frac{an^2}{2} + bn + 2c + dn + e = an^2 + bn + c$$

From part I, and from task 1 above, we know the constants in this equation are:

$$a = .0509 \quad b = -.4771 \quad c = .9356$$
$$d = .0005 \quad e = -.0009$$

I am omitting leading zeroes (0.0509 vs .0509) for the sake of page space. By plugging and solving:

$$\frac{.0509n^2}{2} - .4771n + 1.8712 + .0005n - .0009 = .0509n^2 - .4771n + .9356$$
$$.02545n^2 - .4766n + 1.8703 = .0509n^2 - .4771n + .9356$$
$$.02545n^2 - .0005n - .9347 = 0$$

$$n = -b \pm \frac{\sqrt{b^2 - 4ac}}{2a} = \frac{-(-.0005) \pm \sqrt{(-.0005)^2 - 4(.02545)(-.9347)}}{2(.02545)}$$
$$= \frac{.0005 \pm \sqrt{2.5 \times 10^{-7} + .09515271}}{.0509} = \frac{.0005 \pm .308468}{.0509}$$
$$n = 6.07 \; or \; -6.05$$

Here, n is represented as a power of two, or as the $\log_2$(input size), so the threshold exists at an input size of 2^n. Thus, my theoretical threshold would require me to use the DFT algorithm for inputs of size 2^6 (64) and below, while the recursive FFT algorithm would theoretically be faster at inputs above this size (2^7 and up).

TASK 3 – EXPERIMENTALLY VALIDATING THE OPTIMAL THRESHOLD

To experimentally validate my theoretical threshold, I timed my optimized Fourier Transformation (FT) code, which involves a wrapper method for both the DFT and FFT functions. When the input size is below or at the threshold, the DFT function is called, otherwise the recursive FFT function is called. Furthermore, the recursive FFT function was modified to call this optimal FT method instead of directly making recursive calls to the recursive FFT function. I implemented the threshold as a global variable so that I could set it for all FT function calls for each test.

First I decided to try running 5 tests, all the same but with different input. The tests consisted of first creating a list of coefficients with a pseudo-random number generator within Python of length $2^n = 2^{20}$. Then this random list would use my optimalFT() function with different thresholds to solve the same input. For each list created, I would use each power of 2, from 2^3 to 2^{n-1} as a threshold to solve the same input. After taking ~5.5 hours to complete a FT on input size 2^{20} with a threshold at 2^{14}, my laptop mysteriously shut down; it took about 8-10 hours total for the test to reach this point. The reason the results end is unknown, as I let the experiments run independently without supervision. When I found the laptop off, I immediately assumed the battery ran out of power, but upon restarting the computer without the charging cable, I saw that the battery was full! My best guess is that the CPU in my laptop, which is usually VERY hot normally, overheated and caused the laptop to shut down. Thus, my experiment was cut off early. Fortunately, my code was writing results off to a file, and this file was not corrupted by the early shut down.

Next, I figured that I should avoid running my experiment at thresholds above 2^{14}; thresholds this high were yielding large execution times consistently, so I assume that larger thresholds will only take longer than 10 hours. I ran the same tests as before, but prevented the

thresholds above 2^{14} to be tested. As an extra precaution, I turned my laptop on its side like an open book standing vertically so that the CPU would be at the top and allow heat to dissipate into the surrounding air. This test, likewise, cut off early, after running for some 10-12 hours completing almost 2 tests. The computer seems to have overheated during the second test, while testing the threshold at 2^{14}. I decided to stop testing large thresholds, as the salvaged results consistently had approximately the same execution times – for thresholds above 2^{10}, execution times are consistently within the same range of thousands of seconds +/- 30 seconds or so. Also, between each threshold 2^x and 2^{x+1} for $x > 10$, the time seems to double consistently for each step. These consistencies lead me to believe that, generally, my timing results for thresholds above 2^{10} elicit high precision.

Finally, I decided to only test thresholds from 2^3 to 2^{10} for an input size of 2^{20}, which successfully tested the thresholds for five different randomly-generated inputs. Again, these results are consistent with the results from the previous experiments. The results indicated an interesting pattern between the difference in execution times when changing the threshold. When the threshold is below 2^7, going to the next threshold value increases the execution time with less than double the time, so $T(2^x) < 1.5 * T(2^{x-1})$ for these lower thresholds. Above 2^6, an increase in threshold approaches doubling the running time, so $T(2^x) > 1.5 * T(2^{x-1})$ for these larger thresholds; $T(2^x)$ approaches $2*T(2^{x-1})$ as the threshold got larger. Stated differently, thresholds lower than my theoretical threshold do not differ in execution time significantly (by a factor <1.5), while thresholds above my theoretical threshold differ in execution time by a factor greater than 1.5.

Another observation I noticed about the execution times was that with the threshold at 2^3, execution time was about 100 seconds to 2 minutes. The execution time for the threshold at 2^6 or 2^7 was generally about 1 or 2 minutes more (so 4 minutes total). Execution times for thresholds higher than 2^7 were 5 minutes more than the execution time for the threshold at 2^3. Thus, these higher thresholds increase execution time much more drastically, making them plainly non-ideal and therefore non-optimal.

This argument above, however, is highly nuanced and I am being very generic. Similar arguments could be said for thresholds near 2^6, so given a more rigorous experiment, the actual threshold could be something besides 6, +/- 1 or so.

CONCLUSION

To demonstrate my conclusion, I'd like to share one of my favorite jokes about statistics:

> *Three professors (a physicist, a chemist, and a statistician) are called in to see their dean. Just as they arrive the dean is called out of his office, leaving the three professors there. The professors see with alarm that there is a fire in the wastebasket.*
>
> *The physicist says, "I know what to do! We must cool down the materials until their temperature is lower than the ignition temperature and then the fire will go out."*
>
> *The chemist says, "No! No! I know what to do! We must cut off the supply of oxygen so that the fire will go out due to lack of one of the reactants."*

While the physicist and chemist debate what course to take, they both are alarmed to see the statistician running around the room starting other fires. They both scream, "What are you doing?"
To which the statistician replies, "Trying to get an adequate sample size."

In conclusion, without further testing, my implementations of the Fourier Transformations in Python lead me to a theoretical optimal threshold of 2^6 to switch between my DFT and recursive FFT functions. Through experimentation, this was shown to be an opportune threshold, as using higher thresholds creates large increases in execution time, and smaller thresholds only perform slightly better. I will note, however, that as the threshold decreased below this theoretical optimal value, the execution time went down. This leads me to also conclude that running the recursive algorithm may just be faster in all cases.

If you would like to review any portion of my code or data, please ask.