CMPS 2200 Assignment 2

Name: Viraj Choksi

In this assignment we'll work on applying the methods we've learned to analyze recurrences, and also see their behavior in practice. As with previous assignments, some of of your answers will go in `main.py` and `test_main.py`. You should feel free to edit this file with your answers; for handwritten work please scan your work and submit a PDF titled `assignment-02.pdf` and push to your github repository.

Part 1. Asymptotic Analysis

Derive asymptotic upper bounds of work for each recurrence below.

- $W(n) = 2W(n/3) + 1$.

  $a=2,\ b=3,\ f(n)=O(1)$

  $c=0 < \log_3 2$ (case 1)

  Complexity $= O(n^{\log_3 2})$

- $W(n) = 5W(n/4) + n$.

  $a=5,\ b=4,\ f(n)=O(n)$

  $c=1 < \log_4 5$ (case 1)

  complexity $= O(n^{\log_4 5})$

- $W(n) = 7W(n/7) + n$.

  $a=7,\ b=7,\ f(n)=O(n)$

  $c=1=\log_7 7$ (case 2)

  Complexity $= O(n \log n)$

- $W(n) = 9W(n/3) + n^2$.

  $a=9,\ b=3,\ f(n)=O(n^2)$

  $c=2=\log_3 9$ (case 2)

  Complexity $= O(n^2 \log n)$

- $W(n) = 8W(n/2) + n^3$.

  $a=8,\ b=2,\ f(n)=O(n^3)$

  $c=3=\log_2 8$ (case 2)

  Complexity $= O(n^3 \log n)$

- $W(n) = 49W(n/25) + n^{3/2} \log n$.

$a = 49$, $b = 25$, $f(n) = O(n^{3/2} \log n)$

$c = 3/2 > \log_{25} 49$ (case 3)

Complexity $= O(n^{3/2} \log n)$

- $W(n) = W(n-1) + 2$.

Expand: $W(1) = c$

$W(n-1) = W(n-2) + 2$ $\quad W(n) = c + \sum_{k=1}^{n} 2$

$W(n-2) = W(n-3) + 2$ $\quad W(n) = c + 2n$ // $W(n) = O(n)$ $\therefore$ Complexity $= O(n)$

- $W(n) = W(n-1) + n^c$, with $c \geq 1$.

Expand: $W(1) = c$

$W(n-1) = W(n-2) + (n-1)^c$ $\quad W(n) = c + \sum_{k=1}^{n} k^c$

$W(n-2) = W(n-3) + (n-2)^c$ $\quad$ Complexity $= O(n^{c+1})$

- $W(n) = W(\sqrt{n}) + 1$.

$W(\sqrt{n}) = W(\sqrt{\sqrt{n}}) + 1$

Depth of recursion is $\log \log n$

Complexity $= O(\log \log n)$

Part 2. Algorithm Comparison

Suppose that for a given task you are choosing between the following three algorithms:

- Algorithm $\mathcal{A}$ solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm $\mathcal{B}$ solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
- Algorithm $\mathcal{C}$ solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

  What are the asymptotic running times of each of these algorithms? Which algorithm would you choose?

Algorithm A: $T(n) = 5T(n/2) + O(n)$

$a = 5$ $\quad b = 2$ $\quad f(n) = O(n)$

$c = 1 < \log_2 5$ (case 1)

Complexity $= O(n^{\log_2 5})$

Algorithm B: $T(n) = 2T(n-1) + O(1)$

Expand: $O(2^n)$

Complexity $= O(2^n)$

Algorithm C: $T(n) = 9T(n/3) + O(n^2)$

$a = 9 \quad b = 3 \quad f(n) = O(n^2)$

$c = 2 = \log_3 9$ (case 2)

complexity $= O(n^2 \log n)$

As seen by the complexity of all these algorithms, the algorithm that I would choose is algorithm A. This is because for a sufficiently large n, A has the lowest complexity, making it the best algorithm.

.
.
. .

Part 3: Parenthesis Matching

A common task of compilers is to ensure that parentheses are matched. That is, each open parenthesis is followed at some point by a closed parenthesis. Furthermore, a closed parenthesis can only appear if there is a corresponding open parenthesis before it. So, the following are valid:

- `( ( a ) b )`
- `a () b ( c ( d ) )`

but these are invalid:

- `( ( a )`
- `(a ) ) b (`

Below, we'll solve this problem three different ways, using iterate, scan, and divide and conquer.

3a. iterative solution Implement `parens_match_iterative`, a solution to this problem using the `iterate` function. **Hint**: consider using a single counter variable to keep track of whether there are more open or closed parentheses. How can you update this value while iterating from left to right through the input? What must be true of this value at each step for the parentheses to be matched? To complete this, complete the `parens_update` function and the `parens_match_iterative` function. The `parens_update` function will be called in combination with `iterate` inside `parens_match_iterative`. Test your implementation with `test_parens_match_iterative`.

.
.

3b. What are the recurrences for the Work and Span of this solution? What are their Big Oh solutions?

enter answer here

. Work: $O(n)$ Big Oh solutions: Optimal at $O(n)$ since every element must be examined at least 1 time (Span cannot be improved)
. Span: $O(n)$

3c. scan solution Implement `parens_match_scan` a solution to this problem using `scan`. **Hint**: We have given you the function `paren_map` which maps `(` to `1`, `)` to `-1` and everything else to `0`. How can you pass this function to `scan` to solve the problem? You may also find the `min_f` function useful here. Implement `parens_match_scan` and test with `test_parens_match_scan`

.
.

3d. Assume that any `maps` are done in parallel, and that we use the efficient implementation of `scan` from class. What are the recurrences for the Work and Span of this solution?

enter answer here

- Work : O(n) for map + O(n) for scan + O(n) for reduce = O(n)
- Span: O(log n) => scan is in logarithmic time.

3e. divide and conquer solution Implement `parens_match_dc_helper`, a divide and conquer solution to the problem. A key observation is that we *cannot* simply solve each subproblem using the above solutions and combine the results. E.g., consider '((()))', which would be split into '(((' and ')))', neither of which is matched. Yet, the whole input is matched. Instead, we'll have to keep track of two numbers: the number of unmatched right parentheses (R), and the number of unmatched left parentheses (L). `parens_match_dc_helper` returns a tuple (R,L). So, if the input is just '(', then `parens_match_dc_helper` returns (0,1), indicating that there is 1 unmatched left parens and 0 unmatched right parens. Analogously, if the input is just ')', then the result should be (1,0). The main difficulty is deciding how to merge the returned values for the two recursive calls. E.g., if (i,j) is the result for the left half of the list, and (k,l) is the output of the right half of the list, how can we compute the proper return value (R,L) using only i,j,k,l? Try a few example inputs to guide your solution, then test with `test_parens_match_dc_helper`.

.

.

3f. Assuming any recursive calls are done in parallel, what are the recurrences for the Work and Span of this solution? What are their Big Oh solutions?

enter answer here

- Work : O(n) => splitting and merging
- Span : O(log n) => parallel execution

Big Oh solutions: Work is optimal at O(n) since every element must be examined. Span is optimal at O(log n) since it is a divide and conquer algorithm with constant time steps.