Written Problem Set 2: Divide and Conquer

Response by: **Tianze Ren(tr2bx), Ziang Zhang(zz9fs), Mingye Li(fky9xf), Ziyue Xu(rca8vx))**

Collaborators: Tianze Ren(tr2bx), Ziang Zhang(zz9fs), Mingye Li(fky9xf), Ziyue Xu(rca8vx)

Resources: NONE

Problem 1: Flights

You have been hired to plan the flights for Professor Floryan's brand new passenger air company, "Receding Airlines". You are going to provide service to n cities. This airline will only service the United States, and will only fly you North.

You recognize that in order to enable all your passengers to travel from any city to any other city (to the North) with a single flight requires $\Omega(n^2)$ different routes. Prof. Floryan says that the airline cannot be profitable when supporting so many routes. Another option would be to order the cities in a list (from South to North), and have flights that go from the city at index i, to the city at index $i+1$. This, however, would mean some passengers would require $\Omega(n)$ connections to get to their destination. Devise a compromise set of routes which requires no passenger have more than a single connection (i.e. must take at most two flights), and requires no more than $O(n \log n)$ routes. Justify that it satisfies these requirements.

Parameters Let n denote the number of cities. The Routes function takes in an array list of cities and the last (most northern) city as inputs, and outputs a dictionary of routes.
Notes: the cities start with 1 (most southern) instead of 0 for convenience.

Base Case. Max input size: 2;
Results: returns a route between these two cities. If n-2 is 0.

Divide Step. As the list of n cities is already sorted from south to north (as piazza post suggested), we divide them into an array with the first (n-1) cities in the original array and 1 most northern city.
That is saying, Routes(arr[n],n) = Routes(Routes(arr[n-1],n-1)).

Combine Step. Adding routes according including the consideration of the last city:
0) Add a route between n-1 and n.
1) We first evaluate whether there is a direct flight of the n-3, n-4, ...1 cities to n-1.
2) If the answer to the question is true, then do nothing.
3) If the answer to the question is false, add a direct route from the point to the last city (n) and save it to the routes dictionary.

Algorithm. 1) Modify the list of routes based on the first n-1 cities. Then modify the routes again to add the routes needed to the n cities.

Correctness. 1) First, we know that if the plane could only fly north, we need a route between every i and i+1 cities, this case is considered by this algorithm since when there is no valid route, namely within 1 connection, we add a direct route.
2) Second, if the first n-1 cities have valid routes that allow them to get to northern cities within 1 connection, adding the city number n (most northern) will not affect the validness of those in the first n-1 cities.
3) Third, in our algorithm, we check whether there is a route between cities from n-3 to city 1. This is because city n is newly added, it has no routes directly to it at this point, the only possible way to have a

route to city n is having city n-1 as the only connection. And if there is a direct flight between one city and the n-1 city, we know that it can get to city n within 1 connection. Also, the reason that we did not check for n-2 and n-1 is because since we add a route between the n-1 and n route before this, we know that there requires no connection for people travelling from n-1 to n, and 1 connection from n-2 to n.

Number of Routes. By listing out the algorithm and finding patterns, it is found that the number of routes for n cities in this case is going to be
$NumOfRoutes(n) = NumOfRoutes(n-1) + 1 + floor[\frac{n}{2} - 1]$:
$T(n) = \theta(n) + \theta(1) + \theta(n) = \theta(2n) = \theta(n)$ Because n is greater or equal to 2 for this company to exist, and for every n greater or equal to 2, $\log(n) >= 1$, therefore $n\log(n) >= n$. Which means that it the number of routes is always fewer or equal to $n\log(n)$.

Problem 2a: Fast Exponentiation

Base Case. Maximum input: 1;
if n=1, return a.

Divide Step. 1) If n is even, divide the array in half. The sub-problem created is to calculate $a^{\frac{n}{2}}$.
2) If n is odd, divide the array in half as well. The sub-problem created is to calculate $a^{\frac{n-1}{2}}$.
3) The running time for the worst running time will be $O(\log n)$, because each divide is O(1) and we divide n by a half $\log n$ times.

Combine Step. Multiply the result of function call with $n/2$ as the parameter by two if n is even; Multiply the result of function call with $n/2$ as the parameter by two and then plus a if n is odd.

Algorithm. 1) When n is even: $answer = a^n = (a^{\frac{n}{2}})^2$
1) When n is odd: $answer = a^n = (a^{\frac{n-1}{2}})^2 * a$

Recursively divide n to its half until reaching the base case of $n == 0$ or $n == 1$. At each recursion, multiply the next recursion call by two if n is even, or multiply the next recursion call by two and then plus a if n is odd.

Correctness. I will prove it by induction. When $n == 0$, the function returns 1. When $n == 1$, the function returns a. Assume the function is correct for all $n < k$. At $n == k$, if n is even, the result would be $function(a, k/2) * function(a, k/2)$, which is correct according to the assumption since $k/2 < k$; if n is odd, the result would be $function(a, k/2) * function(a, k/2) + a$, which is correct according to the assumption since $k/2 < k$ and a^n when $n == 1$ is a. In summary, for all n, the algorithm is correct.

Running Time. The conquer and combine steps has O(1) time complexity since there is no comparison made, and both of them are done once, which means that it is still at constant time. The Divide step is also O(1) time complexity, and it is done $\log n$ times, because the n will reach 1 after $\log n$ divisions.

Problem 2b: Fast Matrix Exponentiation

Computer graphics software typically represents points in n dimensions as $(n+1)$-dimensional vectors. To make transformations on the points (e.g. rotating a modelled figure, zooming in, or making the figure appear as though seen through a fish-eye lens) we use a $(n+1) \times (n+1)$ matrix which defines the transformation, and then we multiply each vector by this matrix to transform that point. Let's say we are developing software for very high dimension graphics (n dimensions), and we have a transformation that we would like to apply to a particular point n times. Develop an algorithm which can multiply this $(n+1)$-dimensional point by this $(n+1) \times (n+1)$ transformation matrix n times in $o(n^3)$ (little-oh of n^3) time. In summary, your algorithm should compute $T \times T \times T \times ... \times T \cdot v = T^n \cdot v$.

Algorithm. 1) A $(n+1) \times (n+1)$ matrix times a $(n+1) \times (n+1)$ is going to result in a $(n+1) \times (n+1)$ matrix, so steps 2) and 3) are going to describe the algorithm for multiplying them with a shorter running time.

2) Because the transformation matrices are all identical, we can denote the two matrices as $\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$.

3) Finding AA:

$$\begin{bmatrix} Q_1 + Q_4 - Q_5 + Q_7 & Q_3 + Q_5 \\ Q_2 + Q_4 & Q_1 - Q_2 + Q_3 + Q_6 \end{bmatrix}$$

$$\begin{aligned} Q1 &= A_{1,1}A_{1,1} \\ Q2 &= A_{1,2}A_{2,1} \\ Q3 &= A_{1,2}(A_{1,2} + A_{2,2}) \\ Q4 &= A_{2,1}(A_{1,1} + A_{2,2}) \\ Q5 &= A_{2,2}A_{2,2} \end{aligned}$$

4) Then after computing the T^n, we need to multiply the $(n+1) \times (n+1)$ matrix by the $(n+1) \times 1$ vector. That is going to be (Let A be the first matrix and B be the second matrix):

$$\begin{bmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \end{bmatrix}$$

Running Time. 1) Time analysis for $T \times T$:
Number of Multiplications: 5;
Number of Adds: 10;
$T(n+1) = 5T(\frac{n+1}{2}) + 10(\frac{n+1}{2})^2$ And this matches case 1 since $(n+1)^{\log(5)} > n^2$
$T(n+1) = \theta((n+1)^{\log(5)})$
2) To compute T^n, we need to do $T \times T$ for at least $log(n)$ times. So the time required for T^n is $log(n)T(n+1) = \theta(log(n)(n+1)^{\log(5)})$
3) Time for $(n+1) \times (n+1)$ matrix and the $(n+1)$ vector:
Number of Multiplications: 4;
Number of Adds: 4;

$T(n+1) = 4T(\frac{n+1}{2}) + 4(\frac{n+1}{2})^2$
And this is case two:
So, $T(n+1) = \theta((n+1)^{\log(4)} * \log(n+1))$
4) Total Time Complexity
Total time
=$\theta(log(n)(n+1)^{\log(5)}) + \theta((n+1)^{\log(4)} * \log(n+1))$
$= \theta(log(n)(n+1)^{\log(5)}) + (n+1)^{\log(4)} * \log(n+1))$
For every $n > 16$, $log(n)(n+1)^{\log(5)}) + (n+1)^{\log(4)} * \log(n+1) < c \times n^3$ for any c. Therefore, it is strictly bounded by n^3 for large n.

Problem 3: Iterated Functions

$f(n)$	c	$f_c^*(n)$
$n/2$	1	$\lceil \log_2 n \rceil$
$n-1$	0	n
$n/2$	2	$log_2 n - 1$
$\sqrt{n}$	2	$log_{1/2} log_2 n$
$\sqrt{n}$	1	infinity
$n^{1/3}$	2	$log_{1/3} log_2 n$

Justification: For $f(n) = n-1$ and $c = 0$, we need to find the smallest i such that $f(i)(n) \leq c = 0$, where $f(i)(n) = f(f(i-1)(n))$. In this case, we have:

$$f(1)(n) = n - 1$$
$$f(2)(n) = n - 2$$
$$f(3)(n) = n - 3$$
$$\ldots$$
$$f(i)(n) = n - i$$

So we need to solve for i in the equation $n - i \leq 0$, which gives $i = n$. Therefore, $f_c^*(n) = n$.

For $f(n) = \frac{n}{2}$ and $c = 2$, we have:

$$f(1)(n) = \tfrac{n}{2}$$
$$f(2)(n) = \tfrac{n}{4}$$
$$f(3)(n) = \tfrac{n}{8}$$
$$\ldots$$
$$f(i)(n) = \tfrac{n}{2^i}$$

So we need to solve for i in the equation $\frac{n}{2^i} \leq 2$, which gives $n \leq 2^{i+1}$, leading to $log_2 n \leq i + 1$. Therefore, $i = log_2 n - 1$, and similarly $f_c^*(n) = log_2 n - 1$.

For $f(n) = \sqrt{n}$ and $c = 2$, we need to find the smallest value of i such that $f(i)(n) \leq c = 2$. Using the definition of $f(i)(n) = f(f(i')(n))$, where $i' = i - 1$:

$$f(1)(n) = n^{\frac{1}{2}}$$
$$f(2)(n) = f(f(1)(n)) = f(n^{\frac{1}{2}}) = (n^{\frac{1}{2}})^{\frac{1}{2}} = n^{\frac{1}{4}}$$
$$f(3)(n) = n^{\frac{1}{8}}$$
$$\ldots$$
$$f(i)(n) = n^{\frac{1}{2}^i}$$

So we need to solve for i in the equation $\frac{1}{2}^i \leq 2$, which gives $log_2 n \leq \frac{1}{2}^i$. Therefore, $f_c^*(n) = log_{1/2} log_2 n$.

For $f(n) = \sqrt{n}$ and $c = 1$, we need to find the smallest value of i such that $f(i)(n) \leq c = 1$. We have:

$$n^{\frac{1}{2}^i} \leq 1$$

This means that $log_1 n \leq \frac{1}{2}^i$, where $1^n = 1$. Therefore, $f_c^*(n)$ has infinity of values.

For $f(n) = n^{1/3}$ and $c = 2$, we need to find the smallest value of i such that $f(i)(n) \leq c = 2$. We want the smallest i such that $n^{1/3^i} \leq 2$. Taking the logarithm base 3 of both sides of this inequality gives $\frac{1}{3^i} log_3 n log_3 2$, which simplifies to $(\frac{1}{3}^i) log_3 n \leq log_3 2$. Solving for i yields $i \geq \frac{(log_3 n)}{log_3 (1/2)} = \frac{(log_3 n)}{log_3 2}$. Since i must be an integer, the smallest possible value of i is $log_{1/3} log_2 n$.

Problem 4: Solving Recurrences

1.
To apply the Master Theorem to this recurrence relation, we can rewrite it in the form:

$T(n) = aT(n/b) + f(n)$, where $a = 37$, $b = 23$, and $f(n) = n$.

We then compute the value of the function $n^{\log_b a}$, which is $n^{\log_{23} 37} \approx n^{1.15}$.

Comparing $f(n)$ to $n^{\log_b a}$, we have:

$f(n) = n = n^1 = n^{\log_{23} 1} = \Theta(n^1)$,

which falls into case 1 of the Master Theorem.

There exists a constant $\varepsilon > 0$, such that $f(n) = O(n^{\log_{23} 37 - \varepsilon})$.

Therefore, the solution to the recurrence relation is:

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{1.15})$.

Therefore, we can say that the tight Θ bound on the recurrence relation $T(n) = 37 \cdot T(n/23) + n$ is $\Theta(n^{1.15})$.

2.
We can use the Guess and Check method to find a tight bound on the recurrence relation $T(n) = T(n/3) + T(n/4) + T(n/12) + n$.

First, let's make a guess that $T(n) = O(n)$. We can verify this guess by induction, assuming that $T(k) \leq ck$ for all $k < n$, where c is a constant. Then:

Base case: $T(1) = (5/3)$, there exists $c = 2$ such that $(5/3) < 2$.

Hypothesis: We assume that for all $n \leq x$, $T(x) \leq n$.

Inductive Step: $T(n+1) = T((n+1)/3) + T((n+1)/4) + T((n+1)/12) + n = (n+1)/3 + (n+1)/4 + (n+1)/12 + n + 1 = (5n+5)/3$ Given $c = 2$, for all $n \geq 5$, the hypothesis is proved correct.

Therefore, $T(n) = O(n)$.

Next, let's guess that $T(n) = \Omega(n)$. We can verify this guess by induction, assuming that $T(k) \geq ck$ for all $k < n$, where c is a constant. Then:

Base case: $T(1) = (5/3)$, there exists $c = 1$ such that $(5/3) > 1$.

Hypothesis: We assume that for all $n \leq x$, $T(x) \geq n$.

Inductive Step: $T(n+1) = T((n+1)/3) + T((n+1)/4) + T((n+1)/12) + n = (n+1)/3 + (n+1)/4 + (n+1)/12 + n + 1 = (5n+5)/3$ Given $c = 1$, for all $n \geq 0$, the hypothesis is proved correct.

Therefore, $T(n) = \Omega(n)$.

Since we have shown that $T(n) = O(n)$ and $T(n) = \Omega(n)$, we can conclude that $T(n) = \Theta(n)$.

Therefore, the tight bound for the recurrence relation $T(n) = T(n/3) + T(n/4) + T(n/12) + n$ is $\Theta(n)$.

3.
We can use the Master Theorem to find the tight bound for the recurrence relation $T(n) = T(n/2) + 2$ as follows:

First, $a = 1$, $b = 2$, and $f(n) = 2$. Then, we can calculate the value of $n^{\log_b a}$ as follows:

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

There exists a constant c that satisfies $f(n) = c * n^{\log_2 1}$. Thus, we can conclude that $f(n) = \Theta(1)$. We can apply Case 2 of the Master Theorem, which tells us that the Θ bound for $T(n)$ is $\Theta(logn)$.

Thus, we can conclude that the Θ bound for the recurrence relation $T(n) = T(n/2) + 2$ is $\Theta(logn)$.

