1 Fibonacci

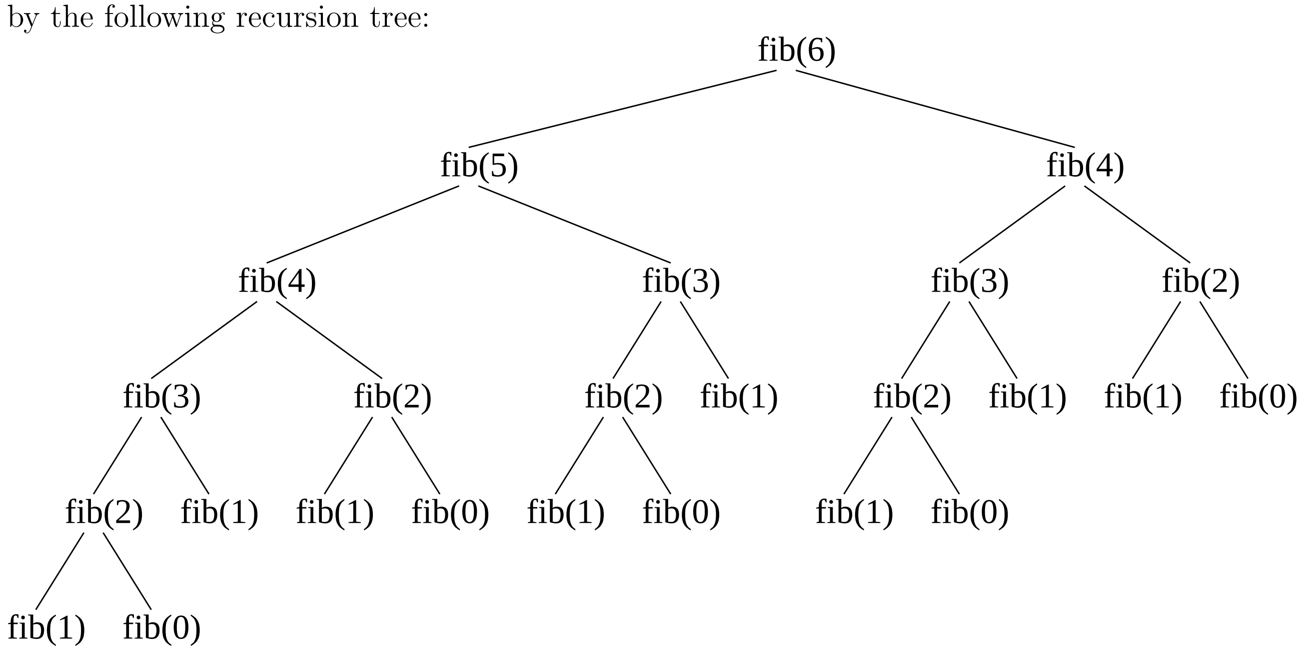
Recall the definition of the n^{th} Fibonacci number F_n:

$$F_n = \begin{cases} n & \text{for } n = 0, 1 \\ F_{n-1} + F_{n-2} & \text{for } n > 1 \end{cases}$$

(5 marks) How many times are each of `fib(6)`, `fib(5)`, `fib(4)`, `fib(3)`, `fib(2)` and `fib(1)` called by the algorithm below in order to calculate F_6?

```
public class Fibonacci
{
    public static int fibonacci(int n)
    {
        if (n ≤ 1)
            return n;
        else
            return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Answer:
To calculate F_6 we first call `fib(6)` which initiates several recursive calls to `fib` as illustrated by the following recursion tree:



Thus, `fib(6)` is called one time, `fib(5)` one time, `fib(4)` two times, `fib(3)` three times, `fib(2)` five times, and `fib(1)` eight times.

(20 marks) In general, how many times is `fib(m)`, $1 \le m \le n$, called to calculate F_n? Prove that your answer is correct by induction.

Answer:
You may have noticed from the previous answer that `fib(6)` is called F_1 times, `fib(5)` is called F_2 times, `fib(4)` is called F_3 times, `fib(3)` is called F_4 times, `fib(2)` is called F_5 times and `fib(1)` is called F_6 times. In general, then, `fib(m)` is called F_{n-m+1} times to calculate F_n. We prove this by induction on n.

base step: If $n = 1$ then $m = 1$ so `fib(m)` is called once which is equal to $F_{n-m+1} = F_1 = 1$. If $n = 2$ then `fib(m=1)` is called $F_{n-m+1} = F_2 = 1$ times and `fib(m=2)` is called $F_{n-m+1} = F_1 = 1$ times.

induction step: Assuming that `fib(m)` is called F_{k-m+1} times when calculating F_k and F_{k-m} times when calculating F_{k-1} for some constant $k \ge 2$, we prove that `fib(m)` is called F_{k-m+2} times when calculating F_{k+1}.

To calculate F_{k+1}, we make one call to `fib(k+1)` which in turn makes two recursive calls: `fib(k)` and `fib(k-1)`. The call to `fib(k)` calculates F_k so by our induction assumption it calls `fib(m)`, for $1 \le m \le k$, F_{k-m+1} times. The call to `fib(k-1)` calculates F_{k-1} so by our induction assumption it calls `fib(m)`, for $1 \le m \le k-1$, F_{k-m} times. Thus, for $1 \le m \le k-1$, `fib(m)` is called $F_{k-m+1} + F_{k-m}$ times, which by definition is equal to F_{k-m+2}.

For $m = k$, `fib(m)` is called only once, by `fib(k+1)`. Since $F_{k-m+2} = F_2 = 1$, the statement is true for $m = k$. Similarly, for $m = k+1$, `fib(m)` is called only once right at the beginning so the statement is true for $m = k+1$ as well since $Fk - m + 2 = F_1 = 1$.

2 Big-O

(20 marks) Use the 8 properties given on page 117 of the text to prove that $n^2(\log_2(n))^2\sqrt{n} + 3n^3$ is $O(\underline{\hspace{3cm}})$. Fill the blank with a function of your choice. Maximum marks are reserved for answers in which the chosen function is simplified and as small as possible (in the big-O sense).

Answer:

Step	Reason
1. $(\log_2(n))^2$ is $O(\sqrt{n})$	8(Log Powers)
2. $n^2\sqrt{n}$ is $O(n^2\sqrt{n})$	1(Constant Factor)
3. $n^2(\log_2(n))^2\sqrt{n}$ is $O(n^3)$	3(Multiplication) with steps 1 and 2
4. $3n^3$ is $O(n^3)$	1(Constant Factor)
5. $n^2(\log_2(n))^2\sqrt{n} + 3n^3$ is $O(n^3 + n^3)$	2(Addition) with steps 3 and 4
6. $n^3 + n^3$ is $O(n^3)$	5(Polynomial)
7. $n^2(\log_2(n))^2\sqrt{n} + 3n^3$ is $O(n^3)$	4(Transitivity) with steps 5 and 6

Notice that we cannot do any better because than $O(n^3)$ because $n^3 \le n^2(\log_2(n))^2\sqrt{n} + 3n^3$ for all $n \ge 1$.

3 More Big-O

(10 marks) Is n in $\Theta(n^2)$? Prove that your answer is correct using the definition of big-O.

Answer:
If n is $\Theta(n^2)$ then, by definition, n^2 is $O(n)$ which is not true so n is **not** $\Theta(n^2)$.

To prove that n^2 is $O(n)$ we must find constants $c > 0$ and $n_0 \geq 1$ such that $n^2 \leq cn$ for all $n \geq n_0$. However, this is impossible because, for any such c, $n^2 > cn$ for all $n > c$.

4 Master Theorem

(10 marks) Use the Master Theorem to determine the complexity of $T(n)$ defined below:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 32T(\frac{n}{4}) + 5n^2\sqrt{n} + n & n > 1 \end{cases}$$

Answer:
We have $a = 32$ and $b = 4$ so $n^{\log_b(a)} = n^{\log_4(32)} = n^{2.5}$. Since $f(n) = 5n^2\sqrt{n} + n$, big-O property 5 (Polynomial) implies that $f(n)$ is $O(n^{2.5})$. Furthermore, $n^{2.5} \leq f(n)$ for all $n \geq 1$ so by definition $n^{2.5}$ is $O(f(n))$. Therefore, $f(n)$ is $\Theta(n^{2.5}) = \Theta(n^{\log_b(a)})$ so we use case 2 of the Master Theorem which states that $T(n)$ is $\Theta(n^{2.5}\log(n))$.

5 Stacks

(10 marks) What does the method `foo` defined below do to a vector? You may illustrate with an example.

```
public class Foo {
    public static void foo(Vector vector)
    {
        Stack stack = new Stack();
        for (int i = 0; i < vector.size(); i++)
            stack.push(vector.elemAtRank(i));
        while (stack.size() > 0)
            vector.insertAtRank(vector.size(), stack.pop());
    }
}
```

Answer:
The easiest way to solve this problem is to investigate an actual example. Suppose that the vector stores the sequence (a, b, c, d). In the for-loop, then, we push a, then b, then c and finally d onto the stack so that at the end of the for-loop the stack looks like (d, c, b, a), assuming that the left side is the "top" of the stack. In the while-loop, we first execute `vector.insertAtRank(4,d)`, then `vector.insertAtRank(5,c)`, then `vector.insertAtRank(6,b)` and finally `vector.insertAtRank(7,a)`. Thus, in the end, the vector stores the sequence (a, b, c, d, d, c, b, a).

In general, `foo` appends the original sequence stored in the vector in reverse order to the end of the vector.

6 Matrices

Below is a simple algorithm for multiplying two $n \times n$ matrices A and B.

```
public class Matrix
{
    public static double[][] multiply(double[][] A, double[][] B)
    {
        int n = A.length;
        double[][] C = new double[n][n];

        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++) {
                C[i][j] = 0;
                for(int k = 0; k < n; k++)
                    C[i][j] += A[i][k] * B[k][j];
            }
        return C;
    }
}
```

Multiplying two $n \times n$ matrices A and B could also be accomplished using the following more complicated algorithm:

1. Construct $\frac{n}{2} \times \frac{n}{2}$ matrices A_1, A_2, A_3 and A_4 from A where

$$A = \begin{matrix} A_1 & A_2 \\ A_3 & A_4 \end{matrix}$$

   and similarly $\frac{n}{2} \times \frac{n}{2}$ matrices B_1, B_2, B_3 and B_4 from B where

$$B = \begin{matrix} B_1 & B_2 \\ B_3 & B_4 \end{matrix}.$$

2. Recursively multiply seven pairs of $\frac{n}{2} \times \frac{n}{2}$ matrices to obtain seven $\frac{n}{2} \times \frac{n}{2}$ matrices: $P_1 = A_1(B_2 - B_4)$, $P_2 = (A_1 + A_2)B_4$, $P_3 = (A_3 + A_4)B_1$, $P_4 = A_4(B_3 - B_1)$, $P_5 = (A_1 + A_4)(B_1 + B_4)$, $P_6 = (A_2 - A_4)(B_3 + B_4)$ and $P_7 = (A_1 - A_3)(B_1 + B_2)$.

3. Obtain four more $\frac{n}{2} \times \frac{n}{2}$ matrices: $R_1 = P_5 + P_4 - P_2 + P_6$, $R_2 = P_1 + P_2$, $R_3 = P_3 + P_4$ and $R_4 = P_5 + P_1 - P_3 - P_7$.

4. Finally return R, the product of A and B:

$$R = \begin{matrix} R_1 & R_2 \\ R_3 & R_4 \end{matrix}.$$

(25 marks) Which algorithm is faster for large n? Support your answer mathematically.

Answer:

(10 marks)First Algorithm Analysis

We first consider the number of times the comparison $k < n$ is executed. Each time the innermost for-loop is executed, this comparison is executed $n + 1$ times. Each time the middle for-loop is executed, the innermost for-loop is executed n times; therefore the comparison $k < n$ is executed $n(n + 1)$ times. The outermost for-loop is executed one time so the middle for-loop is executed n times in total. Therefore, the comparison $k < n$ is executed a total of $nn(n + 1) = n^3 + n^2$ times. Thus, the first algorithm uses *at least* a polynomial of degree 3 time; that is, it uses *at least* cn^3 time for some constant $c > 0$.

(10 marks)Second Algorithm Analysis

We analyze each step:

1. The first step of uses at most $O(n^2)$ time because A and B contain n^2 elements.

2. The second step first performs several additions, subtractions and multiplications. Each operation is applied to matrices containing $\left(\frac{n}{2}\right)^2 = \frac{n^2}{4}$ elements so each addition and subtraction uses at most $O(n^2)$ time. There are 10 additions and subtractions so altogether these use $O(n^2)$ time. Each multiplication is recursive and is applied to two $\frac{n}{2} \times \frac{n}{2}$ matrices, so if this algorithm uses $T(n)$ time to multiply $n \times n$ matrices then each of these recursive multiplications uses $T(\frac{n}{2})$ time. There are seven such multiplications so altogether they use $7T(\frac{n}{2})$ time. Therefore, in total this step uses $7T(\frac{n}{2}) + O(n^2)$ time.

3. The third step consists of 8 more matrix additions and subtractions so it uses at most $O(n^2)$ time.

4. The last step simply combines four $\frac{n}{2} \times \frac{n}{2}$ matrices into a single $n \times n$ matrix so it uses at most $O(n^2)$ time.

In total, then, the time $T(n)$ that this algorithm uses to multiply two $n \times n$ matrices is at most $7T(\frac{n}{2}) + O(n^2)$. This is a recurrence equation so we use the Master Theorem to simplify $T(n)$:

> We have $a = 7$ and $b = 2$ so $n^{\log_b(a)} = n^{\log_2(7)} \approx n^{2.81}$. We also have that $f(n)$ is $O(n^2)$ so $f(n)$ is $O(n^{\log_2(7)-\epsilon})$ for $\epsilon = \log_2(7) - 2 \approx 0.81$; therefore, we can use case 1 which states that $T(n)$ is $\Theta(n^{\log_2(7)}) \approx \Theta(n^{2.81})$.

Therefore, this second algorithm uses *at most* $\approx c'n^{2.81}$ time for some constant $c' > 0$ and large n.

(5 marks)Comparison:

The second algorithm is faster for large n because $cn^3 > c'n^{2.81}$ for all $n > \left(\frac{c'}{c}\right)^6$.