Algorithm Analysis and Design (CS1.301)

Monsoon 2021, IIIT Hyderabad
25 August, Wednesday (Lecture 3)

Multiple Ways of Solving Tractable Problems

We will ask the following three questions about any algorithm that we identify for a problem:

- Is it correct?
- What is its running time as a function of the input size?
- Can it be improved? If yes, how, and if not, why not?

Computing the $n^{\textbf{th}}$ Fibonacci Number

We know that the definition of the n^{th} Fibonacci number F_n is given by $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.

Given n, we wish to find F_n.

Solution 1

The most obvious algorithm simply makes use of the definition:

```
function fib1(n)
if n = 0: return 0
if n = 1: return 1
return fib1(n-1) + fib1(n-2)
```

Proving the correctness of this solution is straightforward.

In practice, however, this algorithm is too inefficient to be useful – with regard to both space and time efficiency.
To calculate the runtime, we note that

$$T(n) \leq 2, n \leq 1$$

$$T(n) = T(n-1) + T(n-2) + 3, n > 1$$

Therefore, $T(n) \geq F_n$ for all n. Noting that $F_n \approx 2^{0.694n}$, we can see that the runtime increases exponentially for this algorithm

Solution 2

We can implement memoisation to improve the runtime – avoiding overlapping computations by making use of iteration instead of recursion.

```
function fib2(n)
if n = 0: return 0
create array f[0..n]
f[0] = 0, f[1] = 1
for i = 2..n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

The correctness of this algorithm, too, is trivial.

Although this algorithm appears to be linear, it is in fact worse than that. This is because we need to use an arbitrary-size integer datatype (and not the typical bounded `int` type), for which addition is not a constant-time operation.

From the approximation in the previous section, we know that F_n is approximately $0.694n \in \mathcal{O}(n)$ bits long. Therefore each addition is $\mathcal{O}(n)$, which makes the overall runtime $\mathcal{O}(n^2)$.

Solution 3

We can calculate F_n without finding F_i for all $i \leq n$, but only a logarithmic fraction of it. To find such an algorithm, we note that

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix},$$

and therefore,

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Now, to calculate the value of the matrix power, we can use a "doubling technique" – we find M^2, M^4, M^8, and so on by successive doubling, and multiply some of them together to find M^n. For example, to find M^{20}, we square four times, and then find $M^{16} \cdot M^4$.

Since there are a logarithmic number of such squaring operations, each of which takes a constant number of integer multiplications, the complexity of this algorithm is $\mathcal{O}(M(n) \log n)$ (where $M(n)$ is the complexity of multiplying two n-bit integers, considered below). Clearly, if $M(n)$ is better than $\mathcal{O}(\frac{n^2}{\log n})$, this algorithm turns out to be better than algorithm 2.

Solution 4

We can use the direct formula for F_n:

$$F_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^2$$

However, this algorithm runs into accuracy issues due to the nature of irrational numbers. If we can store a sufficient-size array containing the value of $\sqrt{5}$, we still need to raise it to the power n, which cannot be better than algorithm 3 above.

The fastest known algorithm for calculating F_n runs in $\mathcal{O}(n \log^2 n)$.

Multiplying Large Integers

Given two n-digit numbers a and b, we wish to find $a \cdot b$.

Solution 1

The ordinary "school" algorithm we use takes $\mathcal{O}(n^2)$ single-digit multiplications and shifts and $\mathcal{O}(n)$ additions. Thus, this algorithm is $\mathcal{O}(n^2)$, which is not ideal.

Solution 2 (The Karatsuba Algorithm)

Consider multiplying two complex numbers $(a + bi) \cdot (c + id)$. At first sight, it appears that this takes 4 multiplications: $(ac - bd) + (ad + bc)i$.

However, it can be done in 3 multiplications: simply find $a \cdot c$, $b \cdot d$, and $(a + b) \cdot (c + d)$. The difference of the first two is the real part of the answer; the imaginary part is given by subtracting each of them from the third.

We can make use of this idea for the multiplication of real integers as well, called the *divide-and-conquer* method. Note that any number x can be written as $2^{\frac{n}{2}} x_1 + x_0$, splitting it into two $\frac{n}{2}$-bit numbers.
Thus, to multiply a and b, we apply the above technique to the multiplication $(2^{\frac{n}{2}} a_1 + a_0) \cdot (2^{\frac{n}{2}} b_1 + b_0)$. This can be carried out in three $\frac{n}{2}$-bit multiplications and some additions and subtractions, which we can recursively compute.

To find the time complexity of this algorithm, note that

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + T(1 + \frac{n}{2}) + \Theta(n),$$

from which we get that $T(n) \in \mathcal{O}(n^{\log_2 3}) = \mathcal{O}n^{1.585})$. Thus we have improved on the running time of algorithm 1.

Using this algorithm for multiplication in algorithm 3 of the Fibonacci problem, we improve on the $\mathcal{O}(n^2)$ bound for that problem, achieving $\mathcal{O}(n^{1.585} \log n)$.

The $\mathcal{O}(n^{1.585})$ running time can be improved upon using Fast Fourier Transform (FFT)-based algorithms – these reduce the running time to $\mathcal{O}(n \log n \log\log n)$.

An algorithm running in $\mathcal{O}(n \log n 2^{\mathcal{O}(\log * n)})$ is also known; the fastest known method, however, is $\mathcal{O}(n \log n)$.