[AI2615 Lecture 1] Word-RAM, Big-O notation, Fibonacci Numbers

1 The Computation Model

We already know from the introduction that what we mean by a "problem", and what is an "algorithm". Consider the problem of finding the maximum number from n numbers. We can write the following pseudo-code:

Algorithm: Find the Max Number
Input : n nonnegative numbers $a[1], a[2], \ldots, a[n]$.
Output: $\max_{i \in [n]} a[i]$.

```
max ← 0;
for i ← 1 to n do
    if a[i] > max then max ← a[i];
end
return max;
```

What is the running time of the algorithm? It really depends on the *computation model* we are working on.

In this course, we will work on the so-called word-RAM [Wik21]. The basic cell our algorithm can operate on is a w-bit "word". The memory consists of at most 2^m cells and each cell can store a word, namely a number in $\{0, 1, 2^m - 1\}$. We can access each cell of the memory in constant time and the basic arithmetic/logical operations ($+, -, \times, /$, boolean comparison, etc) on words cost constant time as well. For example, in the algorithm above, we can fetch the value of $a[i]$ from the memory and compare it with the value of the variable "max" in constant time. This is not far from practice since nowadays most CPUs have word size 64 and the memory is usually much less than 2^{64} bits.

2 The Big-O notation

In word-RAM, the above algorithm for finding max number cost at most $c \cdot n$ operations for some constant $c > 0$. In algorithm analysis, it is more convenient to drop the constant c and use the big-O notations to represent the running time of an algorithm.

Let $f, g : \mathbb{N} \to \mathbb{N}$ be two functions defined on $\mathbb{N}$. We say $f = O(g)$ if for some constant $c > 0$, $f(n) \leq c \cdot g(n)$ holds for every $n \in \mathbb{N}$. This is in fact a "slack" way to express $f \leq g$ ignoring constant multiples. We write $f = \Theta(g)$ iff $f = O(g)$ and $g = O(f)$.

We also say $f = \Omega(g)$ if $g = O(f)$. Our slack way to express $f < g$ is to use the little-O notation. We say $f = o(g)$ if for every $c > 0$, there exists

This is in fact consistent with what we learnt in the analysis: For two functions $f, g : \mathbb{R} \to \mathbb{R}_{>0}$, $f = O(g) \iff \lim_{x \to \infty} f/g \leq C$ for some constant $C > 0$.

some $n_0 \in \mathbb{N}$ such that $f(n) < c \cdot g(n)$ for any $n > n_0$. We say $f = \omega(g)$ if $g = o(f)$.

In analysis, $f = o(g)$ if $\lim_{x\to\infty} f/g = 0$

You can find examples and practice the use of these notations in the textbook. Here, we emphasis that $n \log n$ is more close to n than n^2 since $n \log n = o(n^{1.00001})$.

3 Computing Fibonacci Numbers

We're now ready to take our first steps into the world of algorithms. The *Fibonacci numbers* are a well-known sequence defined as

$$\texttt{Fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \texttt{Fib}(n-1) + \texttt{Fib}(n-2) & \text{if } n \geq 2. \end{cases}$$

The first few numbers in the sequence are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

In fact, $\texttt{Fib}(n)$ grows very fast, at a rate exponential in n. One can prove that $\texttt{Fib}(n) \geq 1.6^n$ for large n. Now we design algorithms to compute $\texttt{Fib}(n)$.

The first algorithm simply follows from the definition.

```
function fib1(int n)
    if n = 0 then return 0;
    if n = 1 then return 1;
    return fib1(n-1)+fib1(n-2);
end
```

`fib1` is obviously correct. Let us bound its running time. Let $T_1(n)$ denote the number of steps `fib1`(n) takes to output. Then clearly $T_1(0) = 1$, $T_1(1) = 2$ and for $n \geq 2$, $T_1(n) \geq T_1(n-1) + T_1(n-2) + 3$. Comparing to the definition of `Fib`, we can conclude that for every $n \geq 0$, $T(n) \geq \texttt{Fib}(n)$. So for large n, $T_1(n) > 1.6^n$, which is exponential in n.

`fib1` is inefficient since many of its computations are repeated. We can

improve it by storing the value already computed.

```
function fib2(int n)
    create an array f[0, ..., n];
    f[0] ← 0;
    f[1] ← 1;
    for k ← 2 to n do
1       f[k] ← f[k − 1] + f[k − 2];
    end
    return f[n];
end
```

How many steps `fib2` takes before termination? Let us denote this number by $T_2(n)$. Clearly the Line 1 has been executed exactly $n-1$ times. So do we have $T_2(n) = O(n)$?

No. Recall the computation model we are working on. In the word-RAM, we only assume those arithmetic operations *within* a word take constant number of steps. We assume the word size w is a constant[1]. Therefore, we need $O(k)$ words to store $\texttt{Fib}(k)$. Adding two length-k numbers takes $O(k)$ time. Therefore, $T(n) = O(2+3+\cdots+n) = O(n^2)$.

[1] This slightly deviates from our definition of the word-RAM. We create in `fib2` an array $f[0,\ldots,n]$ and therefore, in order to random access $f[n]$, w is at least $\log n$. However, this would introduce an annoying $\log\log n$ term in the expression. Therefore, in order to keep our discussion clean, we slightly change the model and assume w is constant.

Can we do it better? Linear algebra helps now. Note that we have for every $n \geq 1$, it holds that

$$\begin{bmatrix} \texttt{Fib}(n-1) \\ \texttt{Fib}(n) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \texttt{Fib}(n-2) \\ \texttt{Fib}(n-1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Therefore in order the compute $\texttt{Fib}(n)$, we only need to compute A^{n-1} for $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$. There is a trick to compute A^n faster than simply multiplying A by n times. First assume $n = 2^k$ for some $k \in \mathbb{N}$. Observe that $A^{2^k} = \left(A^{2^{k-1}}\right)^2$. We can obtain A^{2^k} by consecutively compute $A^1, A^2, A^4, A^8, \ldots, A^{2^{k-1}}, A^{2^k}$. We need to double a matrix for $k-1$ times and each time involves a constant number of multiplications. If n is not of the form 2^k, let $\overline{k} = \lceil \log_2 n \rceil$. We can write n as $n = \sum_{i=0}^{\overline{k}} a_i \cdot 2^i$ where each $a_i \in \{0,1\}$. Then $A^n = \prod_{i=0}^{\overline{k}} \left(A^{2^i}\right)^{a_i}$.

We also need to take into account the cost of computing $\left(A^{2^k}\right)^2$. Note that the numbers in A^{2^k} are of length $O(2^k)$ (Why?) and we need to compute the product of two length-$O(2^k)$ numbers for constant number of times. Let $M(t)$ to denote the number of steps to compute two length-t numbers. Then the total number of steps of our algorithm is

The algorithm we learnt in the elementary school to multiply two numbers takes $M(t) = O(t^2)$.

$$T_3(n) = O(M(1) + M(2) + M(4) + M(8) + \cdots + M(2^{\overline{k}})).$$

If $M(t) = \Theta(t^\alpha)$ for some $\alpha \in [1,2]$, then one can prove that $T_3(n) = O(M(n))$. Therefore, the performance of the algorithm depends on how fast

we can multiply two length-n numbers. We will see in future classes that we can do this in $O(n \log n)$ using the method of *Fast Fourier Transform*.

References

[Wik21] Wikipedia contributors. Word RAM — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Word_RAM&oldid=1020517599, 2021. [Online; accessed 18-February-2022]. 1