2.1 Computing the nth Fibonacci number

Remember the famous sequence of numbers invented in the 15th century by the Italian mathematician Leonardo Fibonacci? The sequence is represented as $F_0, F_1, F_2 \ldots$, where $F_0 = 0$, $F_1 = 1$, and for all $n \geq 2$, F_n is defined as $F_{n-1} + F_{n-2}$. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ The value of F_{30} is greater than a million! It is easy to see that the Fibonacci numbers grow exponentially. As an exercise, try to show that $F_n \geq 2^{n/2}$ for sufficiently large n by a simple induction.

Here is a simple program to compute Fibonacci numbers that slavishly follows the definition.

function $F(n$: integer): integer
if $n = 0$ then return 0
else if $n = 1$ then return 1
else return $F(n-1) + F(n-2)$

The program is obviously correct. However, it is woefully slow. As it is a recursive algorithm, we can naturally express its running time on input n with a *recurrence equation*. In fact, we will simply count the number of addition operations the program uses, which we denote by $T(n)$. To develop a recurrence equation, we express $T(n)$ in terms of smaller values of T. We shall see several such recurrence relations in this class.

It is clear that $T(0) = 0$ and $T(1) = 0$. Otherwise, for $n \geq 2$, we have

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

because to computer $F(n)$ we compute $F(n-1)$ and $F(n-2)$ and do one other addition besides. This is (almost) the Fibonacci equation! Hence we can see that the number of addition operations is growing very large; it is at least $2^{n/2}$ for $n \geq 4$ (you can verify this via induction).

Can we do better? This is the question we shall always ask of our algorithms. The trouble with the naive algorithm the wasteful recursion: the function F is called with the same argument over and over again, exponentially many times (try to see how many times $F(1)$ is called in the computation of $F(5)$). A simple trick for improving performance is to avoid repeated calculations. In this case, this can be easily done by avoiding recursion and just calculating successive values:

```
function F(n: integer): integer array A[0...n] of integer
A[0] = 0; A[1] = 1
for i = 2 to n do:
 A[i] = A[i-1] + A[i-2]
return A[n]
```

This algorithm is of course correct. Now, however, we only do $n-1$ additions.

It seems that we have come so far, from exponential to polynomially many operations, that we can stop here. But in the back of our heads, we should be wondering *an we do even better?* Surprisingly, we can. We rewrite our equations in matrix notation. Then

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

Similarly,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix},$$

and in general, Similarly,

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

So, in order to compute F_n, it suffices to raise this 2 by 2 matrix to the nth power. Each matrix multiplication takes 12 arithmetic operations, so the question boils down to the following: *how many multiplications does it take to raise a base (matrix, number, anything) to the nth power?* The answer is $O(\log n)$. To see why, consider the case where $n > 1$ is a power of 2. To raise X to the nth power, we compute $X^{n/2}$ and then square it. Hence the number of multiplications $T(n)$ satisfies

$$T(n) = T(n/2) + 1,$$

from which we find $T(n) = \log n$. As an exercise, consider what you have to do when n is not a power of 2.

So we have reduced the computation time exponentially again, from $n-1$ arithmetic operations to $O(\log n)$, a great achievement. Well, not really. We got a little too abstract in our model. In our accounting of the time requirements for all three methods, we have made a grave and common error: we have been too liberal about what constitutes an elementary step. In general, we often assume that each arithmetic step takes unit time, because the numbers involved will be typically small enough that we can reasonably expect them to fit within a computer's word. Remember, the number n is only $\log n$ bits in length. But in the present case, we are doing arithmetic on huge numbers, with about n bits, where n is pretty large. When dealing with such huge numbers, if exact computation is required we have to use sophisticated long integer packages. Such algorithms take $O(n)$ time to add two n-bit

numbers. Hence the complexity of the first two methods was larger than we actually thought: not really $O(F_n)$ and $O(n)$, but instead $O(nF_n)$ and $O(n^2)$, respectively. The second algorithm is still exponentially faster. What is worse, the third algorithm involves multiplications of $O(n)$-bit integers. Let $M(n)$ be the time required to multiply two n-bit numbers. Then the running time of the third algorithm is in fact $O(M(n))$.

The comparison between the running times of the second and third algorithms boils down to a most important and ancient issue: *can we multiply two n-bit integers faster than* $\Omega(n^2)$ *?* We saw in the first lecture that indeed this is possible, using Karatsuba's algorithm!

As a final consideration, we might consider the mathematicians' solution to computing the Fibonacci numbers. A mathematician would quickly determine that

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Justification of this is beyond the scope of the course, but as a bonus tidbit of knowledge for those familiar with linear algebra, the reason is that if we write

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

then we can diagonalize and write $A = PDP^{-1}$ where D has the eigenvalues of A on the diagonal. Then $A^n = PD^nP^{-1}$, and one can check that the eigenvalues of A are indeed $(1+\sqrt{5})/2$ and $(1-\sqrt{5})/2$, and the formula stated above is implied by also taking P and P^{-1} into consideration. In any case, using this mathematical fact, how many operations does it take to compute F_n? Note that this calculation would require floating point arithmetic. Whether in practice that would lead to a faster or slower algorithm than one using just integer arithmetic might depend on the computer system on which you run the algorithm.

2.2 Asymptotic Notation

In order to discuss algorithms effectively, we need to start with a basic set of tools. Here, we explain these tools and provide a few examples. Rather than spend time honing our use of these tools, we will learn how to use them by applying them in our studies of actual algorithms.

Induction

The standard form of the induction principle is the following:

If a statement $P(n)$ *holds for* $n = 1$*, and if for every* $n \geq 1$ $P(n)$ *implies* $P(n+1)$*, then P holds for all n.*

Let us see an example of this:

Claim 2.1 *Let* $S(n) = \sum_{i=1}^{n} i$. *Then* $S(n) = \frac{n(n+1)}{2}$.

Proof: The proof is by induction.

Base Case: We show the statement is true for $n = 1$. As $S(1) = 1 = \frac{1(2)}{2}$, the statement holds.

Induction Hypothesis: We assume $S(n) = \frac{n(n+1)}{2}$.

Reduction Step: We show $S(n+1) = \frac{(n+1)(n+2)}{2}$. Note that $S(n+1) = S(n) + n + 1$. Hence

$$\begin{aligned} S(n+1) &= S(n) + n + 1 \\ &= \frac{n(n+1)}{2} + n + 1 \\ &= (n+1)\left(\frac{n}{2} + 1\right) \\ &= \frac{(n+1)(n+2)}{2}. \end{aligned}$$

■

The proof style is somewhat pedantic, but instructional and easy to read. We break things down to the base case – showing that the statement holds when $n = 1$; the induction hypothesis – the statement that $P(n)$ is true; and the reduction step – showing that $P(n)$ implies $P(n+1)$.

Induction is one of the most fundamental proof techniques. The idea behind induction is simple: take a large problem ($P(n+1)$), and somehow *reduce* its proof to a proof of a smaller problems (such as $P(n)$; $P(n)$ is smaller in the sense that $n < n+1$). If every problem can thereby be broken down to a small number of instances (we keep reducing down to $P(1)$), these can be checked easily. We will see this idea of *reduction*, whereby we reduce solving a problem to a solving an easier problem, over and over again throughout the course.

As one might imagine, there are other forms of induction besides the specific standard form we gave above. Here's a different form of induction, called *strong induction*:

If a statement $P(n)$ *holds for* $n = 1$, *and if for every* $n \geq 1$ *the truth of* $P(i)$ *for all* $i \leq n$ *implies* $P(n+1)$, *then P holds for all n.*

Exercise: show that every number has a unique prime factorization using strong induction.

O Notation

When measuring, for example, the number of steps an algorithm takes in the worst case, our result will generally be some function $T(n)$ of the input size, n. One might imagine that this function may have some complex form, such as $T(n) = 4n^2 - 3n\log n + n^{2/3} + \log^3 n - 4$. In very rare cases, one might wish to have such an exact form for the running time, but in general, we are more interested in the rate of growth of $T(n)$ rather than its exact form.

The O notation was developed with this in mind. With the O notation, only the fastest growing term is important, and constant factors may be ignored. More formally:

Definition 2.2 *We say for non-negative functions $f(n)$ and $g(n)$ that $f(n)$ is $O(g(n))$ if there exist positive constants c and N such that for all $n \geq N$,*

$$f(n) \leq cg(n).$$

Let us try some examples. We claim that $2n^3 + 4n^2$ is $O(n^3)$. It suffices to show that $2n^3 + 4n^2 \leq 6n^3$ for $n \geq 1$, by definition. But this is clearly true as $4n^3 \geq 4n^2$ for $n \geq 1$. (**Exercise:** show that $2n^3 + 4n^2$ is $O(n^4)$.)

We claim $10\log_2 n$ is $O(\ln n)$. This follows from the fact that $10\log_2 n \leq (10\log_2 \mathrm{e})\ln n$.

If $T(n)$ is as above, then $T(n)$ is $O(n^2)$. This is a bit harder to prove, because of all the extraneous terms. It is, however, easy to see; $4n^2$ is clearly the fastest growing term, and we can remove the constant with O notation. Note, though, that $T(n)$ is $O(n^3)$ as well! The O notation is not tight, but more like a $\leq$ comparison.

Similarly, there is notation for $\geq$ and $=$ comparisons.

Definition 2.3 *We say for non-negative functions $f(n)$ and $g(n)$ that $f(n)$ is is $\Omega(g(n))$ if there exist positive constants c and N such that for all $n \geq N$,*

$$f(n) \geq cg(n).$$

We say that $f(n)$ is $\Theta(g(n))$ if both $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The O notation has several useful properties that are easy to prove.

Lemma 2.4 *If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(g_1(n) + g_2(n))$.*

Proof: There exist positive constants c_1, c_2, N_1, and N_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n \geq N_1$ and $f_2(n) \leq c_2 g_2(n)$ for $n \geq N_2$. Hence $f_1(n) + f_2(n) \leq \max\{c_1, c_2\}(g_1(n) + g_2(n))$ for $n \geq \max\{N_1, N_2\}$. ■

Exercise: Prove similar lemmata for $f_1(n)f_2(n)$. Prove the lemmata when O is replaced by Ω or Θ.

Finally, there is a bit for notation corresponding to $<<$, when one function is (in some sense) much less than another.

Definition 2.5 *We say for non-negative functions $f(n)$ and $g(n)$ that $f(n)$ is is $o(g(n))$ if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Also, $f(n)$ is $\omega(g(n))$ if $g(n)$ is $o(f(n))$.

We emphasize that the O notation is a tool to help us analyze algorithms. It does not always accurately tell us how fast an algorithm will run in practice. For example, constant factors make a huge difference in practice (imagine increasing your bank account by a factor of 10), and they are ignored in the O notation. Like any other tool, the O notation is only useful if used properly and wisely. Use it as a guide, not as the last word, to judging an algorithm.

Recurrence Relations

A recurrence relation defines a function using an expression that includes the function itself. For example, the Fibonacci numbers are defined by:

$$F(n) = F(n-1) + F(n-2),\ F(1) = F(2) = 1.$$

This function is well-defined, since we can compute a unique value of $F(n)$ for every positive integer n.

Note that recurrence relations are similar in spirit to the idea of induction. The relations defines a function value $F(n)$ in terms of the function values at smaller arguments (in this case, $n-1$ and $n-2$), effectively reducing the problem of computing $F(n)$ to that of computing F at smaller values. Base cases (the values of $F(1)$ and $F(2)$) need to be provided.

Finding exact solutions for recurrence relations is not an extremely difficult process; however, we will not focus on solution methods for them here. Often a natural thing to do is to try to guess a solution, and then prove it by induction. Alternatively, one can use a symbolic computation program (such as Maple or Mathematica); these programs can often generate solutions.

We will occasionally use recurrence relations to describe the running times of algorithms. For our purposes, we often do not need to have an exact solution for the running time, but merely an idea of its asymptotic rate of growth. For example, the relation

$$T(n) = 2T(n/2) + 2n,\ T(1) = 1$$

has the exact solution (for n a power of 2) of $T(n) = 2n\log_2 n + n$. (**Exercise:** Prove this by induction.) But for our purposes, it is generally enough to know that the solution is $\Theta(n \log n)$.

The following theorem is extremely useful for such recurrence relations:

Theorem 2.6 *The solution to the recurrence relation $T(n) = aT(n/b) + cn^k$, where $a \geq 1$ and $b \geq 2$ are integers and c and k are positive constants satisfies:*

$$T(n) \text{ is } \begin{cases} \Theta\left(n^{\log_b a}\right) & \text{if } a > b^k \\ \Theta\left(n^k \log n\right) & \text{if } a = b^k \\ \Theta\left(n^k\right) & \text{if } a < b^k. \end{cases}$$

Data Structures

We shall regard integers, real numbers, and bits, as well as more complicated objects such as lists and sets, as primitive data structures. Recall that a list is just an ordered sequence of arbitrary elements.

$$\text{List } q := [x_1, x_2, \ldots, x_n].$$

x_1 is called the head of the list.

x_n is called the tail of the list.

$n = |q|$ is the size of the list.

We denote by $\circ$ the concatenation operation. Thus $q \circ r$ is the list that results from concatenating the list q with the list r.

The operations on lists that are especially important for our purposes are:

head(q)	return(x_1)
push(q,x)	$q := [x] \circ q$
pop(q)	$q := [x_2, \ldots, x_n]$, return(x_1)
inject(q,x)	$q := q \circ [x]$
eject(q)	$q := [x_1, x_2, \ldots, x_{n-1}]$, return($x_n$)
size(q)	return(n)

The head, pop, and eject operations are not defined for empty lists. Appropriate return values (either an error, or an empty symbol) can be designed depending on the implementation.

A *stack* is a list that supports operations head, push, pop.

A *queue* is a list that supports operations head, inject and pop.

A *deque* supports all these operations.

Note that we can implement lists either by arrays or using pointers as the usual linked lists. Arrays are often faster in practice, but they are often more complicated to program (especially if there is no implicit limit on the number of items). In either case, each of the above operations can be implemented in a constant number of steps.

Application: Mergesort

For the rest of these notes, we will review the procedure mergesort. The input is a list of n numbers, and the output is a list of the given numbers sorted in increasing order. The main data structure used by the algorithm will be a queue. We will assume that each queue operation takes 1 step, and that each comparison (is $x > y$?) takes 1 step. We will show that mergesort takes $O(n \log n)$ steps to sort a sequence of n numbers.

The procedure mergesort relies on a function merge which takes as input two *sorted* (in increasing order) lists of numbers and outputs a single sorted list containing all the given numbers (with repetition).

```
function merge (s,t)
    list s,t
    n = size(s) + size(t)
    inject(s,∞)
    inject(t,∞)
    v = [ ]
    for i = 1 to n
        if head(s) < head(t) then
            inject(v, pop(s))
        else
            inject(v, pop(t))
    return v
end merge
```

```
function mergesort (s)
    list s, q
    q = [ ]
```

$$Q : [\,[7,9],[1,4],[6,16],[2,10] * [3,11,12,14],[5,8,13,15]\,]$$
$$Q : [\,[6,16],[2,10] * [3,11,12,14],[5,8,13,15],[1,4,7,9]\,]$$

Figure 2.1: One step of the mergesort algorithm.

```
    for x ∈ s
        inject(q, [x])
    while size(q) ≥ 2
        u := pop(q)
        v := pop(q)
        inject(q, merge(u, v))
    end
    if q = [ ] return [ ]
        else return q(1)
end mergesort
```

The total time taken by merge is $O(|s| + |t|)$ steps.

Question: Can you design a recursive (rather than iterative) version of merge? How much time does is take? Which version would be faster in practice, the recursive or the iterative?

The iterative algorithm mergesort above uses q as a queue of lists. (Note that it is perfectly acceptable to have lists of lists!) It repeatedly merges together the two lists at the front of the queue, and puts the resulting list at the tail of the queue.

The correctness of the algorithm follows easily from the fact that we start with sorted lists (of length 1 each), and merge them in pairs to get longer and longer sorted lists, until only one list remains. To analyze the running time of this algorithm, let us place a special marker $*$ initially at the end of the q. Whenever the marker $*$ reaches the front of q, and is either the first or the second element of q, we move it back to the end of q. Thus the presence of the marker $*$ makes no difference to the actual execution of the algorithm. Its only purpose is to partition the execution of the algorithm into phases: where a phase is the time between two successive visits of the marker $*$ to the end of the q. Then we claim that the total time per phase is $O(n)$. This is because each phase just consists of pairwise merges of disjoint lists in the queue. Each such merge takes time proportional to the sum of the lengths of the lists, and the sum of the lengths of all the lists in q is n. On the other hand, the number of lists is halved in each phase, and therefore the number of phases is at most $\log n$. Therefore the total running time of mergesort is $O(n \log n)$.

An alternative analysis of mergesort depends on a recursive, rather than iterative, description. Suppose we have

an operation that takes a list and splits it into two equal-size parts. (We will assume our list size is a power of 2, so that all sublists we ever obtain have even size or are of length 1.) Then a recursive version of mergesort would do the following:

```
function mergesort (s)
    list s, s1, s2
    if size(s) = 1 then return(s)
    split(s, s1, s2)
    s1 = mergesort(s1)
    s2 = mergesort(s2)
    return(merge(s1, s2))
end mergesort
```

Here split splits the list s into two parts of equal length s_1 and s_2. The correctness follows easily from induction.

Let $T(n)$ be the number of comparisons mergesort performs on lists of length n. Then $T(n)$ satisfies the recurrence relation $T(n) \leq 2T(n/2) + n - 1$. This follows from the fact that to sort lists of length n we sort two sublists of length $n/2$ and then merge them using (at most) $n-1$ comparisons. Using our general theorem on solutions of recurrence relations, we find that $T(n) = O(n \log n)$.

Question: The iterative version of mergesort uses a queue. Implicitly, the recursive version is using a stack. Explain the implicit stack in the recursive version of mergesort.

Question: Solve the recurrence relation $T(n) = 2T(n/2) + n - 1$ exactly to obtain an upper bound on the number of comparisons performed by the recursive mergesort variation.