5. Recursion, part 3

Programming and Algorithms II
Degree in Bioinformatics
Fall 2017
Recurrences

A way of recursively defining functions
We will use them to estimate running times of algorithms

\[ T(n) = \text{“maximum running time of the algorithm on an input of size n”} \]

Typical case:
\[ T(n) = f( T(\text{values smaller than n} ) ) \quad \text{if } n > \text{base cases} \]
\[ T(\text{base cases}) = \text{something not depending on } n \]
Recurrences

Recurrences can be solved in many ways
There are general methods that we don’t see here

Blackboard. Assuming \( T(1) = O(1) \):

\[
\begin{align*}
T(n) &= O(1) + T(n-1) \quad \Rightarrow \quad T(n) = O(n) \quad \text{(linear search)} \\
T(n) &= O(1) + T(n/2) \quad \Rightarrow \quad T(n) = O(\log n) \quad \text{(binary search)} \\
T(n) &= O(n) + T(n/2) \quad \Rightarrow \quad T(n) = O(n) \quad (?) \\
T(n) &= O(n) + T(n-1) \quad \Rightarrow \quad T(n) = O(n^2) \quad \text{(ins./sel. sort)} \\
T(n) &= O(n) + 2T(n/2) \quad \Rightarrow \quad T(n) = O(n \log n) \quad \text{(mergesort)} \\
T(n) &= O(1) + 2T(n-1) \quad \Rightarrow \quad T(n) = O(2^n) \quad \text{(set generation)}
\end{align*}
\]
Fibonacci numbers

\[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \]

We have
\[ A(n) \leq \text{fib}(n) \leq B(n) \]

where
\[ B(n) = B(n-1) + B(n-1) = \ldots = O(2^n) \]
\[ A(n) = A(n-2) + A(n-2) = \ldots = O(2^{n/2}) = O(1.41^{\ldots n}) \]

leads to guessing \( \text{fib}(n) \approx \varphi^n \) for some \( \varphi \) in \([1.41\ldots 2]\)
Fibonacci numbers

If \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \approx \varphi^n \)
then \( \varphi \) should satisfy
\[
\varphi^n = \varphi^{n-1} + \varphi^{n-2}
\]
dividing by \( \varphi^{n-2} \)
\[
\varphi^2 = \varphi + 1
\]
which means \( \varphi = \frac{1+\sqrt{5}}{2} = 1.618\ldots \), called the Golden Ratio
Pretty accurate estimate of \( \text{fib}(n) \) for large \( n \)

https://en.wikipedia.org/wiki/Golden_ratio
Merging two sorted lists

Given two lists that are sorted, compute a list with their union

\[ [1, 2, 2, 5, 6, 6, 9, 10, 10, 12] \]
\[ [0, 2, 4, 5, 5, 7, 8, 9, 9, 11, 12] \]

→

\[ [0, 1, 2, 2, 2, 4, 5, 5, 5, 6, 6, 7, 8, 9, 9, 9, 10, 10, 11, 12, 12] \]
def merge(lst1, lst2):
    i = 0
    j = 0
    result = []
    while i < len(lst1) and j < len(lst2):
        if lst1[i] <= lst2[j]:
            result.append(lst1[i])
            i = i+1
        else:
            result.append(lst2[j])
            j = j+1
    result.extend(lst1[i:])
    result.extend(lst2[j:])
    return result

At every iteration, we move either one element from lst1 or from lst2
Loop body is O(1)
Time O(len(lst1) + len(lst2))
Mergesort

Idea:

Given that merging two sorted lists is easy...

1. Split your big list into two lists
2. Sort each one separately
3. Merge the resulting sorted lists

Base case: list is sufficiently small to sort some other way
def mergeSort(lst):
    if len(lst) <= 1:
        return lst
    else:
        mid = (len(lst)+1)//2  # +1 is optional
        lefthalf = mergeSort(lst[:mid])
        righthalf = mergeSort(lst[mid:])
        return merge(lefthalf, righthalf)
Mergesort

\[ T(n) = 2T(n/2) + O(n) \quad (O(n) \text{ due to merge}) \]

\[ T(n) = cn + 2T(n/2) = cn + 2(cn/2 \quad + 2T(n/4)) \]

\[ = 2cn + 4T(n/4) = 3cn + 8T(n/8) = \ldots \]

\[ = i \times cn + 2^i \times T(n/2^i) \]

If we stop when \( n/2^i = 1 \), we have \( i = \log_2 n \), so time is

\[ T(n) = (\log n) \times cn = O(n \log n) \]

Uses additional memory, not in-place sorting

With some care, the copying of extra lists can be optimized

Good way of sorting sequential files in external memory
Multiplying large numbers

Given: two “large” integers $x, y$

Return: their product $x*y$

We can represent them e.g. in lists:

$[123, 456, 789]$ means the integer $12345689$

If each list element is $<B$, this is like using base $B$

(in many programming languages, int’s are limited to some fixed range e.g. $[-2^{32}...2^{32}]$; overflow is produced if exceeded; Python automatically extends int’s to be as large as required, BigInts)
Multiplying large numbers

Assumption:
• Sum is linear in #digits of operands
• Multiplying by $B^i$ is fast
• Same as adding 0’s at the end
• Dividing by $B^i$ is fast
• Same as removing digits from the end
• $O(#\text{digits} + i)$ time

In computers, $B=2$. Hardware directly supports product and division by powers of 2 (shifts)
Multiplying large numbers

School algorithm \((x,y)\)

\[
\text{sum} = 0 \\
\text{for } i \text{ in } [0 \ldots \text{numdigits}(y)-1] \\
\text{sum} += x \times y[i] \times 2^i
\]

Product by a digit \(y[i]\) and by \(2^i\) is \(O(\text{numdigits}(x))\)

\(O(n^2)\) if both \(x\) and \(y\) have \(n\) digits (and \(B\) constant)
Recursively

Numbers with 2k digits

\[(x_1 \times 2^k + x_0) \times (y_1 \times 2^k + y_0) = \]
\[x_1 \times y_1 \times 2^{2k} + (x_1 \times y_0 + x_0 \times y_1) \times 2^k + x_0 \times y_0\]

Two n-bit numbers \(\rightarrow\)

- 4 products of n/2 bit numbers
- + 3 sums + 2 shifts

\[T(n) = O(n) + 4T(n/2)\]
Recursively

\[ T(n) = cn + 4T(n/2) = cn + 4c(n/2) + 16T(n/4) = ... \\
= cn + 2cn + 4cn + 8cn + ... = O(n^2) \]

Otherwise: if we assume \( T(n) \approx O(n^a) \)
We must have

\[ d \cdot n^a = cn + 4(d(n/2)^a) = cn + 4/(2^a) \cdot dn^a \]

so \( 4/(2^a) \) must be \( \approx 1 \), so \( a=2 \)

Same \( O(...) \) time as school algorithm!
Can we do better?

We did \((x_1 \cdot 2^k + x_0) \cdot (y_1 \cdot 2^k + y_0) = x_1 \cdot y_1 \cdot 2^{2k} + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^k + x_0 \cdot y_0\)

4 multiplications \(\rightarrow\) 4 in the recurrence

\[ \rightarrow \frac{4}{(2^a)} = 1 \rightarrow O(n^2) \]

Can we save one multiplication?

Yes!
The Karatsuba – Ofman trick
Karatsuba Ofman method

We did \((x_1 \cdot 2^k + x_0) \cdot (y_1 \cdot 2^k + y_0) =\)
\[x_1 \cdot y_1 \cdot 2^{2k} + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^k + x_0 \cdot y_0\]

\(a = x_1 \cdot y_1\)
\(b = x_0 \cdot y_0\)
\(c = (x_1 + x_0) \cdot (y_1 + y_0) = x_1 \cdot y_1 + x_1 \cdot y_0 + x_0 \cdot y_1 + x_0 \cdot y_0\)
\(c = c - a - b = x_1 \cdot y_0 + x_0 \cdot y_1\)
result = \(a \cdot 2^{2k} + c \cdot 2^k + b\)

3 multiplications of \(n/2\) bit numbers, 6 sums, 2 shifts
The Karatsuba Ofman method

T(n) = cn + 3T(n/2)

If we assume T(n) ≈ O(n^a)

d n^a = cn + 3(d(n/2)^a) = cn + 3/2^a d n^a

so 3/2^a must be ≈ 1, so 3 = 2^a, so a = \log_2(3) = 1.585...

An algorithm for product running in time O(n^{1.585...})
Hard to imagine (and program) without recursion!
Don’t teach it to school kids
Better than school method for more than about 500 bits
Even better method

Schonhäge-Strassen’s algorithm. Very cool
Based on Discrete Fourier Transform
Starts like mergesort, then gets complex (pun!)

\[ O(n \log (n) \log \log(n)) \] time

Beats Karatsuba Ofman for 100,000’s of bits or so

Can we do \( O(n) \) (like sum)? We don’t know