

31251 – Data Structures and Algorithms

Week 4

Luke Mathieson

In this week's episode:

- Divide-and-Conquer
- Recursion
- Recursion vs Iteration

Divide-and-Conquer

Multiplying Large Numbers

- Everyone remembers (!) the multiplication algorithm we learnt in school:

$$\begin{array}{r} 14 \quad 13 \quad 9 \quad \times \\ 4 \quad 2 \\ \hline 1 \quad 18 \quad 7 \quad 8 \\ 1 \quad 7 \quad 5 \quad 6 \quad 0 \\ \hline 1 \quad 8 \quad 4 \quad 3 \quad 8 \end{array}$$

- Taking the length of the numbers as the size of the input, this is a $\Theta(n^2)$ -time algorithm.
- Can we do better?

Multiplying Large Numbers

- Maybe we can try breaking the problem down.
- If we have two numbers $a = a_1 a_2 \dots a_n$ and $b = b_1 b_2 \dots b_{mn}$, we can divide them each in half: $a_l = a_1 \dots a_{\lceil \frac{n}{2} \rceil}$, $a_r = a_{\lfloor \frac{n}{2} \rfloor} \dots a_n$, $b_l = b_1 \dots b_{\lceil \frac{n}{2} \rceil}$, and $b_r = b_{\lfloor \frac{n}{2} \rfloor} \dots b_n$.
- Then $a \times b = 10^n a_l b_l + 10^{\frac{n}{2}} (a_l b_r + b_l a_r) + a_r b_r$.
- So now we only have 4 half sized multiplications! ... Wait... that's exactly the same...

Now here's the tricky bit

- We want $a_l b_l$, $a_l b_r$, $b_l a_r$ and $a_r b_r$.
- Hey... $r = (a_l + a_r) \times (b_l + b_r) = a_l b_l + (a_l b_r + b_l a_r) + a_r b_r$.
- So if we calculate $p = a_l b_l$, $q = a_r b_r$ and r ...
- $a \times b = 10^n p + 10^{\frac{n}{2}}(r - p - q) + q$.
- So... we're now doing 3 multiplications of half the size. (At the cost of some addition and subtraction.)

What was the point of all that?

- So we're now doing 3 multiplications instead of 4... woo...
- What if we can break down a_l , b_l , a_r and b_r again?!?
- This is where recursion and abstracted functional design become the algorithmic paradigm of *divide-and-conquer*.
- Divide-and-conquer is where we solve a problem by recursively decomposing the instance into smaller instances of subproblems.
- If we do it correctly, we can do better than the naïve approach.

How well do we do with the multiplication?

- So how much time do we save with our better multiplication algorithm?
- To express running times of recursive algorithms, we use recurrence relations.

-

$$T_{mult}(n) = \begin{cases} 3T_{mult}(\frac{n}{2}) + c_1n & \text{for large enough } n \\ c_2n & \text{otherwise.} \end{cases}$$

- Okay... so how do we solve that?

For many recurrences we can use the Master Theorem. If $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ for some a, b and $f \in \Theta(n^k)$, then:

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

Back to the multiplication

- For our multiplication algorithm, $a = 3$, $b = 2$ and $k = 1$.
- So $a > b^k$, and $T_{mult} \in O(n^{\log_b a}) = O(n^{\log_2 3}) = O(n^{\approx 1.585})$.
- How much better is that? fooplot.com
- By breaking the problem down correctly, we have made a significant improvement! (Though there are even better multiplication algorithms)

- The general framework for a divide-and-conquer algorithm is:
 - ① Divide the instance into a set of subproblems.
 - ② If a subproblem is small enough, solve it, otherwise recursively split the subproblem.
 - ③ Combine the subproblem solutions into a whole solution.

What do we need to make Divide-and-Conquer work?

- A problem has to be recursively similar – can an instance be broken up into smaller instances of the same problem?
- The number and size of the subproblems must be in the right balance – this depends on the “simple” solution’s complexity.
- We must be able to recombine subsolutions into a solution efficiently.

Some example Divide-and-Conquer algorithms

- Binary Search – find if an element is in a sorted array (probably the simplest d&c algorithm):
 - ① Check the middle of the array, is the middle greater or smaller than the target element?
 - ② If it's the same, you've found it! If the array is size one and it's not the same, it's not there!
 - ③ If it's greater, search the left side of the array. Otherwise search the right.
- $O(\log n)$ time.

Some example Divide-and-Conquer algorithms

- Fast exponentiation – calculate x^n for large n :

Function $\text{exp}(x, n)$

if $n == 0$ **then**

 | return 1;

end

if n is odd **then**

 | return $x \times \text{exp}(x, n - 1)$;

end

else $x' = \text{exp}(x, \frac{n}{2})$;

 return $x' \times x'$;

 ;

end

- If we use D&C multiplication, and $|x| = m$, $\Theta(m^{\log_2 3} n^{\log_2 3})$

Some example Divide-and-Conquer algorithms

- Sorting: both Mergesort and Quicksort are D&C algorithms.
We'll see them soon!

Recursion

- Recursion is conceptually a central component of D&C.
- You can in fact phrase everything in computer science in terms of recursion (not a great idea, but possible).
- It often leads to “neat” code.
- It also often leads to inefficient and broken code.
- We will steal Jeff Edmonds “Friends” metaphor for framing recursive algorithm design (J. Edmonds, “How to Think About Algorithms”, Cambridge University Press, 2008.)

Designing a Recursive Algorithm with your “Friends”

- Carefully specify:
 - ① The Preconditions: what must be true about the input before you start the algorithm.
 - ② The Postconditions: what must be true about the output when you're done.
- These conditions then apply at every step of the recursion.
- Work out how to measure the “size” of an instance.

Designing a Recursive Algorithm with your “Friends”

- Consider a general instance of the problem.
 - ① Imagine you have friends who can magically solve any instance of the problem strictly smaller than yours if it meets the preconditions.
 - ② From your instance, construct subinstances that meet the preconditions.
 - ③ Get your friends to solve them.
 - ④ Recombine the subsolutions.

Designing a Recursive Algorithm with your “Friends”

- What if the subinstances don't fit the preconditions?
 - Then you have to rethink your preconditions (and maybe the postconditions).
- Keep the number of cases as small as possible.
- If the subinstance is small enough, just solve it using brute force.
- Analyse the algorithm using a recurrence.

Recursion vs Iteration

- Recursion often gives nice “simple” algorithms.
- Making them efficient can be trickier.
- Iterative algorithms are often easily made efficient (and easily optimised by compilers).
- But they result in hard-to-understand code (what are all those loop indices doing...?).

Which one is best then?

- There's no simple answer to which one to use.
- But both approaches can always be used.
 - Iteration and recursion are equally as powerful!
- You must think about what works best for the problem.

- Linear searching is trivial iteratively, mildly annoying recursively.
- Binary search is a great recursive algorithm, very messy iteratively.
- A lot of mathematical functions (e.g. the Fibonacci sequence) are easily expressed recursively – but are often much more efficient expressed iteratively.
- Traversing list-like structures – simple iteratively, but can be done recursively (remember `tail()`!).

Things to consider

- So when you're faced with deciding between iteration and recursion consider:
 - ① Is my problem naturally recursive or iterative?
 - ② Do I know how to terminate the algorithm (base cases for recursion, loop conditions for iteration)?
 - ③ Will I re-use the same information over and over?
 - A candidate for memoization - recursion, but where we remember what we've already calculated.
 - If memoization isn't good, then this should probably be iterative.
 - ④ Are the limits well defined (not just can I guarantee they will be met, but do I know them in advance)?
 - If not, recursion will probably be easier.
 - ⑤ Can I do it tail-recursively?
 - Has only one recursive call, at the end of the function - this is often easy to read, but the compiler can also optimise it into iteration.