31251 – Data Structures and Algorithms Week 4

Luke Mathieson / Marco Tomamichel

In this week's episode:

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



Multiplying Large Numbers

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

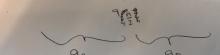
- Taking the length of the numbers as the size of the input, this is a $\Theta(n^2)$ -time algorithm.
- Can we do better?

Multiplying Large Numbers

- Maybe we can try breaking the problem down.
- If we have two numbers $a=a_1a_2\ldots a_n$ and $b=b_1b_2\ldots b_n$, we can divide them each in half: $a_l=a_1\ldots a_{\left\lceil\frac{n}{2}\right\rceil}$, $a_r=a_{\left\lceil\frac{n}{2}\right\rceil}\ldots a_n$, $b_l=b_1\ldots b_{\left\lceil\frac{n}{2}\right\rceil}$, and $b_r=b_{\left\lceil\frac{n}{2}\right\rceil}\ldots b_n$.
- 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...

$$4 \times \left(\frac{n}{2}\right)^2 = n^2$$

$$a = 356781534$$
 $a_1 a_2 a_3 - a_n$



$$a_{2}$$
 a_{1}
 a_{2}
 a_{3}
 a_{4}
 a_{5}
 a_{5}
 a_{5}
 a_{7}
 a_{7

Now here's the tricky bit

- We want a_1b_1 , a_1b_r , b_1a_r and a_rb_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.)

$$3 \times \left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2 < n^2$$

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.

$$T_{mult}(n) = egin{cases} 3\,T_{mult}(rac{n}{2}) + c_1 n & ext{ for large enough } n \geq N_0 \ c_2 & ext{ otherwise.} \end{cases}$$

• Okay... so how do we solve that?

How do we solve a recursion?

• Let's start with our example relation:

$$T_{mult}(n) = egin{cases} 3\,T_{mult}(rac{n}{2}) + c_1 n & ext{ for large enough } n \geq N_0 \ c_2 & ext{ otherwise.} \end{cases}$$

- To solve it, we either take out paper and pencil...
- Or...

$$T_{\text{mult}}(n) = 3T_{\text{mult}}(\frac{n}{2}) + \Theta(n)$$

$$= 3\left(3T_{\text{mult}}(\frac{n}{4}) + \Theta(n)\right) + \Theta(n)$$

$$= 9T_{\text{mult}}(\frac{n}{4}) + \Theta(n)$$

= 3 log Tmult(1) + 6(n)

 $= 2^{\log_2 3 \cdot \log_2 1} \Theta(1) + \Theta(n)$

O(m n log23)

The Master Theorem

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:

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

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? http://www.fooplot.com
- By breaking the problem down correctly, we have made a significant improvement! (Though there are even better multiplication algorithms)

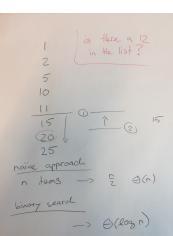
Divide-and-Conquer

- 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.

- Binary Search find if an element is in a sorted array (probably the simplest Divide-and-Conquer 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.
- This takes $O(\log n)$ time. Why?
- We can just try to solve this in our head...
- Or use that $T(n) = T(\frac{n}{2}) + O(1)...$



Recall: 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:

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

- In our example $T(n) = T(\frac{n}{2}) + O(1)$.
- Thus, a = 1, b = 2, k = 0.
- This implies $a = b^k$.
- And thus $T(n) \in \Theta(\log n)$.

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

```
Function exp(x,n)
    if n == 0 then
      return 1;
    if n is odd then
   return x \times exp(x, n-1);
   else x' = exp(x, \frac{n}{2});
return x' \times x';
```

end

• If we use Divide-and-Conquer multiplication, and |x| = m (i.e. we assume x has length m), then

$$\Theta(m^{\log_2 3} \log n)$$

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

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

Recursion

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"

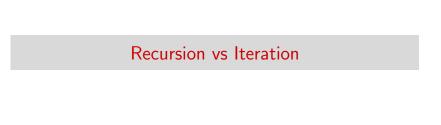
- Carefully specify:
 - 1 The Preconditions: what must be true about the input before you start the algorithm.
 - 2 The Postconditions: what must be true about the output when you're done.
- These conditions then apply at every step of the recursion.
- Work out how to measure the "size" of an instance.

Designing a Recursive Algorithm with your "Friends"

- Consider a general instance of the problem.
 - 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.

Some Examples

- 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.
 - 6 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.