Today

- Recursive algorithms
- Efficiency of recursion

See Aho, Hopcraft and Ullman, "Data structures and Algorithms", chapters 1,2.

Acknowledgements to Chris Mellish for slides.

$O(.)$ notation: official definition

For the record, the official definition of
when a function $T(n)$ is in the class $O(f(n))$ is as follows:

$T(n)$ is in $O(f(n))$ means that:

there are numbers k, n_0 such that for all $n > n_0$, $T(n) \leq k.f(n)$

It follows that $1000 + 67x^2 + 45x^3$ is in $O(x^3)$
(there is a bit of work to do here).

Smallest element of a list

Recall basic (1 step) operations on lists:

- Is the list empty?
- Get the first element, get the rest of the list
- Build a new list by tacking a new element on front of a list

Also arithmetic comparison $(=, <, \ldots)$ treated as single step.

Pseudo Code

```
To find smallest element of list L:
   If L is NIL, return NIL and stop
   Otherwise if the rest of L is NIL
      return the first of L and stop
   Otherwise  let S be smallest element of the rest of L
      If S is less than the first element of L
         return S and stop
      Otherwise
         return the first element of L
```

Note recursive case:

```
S be smallest element of the rest of L.
```

and two base cases. This may not be the most efficient way to solve the problem!

Termination

How can we work out if a recursive algorithm terminates
(i.e. stops and returns an answer)?

- There must be at least one base case, and recursion must eventually use one of them.
- The recursive sub-problem must be "smaller" than the original.
- show termination by
  – measuring complexity (e.g. by size of input)
  – showing complexity gets smaller in each recursive call
  – showing that it can't get smaller indefinitely, without hitting a base case.

Execution

Imagine a crowd of people executing the algorithm.

- The first person gets the original inputs (and program), and follows the algorithm, until
- when the algorithm is called again, they
  – find an unoccupied person
  – give them the subproblem, and copy of the algorithm
  – get back the answer from them
  – continue with the algorithm
- Finally, the first person hands over the answer.

Each occupied person has own memory, and record of where they are in the algorithm.

Execution ctd

This is roughly how such programs are executed:
a single computer keeps track of the different memories and executions.

Notice that each successive call to the procedure involves a different set of inputs, and the execution has to keep track of how these fit together.

This is an overhead, but it does not affect the the time complexity of execution.

Smallest at work

P1(S,L)	P2(S,L)	P3(S,L)	P4(S,L)
>[8,4,5,9]			
. . .	>[4,5,9]		
. . .	. . .	>[5,9]	
. . .	. . .	. . .	>[9]
. . .	. . .	. . .	<9
. . .	. . .	<5	
. . .	<4		
<4			

where ". . . " indicates waiting.

Complexity of recursive algorithm

- Use the notation $Cost(n)$ to represent the complexity of the algorithm with input of size n.
- Derive an equation for $Cost(n)$ in terms of $Cost(n-1)$ (or the appropriate notion of "smaller"), using the algorithm definition (indicate presence of constants).
- Solve the equation for $Cost(n)$ (ask a mathematician . . .)

Analysing Smallest

- The algorithm hits a base case (constant complexity) or a recursive subgoal (input size $n-1$) with some constant work:
- For the recursive case, we get $Cost(n) = k + Cost(n-1)$
- So:

$$\begin{aligned} Cost(n) &= k + Cost(n-1) \\ &= k + k + Cost(n-2) \\ &= \ldots \\ &= n \times k + Cost(0) \end{aligned}$$

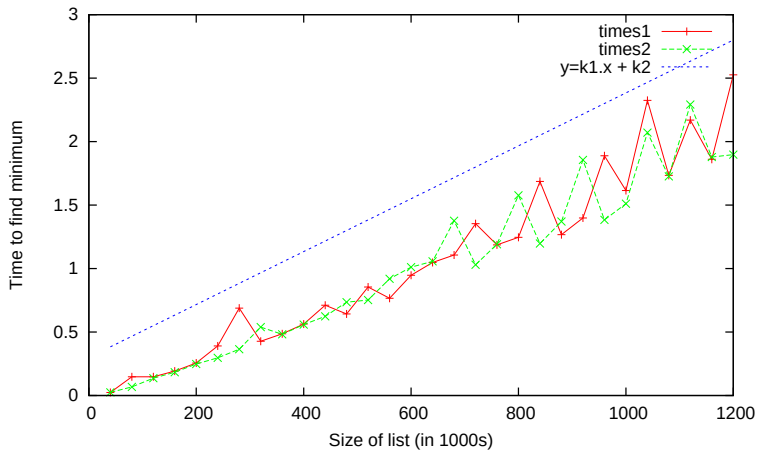
- So the complexity is linear: the algorithm is $O(n)$.

Empirical test

We can program "smallest" in our favourite programming language, and try running it on different sizes of lists, measuring the time taken.

Look at the plotted times as a function of list size (use random lists with entries from suitable range). We see that:

- there is fluctuation in time in actual execution
- times are bounded by a linear function $k_1 x + k_2$
- our analysis is only as good as our various simplifying assumptions about unit steps, etc; this is only an approximation, but it is useful.



Example: Reversing a list I

```
To reverse list L:
   If L is NIL return NIL and stop
   Otherwise
      Let Sub be the reverse of the rest of L
      Let Little be a new list pair
      Set the first of Little to the first of L
      Set the rest of Little to NIL
      Let Res be the result of appending Sub to Little
```

We assume we have a procedure for appending one list to another (tacking one list in front of another) that has linear complexity.

Simulation

P1(L,Sub,Res)	P2(L,Sub,Res)	P3(L,Sub,Res)	P4(L,Sub,Res)
>[a,b,c]			
. . .	>[b,c]		
. . .	. . .	>[c]	
. . .	. . .	. . .	>[ ]
. . .	. . .	. . .	<[ ]
. . .	. . .	<[c]	
. . .	<[c,b]		
<[c,b,a]			

Complexity

- Base case is constant complexity
- Recursive case involves constant + linear + recursion
- $Cost(n) = k + (n-1) + Cost(n-1)$
- $Cost(n) = k \times n + (n-1) + (n-2) + \cdots + 1 + 0 + Cost(0)$
- Complexity is quadratic

More complex Recursions

Often the structure of a recursion follows the structure of the data it operates on (lists,trees).

For example: we can represent a complex truth condition using "and", "or" and simple tests:

```
and(or(and(raining,warm),windy),or(humid,overcast))
```

Data representation

We can use a binary tree structure with labelled nodes.

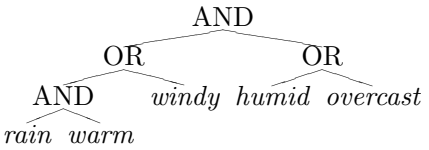
Such a tree is

- Either a leaf node on its own, labelled with a simple test, or
- it is a tree with two sub-trees, labelled with a connective ("and","or").

The primitive operations are:

- forming a tree from two trees and a connective
- deciding if a tree is a leaf, or has sub-trees
- accessing the test identity from a leaf
- accessing connective and subtrees from an internal node.

Example tree



Evaluating a test

```
To evaluate test T:
   If T is a leaf
      determine test and look up test result
   Otherwise
      Let L be result of evaluating left of T
      Let R be result of evaluating right of T
      If connective of T is AND
          If both L and R are TRUE, return TRUE
          Otherwise return FALSE
      Otherwise (so connective is OR)
          If one of L or R is TRUE, return TRUE
          Otherwise return FALSE
```

Assumes that tests are easily looked up. Try simulating this on the example tree.

Complexity of test evaluation

- Measure complexity of test as number of connectives and test expressions together
- base case is constant work
- Recursive case involves constant work + 2 recursive calls. The two recursive calls together involve $n-1$ connective and test expressions.
- $Cost(n) = k_1 + Cost(n-1-k_2) + Cost(k_2)$
- $Cost(n) = k_3 + n$ solves this (whatever k_2 is)
- Complexity is linear

Tractability

Some algorithms are intractable;
they need so much resource in terms of time (or space) that it is not practical to use them to solve big problems.

Often the cut-off point is characterised as follows:

Tractable = Polynomial time computable

Note that if we want real time computation, we will probably want to look a lot lower in the hierarchy.

Exponential time computation (or worse) is definitely bad, though.

Summary

- Recursive algorithms
- Estimating time complexity
- Tree data structure