MANCHESTER 1824
The University of Manchester

Algorithms and Their Performance

Loops, Recursion and Recurrence Relations

Joshua Knowles

School of Computer Science

The University of Manchester

COMP26120-Week 8 LT 1.1, 15 Nov 2013

This Lecture

1. Turning running time on its head — calculating the maximum input size given the amount of resource (time or #computers) available

2. Subroutines

3. Divide-and-conquer and recurrence relations

Calculating Maximum Input Size

Consider we have an equation for the time an algorithm runs

$$t = 3n^2 + 6n + 20$$

where t is the time and n is input size.

For large n we can use big-O to express the time complexity as t is $O(n^2)$.

Let us say that we can handle inputs of size $n = 35$ with current resources (time or computer power).

How big an input can we handle with ***double*** the resources?

Calculating Maximum Input Size

t is $O(n^2)$

We can drop the Big-O and rearrange for n:
$t = n^2$
$n = \sqrt{t}$
We wish to know how much bigger n will be in the case that t is doubled.

So we can find n for $t = 1$ and $t = 2$ and compare the sizes.

$n = n_1 = \sqrt{1} = 1$ when $t = 1$
$n = n_2 = \sqrt{2}$ when $t = 2$
$n_2/n_1 = \sqrt{2}/1 = \sqrt{2} \approx 1.41.$

So the size of instance we can solve is now 41% larger than before. Doubling the resource changes the size of input that we can handle from 35 to 49 only, since $35 \times 1.41 \approx 49$.

Exercise on Max Input Size

How much will the maximum input size grow (for reasonably large n) when we double the resources t available if:

$t = O(n)$

$t = O(1)$

$t = O(n^3)$

$t = O(2^n)$?

Derive the answers for practice.

Table of Max Input Sizes

Given that one operation takes 1 microsecond, complete the following table of maximum problem sizes that can be done in the times shown.

Running Time (μs)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	a	b
$2n^2$	c	d	e
2^n	f	g	h

(Adapted from Goodrich and Tamassia p.19)

E.g.:

$$400a = 60 \times 10^6 \quad (1)$$

$$a = 60 \times 10^6/400 \quad (2)$$

$$= 150,000 \quad (3)$$

$$2^f = 1 \times 10^6 \tag{4}$$

$$f = \log_2(10^6) \tag{5}$$

$$= 19.932 \tag{6}$$

$$= 19 \text{ since problem size must be integer} \tag{7}$$

For practice, fill in the rest.

Simple Loops: Complexity Analysis

```
begin
for i := 0 to N-2
  for j := N-1 down to i
     call_subroutine()
  endfor
endfor
x:=0
for k := 0 to 2*N
  x := x +k
endfor
end
```

What is the asymptotic time complexity when

1. the subroutine has complexity $O(1)$?
2. the subroutine has complexity $O(\log N)$?
3. the subroutine has complexity $O(2^N)$?

Simple Loops: Complexity Analysis

There are three loops. The first two are nested; we must ***multiply*** the number of steps in the inner loop by the number of times it is done (the number of iterations of the outer loop).

There is a subroutine which is inside the inner loop of the nested loop. Its complexity must be ***multiplied*** by the complexity of the nested loop.

The third loop is not nested with the other two. And it does not depend on them. So the algorithm will do the nested loop ***and then*** the final loop. So, we must ***add*** the complexities of the nested loop (including the subroutine call) to the complexity of the final loop.

Simple Loops: Complexity Analysis

Solutions to problems on slide 9.

1. $O(N^2).O(1) + O(N) = O(N^2)$
2. $O(N^2).O(\log N) + O(N) = O(N^2.\log N)$
3. $O(N^2).O(2^N) + O(N) = O(N^2.2^N)$

Divide-and-Conquer

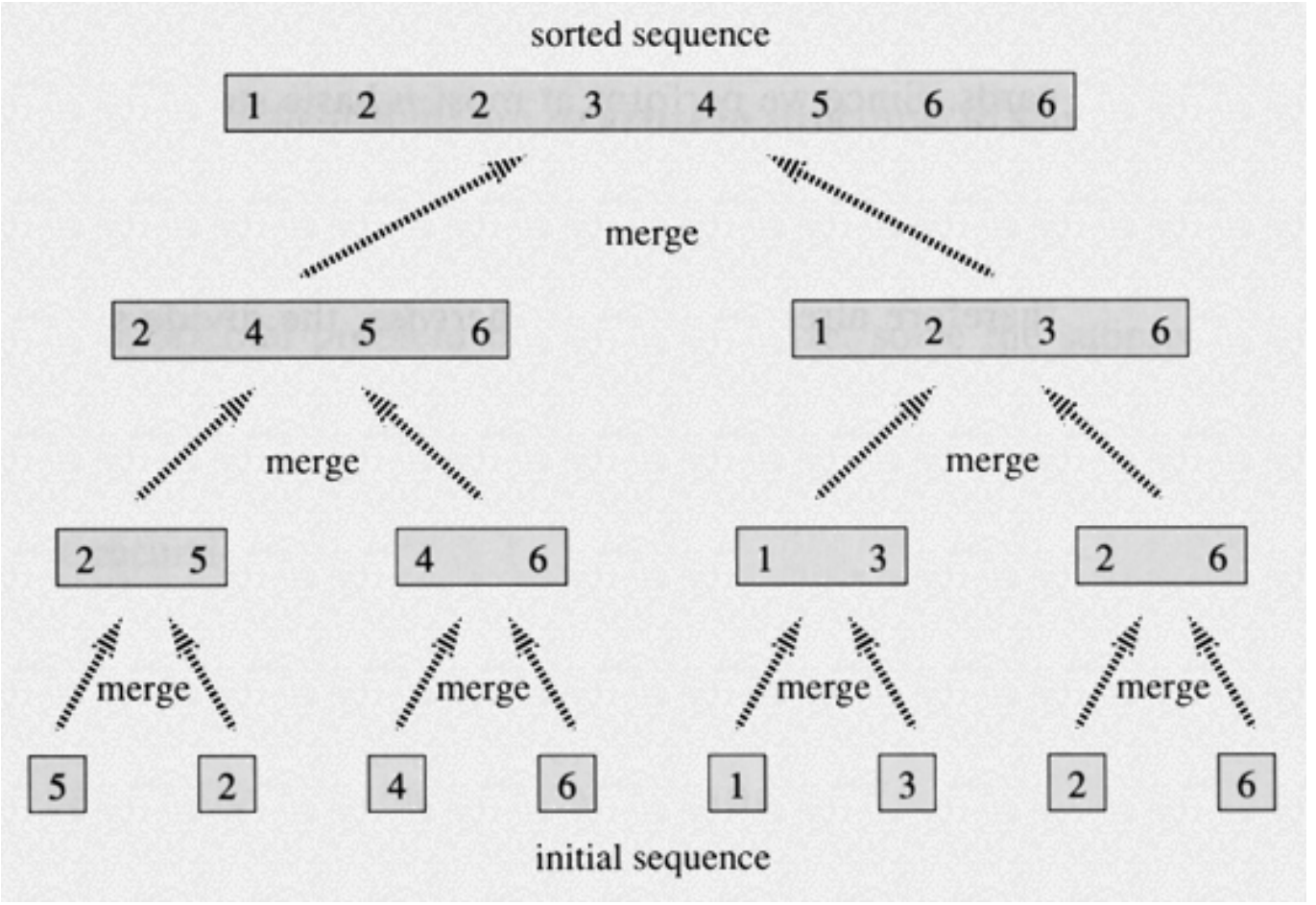
Many fast algorithms derive from the ***Divide-and-Conquer*** strategy.

Merge sort and quicksort are examples.

Fast integer multiplication and matrix multiplication algorithms too.

How do we analyse them?

Merge Sort


sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
initial sequence

Merge Sort

```
function merge_sort(list)
    if length(list) ≤ 1
        return list
    int *left, *right
    int middle := length(list) / 2
    for each x in list up to middle
        add x to left
    for each x in list after middle
        add x to right
    left := merge_sort(left)
    right := merge_sort(right)
    result := merge(left, right)
    return result
```

The merge operation takes $O(n)$ time for two lists total length n.

What about the rest of the algorithm?

Merge Sort (Continued)

We can write a ***recurrence relation*** to define the complexity.

$$T(n) = 2 * T(n/2) + n$$

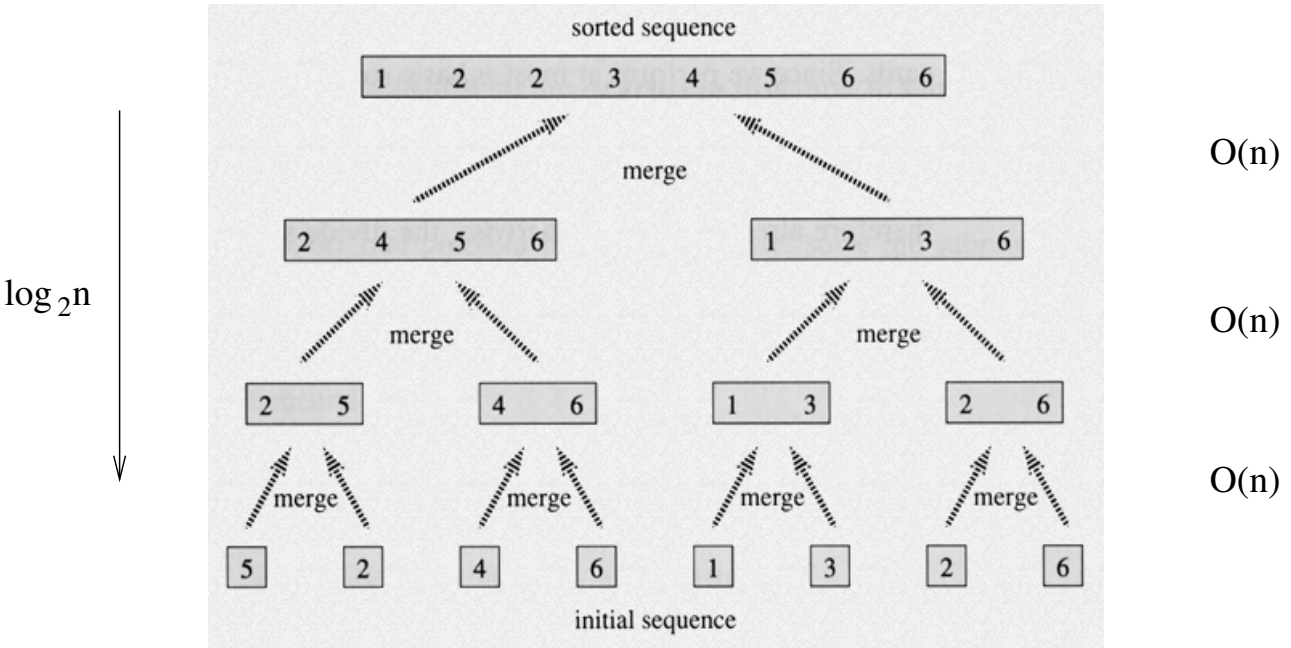
This defines T, the total number of operations done on input of size n, in terms of $T(n/2)$.

This is not a closed-form expression. I.e. not an equation that tells us what the Big-Oh is.

How do we solve this and find $T(n)$?

Merge Sort (Continued)

An easier way.



Since there is $O(n)$ complexity at each level, and a total of $O(\log n)$ levels, the complexity is $O(n \log n)$.

The Master Method

However, usually things will not be so easy.

You should use the Master Method.

A general recipe for solving recurrence relations for divide-and-conquer algorithms.

The Master Method

The recurrence will usually be in the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad \text{where} \quad a \geq 1, b > 1.$$

where

n size of the problem

a number of subproblems in the recursion

n/b size of each subproblem (assume all the same)

$f(n)$ dividing and merging cost(s)

The Master Theorem

The complexity will then be given by one of three cases:

1. If there is a small constant $\epsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.

2. If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.

3. If there are small constants $\epsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.

These three do not cover all possible cases, but most.

You Need: Big Omega and Big Theta

Big Oh means asymptotically ***less than or equal to***. E.g. $3n^3$ is $O(n^4)$ is valid. Equally, $3n^3$ is $O(n^3)$.

Big Omega means asymptotically ***greater than or equal to***. So $3n^3$ is $\Omega(n^2)$ is valid. So is $3n^3$ is $\Omega(n^3)$.

Big Theta means asymptotically ***equal to***. So $3n^3$ is $\Theta(n^3)$.

$f(n)$ is $O(g(n))$ and $\Omega(g(n)) \Leftrightarrow f(n)$ is $\Theta(g(n))$

Often when we say something is $O(g(n))$ we really mean it is $\Theta(g(n))$.

Practise your Big Omega and Big Theta

True or false?

1. $\log n$ is $\Theta(n \log n)$
2. 2^n is $\Omega(n^{20})$
3. If $f(n)$ is $\Theta(g(n))$, then $f(n)$ is $\Omega(g(n))$
4. $2n^3 + 5n$ is $\Theta(n^3)$

Example - Master Method

Using the Master Method, solve

$$T(n) = 2T(n/2) + n \log n$$

Here, $f(n)$ is $n \log n$, $a = 2$ and $b = 2$.

So $n^{\log_b a} = n^{\log_2 2} = n$.

Try CASE 2 of Master Method:

If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.

For $k = 1$ this works, since $f(n)$ is $\Theta(n \log n)$

Thus, $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$, which is $\Theta(n \log^2 n)$.

See Goodrich and Tamassia pages 269–273 for more examples.

Summary

We've now had 3 lectures on complexity.

You should now be able to

- Appreciate the relative growth rates of functions like n^3, 2^n and $n \log n$.
- Understand big-O
- Calculate the complexity of an algorithm from pseudocode (iterative)
- Do experiments to calculate/estimate running time
- Use the Master Theorem (with notes) to calculate the complexity of a recursive algorithm

Next time: sorting algorithms
Last lecture: amortized complexity + revision