COMP20007 Design of Algorithms

Analysis of Algorithms

Lars Kulik

Lecture 4

Semester 1, 2020

Establishing Growth Rate

In the last lecture we proved $t(n) \in O(g(n))$ for some cases of t and g, using the definition of O directly:

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

for some c and n_0. A more common approach uses

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies } t \text{ grows asymptotically slower than } g \\ c & \text{implies } t \text{ and } g \text{ have same order of growth} \\ \infty & \text{implies } t \text{ grows asymptotically faster than } g \end{cases}$$

Use this to show that $1000n = O(n^2)$.

L'Hôpital's Rule

Often it is helpful to use L'Hôpital's rule:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

where t' and g' are the derivatives of t and g.

For example, we can show that $\log_2 n$ grows slower than $\sqrt{n}$:

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

Induction Trap (Polya)

- $A(n)$: All horses are the same colour
- Base case: $A(1)$ is trivially true (only one horse)
- Assume in a set of n horses, all are the same colour
  - For a set of $n+1$ horses, take the subsets $\{1, \ldots, n\}$ and $\{2, \ldots, n+1\}$.
  - Both subsets are of size n, so all horses are the same colour in each subset (by inductive hypotheses).
  - Since $n-1$ of the horses are the same in both sets, the horses in both sets must be all the same colour, hence all $n+1$ horses are the same colour.
- What went wrong?

Example: Finding the Largest Element in a List

function MaxElement($A[0..n-1]$)
    $max \leftarrow A[0]$
    for $i \leftarrow 1$ to $n-1$ **do**
        if $A[i] > max$ **then**
            $max \leftarrow A[i]$
    return max

We count the number of comparisons executed for a list of size n:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n)$$

Example: Selection Sort

```
function SelSort(A[0..n − 1])
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < a[min] then
                min ← j
        swap A[i] and A[min]
```

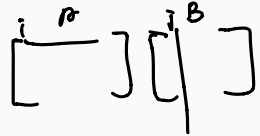
We count the number of comparisons executed for a list of size n:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = (n-1)^2 - \sum_{i=0}^{n-2} i$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

Example: Matrix Multiplication

function
MatrixMult($A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]$)
  for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
      $C[i,j] \leftarrow 0.0$
      for $k \leftarrow 0$ to $n-1$ **do**
        $C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]$
  return C



The number of multiplications executed for a list of size n is:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Analysing Recursive Algorithms

Let us start with a simple example:

function $\mathrm{F}(n)$
  if $n = 0$ **then return** 1
  else return $\mathrm{F}(n-1) \cdot n$

The basic operation here is the multiplication.

We express the cost recursively as well:

$$\begin{aligned} M(0) &= 0 \\ M(n) &= M(n-1) + 1 \quad \text{for } n > 0 \end{aligned}$$

To find a closed form, that is, one without recursion, we usually try "telescoping", or "backward substitutions" in the recursive part.

Telescoping

The recursive equation was:

$$\boxed{M(n) = M(n-1) + 1 \quad (\text{for } n > 0)}$$

Use the fact $M(n-1) = M(n-2) + 1$ to expand the right-hand side:

$$M(n) = [M(n-2) + 1] + 1 = M(n-2) + 2$$

and keep going:

$$\ldots = [M(n-3)+1]+2 = M(n-3)+3 = \ldots = M(n-n)+n = n$$

where we used the base case $M(0) = 0$ to finish.

A Second Example: Binary Search in Sorted Array

function BinSearch($A[], lo, hi, key$)
  if $lo > hi$ **then return** -1
  $mid \leftarrow lo + (hi - lo)/2$
  if $A[mid] = key$ **then return** mid
  else
    if $A[mid] > key$ **then**
      return BinSearch($A, lo, mid - 1, key$)
    else return BinSearch($A, mid + 1, hi, key$)

The basic operation is the key comparison. The cost, recursively, in the worst case:

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n/2) + 1 \quad \text{for } n > 0 \end{aligned}$$

Telescoping

A smoothness rule allows us to assume that n is a power of 2.
The recursive equation was:

$$\boxed{C(n) = C(n/2) + 1 \text{ (for } n > 0)}$$

Use the fact $C(n/2) = C(n/4) + 1$ to expand, and keep going:

$$\begin{aligned} C(n) &= C(n/2) + 1 \\ &= [C(n/4) + 1] + 1 \\ &= [[C(n/8) + 1] + 1] + 1 \\ &\vdots \\ &= \underbrace{[[\ldots[[C(0) + 1] + 1] + \cdots + 1] + 1]}_{1+\log_2 n \text{ times}} \end{aligned}$$

Hence $C(n) = \Theta(\log n)$.

Logarithmic Functions Have Same Rate of Growth

In O-expressions we can just write "log" for any logarithmic function, no matter what its base is.

Asymptotically, all logarithmic behaviour is the same, since

$$\log_a x = (\log_a b)(\log_b x)$$

So, for example, if ln is the natural logarithm then

$$\begin{aligned} \log_2 n &= O(\ln n) \\ \ln n &= O(\log_2 n) \end{aligned}$$

Also note that since $\log n^c = c \cdot \log n$, we have, for all constants c,

$$\log n^c = O(\log n)$$

Summarising Reasoning with Big-Oh

$$
\begin{aligned}
O(f(n)) + O(g(n)) &= O(\max\{f(n), g(n)\}) \\
c \cdot O(f(n)) &= O(f(n)) \\
O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)).
\end{aligned}
$$

The first equation justifies throwing smaller summands away.

The second says that constants can be thrown away too.

The third may be used with some nested loops. Suppose we have a loop which is executed $O(f(n))$ times, and each execution takes time $O(g(n))$. Then the execution of the loop takes time $O(f(n) \cdot g(n))$.

Some Useful Formulas

From Stirling's formula:

$$n! = O(n^{n+\frac{1}{2}})$$

Some useful sums:

$$\begin{aligned} \sum_{i=0}^{n} i^2 &= \tfrac{n}{3}(n+\tfrac{1}{2})(n+1) \\ \sum_{i=0}^{n}(2i+1) &= (n+1)^2 \\ \sum_{i=1}^{n} 1/i &= O(\log n) \end{aligned}$$

1+3+5+7-

See also Levitin's Appendix A.

Levitin's Appendix B is a tutorial on recurrence relations.

The Road Ahead

You will become much more familiar with asymptotic analysis as we use it on algorithms that we meet.

We shall begin the study of algorithms by looking at brute force approaches.