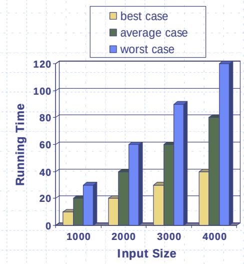
G52ADS: second lecture on Analysis of Algorithms

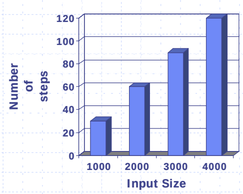
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.



Counting basic operations

- issues with implementing an algorithm in a particular language and running it on particular hardware
- Instead we determine how many steps the algorithm has to perform, as a function of the input size, in the worst case



Primitive Operations

- Basic computations performed by an algorithm
- Assumed to take a constant amount of time in the RAM model
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm	# operations
Algorithm *arrayMax*(*A*, *n*)	
currentMax ← *A*[0]	2
for *i* ← 1 **to** *n* − 1 **do**	2*n*
if *A*[*i*] > *currentMax* **then**	2(*n* − 1)
currentMax ← *A*[*i*]	2(*n* − 1)
{ increment counter *i* }	2(*n* − 1)
return *currentMax*	1
Total	8*n* − 3

Another example

Algorithm: *alg*

Input: positive integer n, which is a power of 2

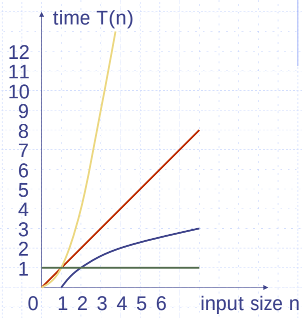
Output: integer m such that $2^m = n$

m* ← *0	1
while (n ≥ 2)	$\log_2(n)$
n* ← *n/2	$2 \log_2(n)$
m++	$2 \log_2(n)$
return m	1

all together $5 \log_2(n) + 2$

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ *1*
  - Logarithmic ≈ log *n*
  - Linear ≈ *n*
  - N-Log-N ≈ *n* log *n*
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n



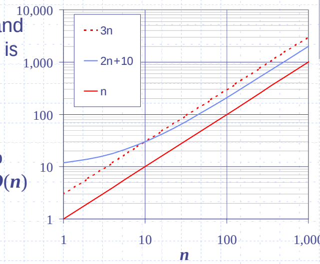
T(n) = 1, T(n) = $\log_2$ n,

T(n) = n, T(n) = n²

Big-Oh Notation

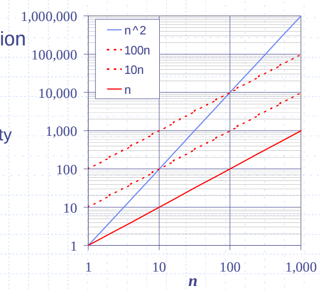
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \le cg(n)$ for $n \ge n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

  need c > 0 and $n_0 \ge 1$ such that 7n-2 ≤ c•n for $n \ge n_0$

  this is true for example for c = 7 and n_0 = 1
- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need c > 0 and $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le c \bullet n^3$ for $n \ge n_0$

  this is true for c = 4 and n_0 = 21
- 3 log n + 5

  3 log n + 5 is O(log n)

  need c > 0 and $n_0 \ge 1$ such that 3 log n + 5 ≤ c•log n for $n \ge n_0$

  this is true for c = 8 and n_0 = 2

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

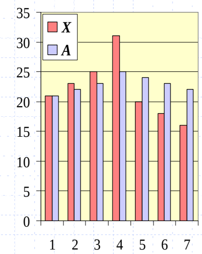
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8n - 3$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$

- Computing the array A of prefix averages of another array X has applications to financial analysis



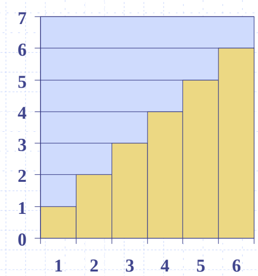
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X        #operations
    A ← new array of n integers                    n
    for i ← 0 to n − 1 do                          n
        s ← X[0]                                   n
        for j ← 1 to i do                          1 + 2 + …+ (n − 1)
            s ← s + X[j]                           1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                         n
    return A                                       1
```


Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



Other way...

1 + 2 + ... + (n-1) + n = ?

Easier to compute the sum twice:

1 + 2 + ... + (n-1) + n
+
n + (n-1) +...+ 2 + 1
= (n+1)+ (n+1)+...+(n+1) +(n+1) = n(n+1)
...and divide by 2:
1 + 2 + ... + (n-1) + n = n(n+1)/2.

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X        #operations
    A ← new array of n integers                    n
    s ← 0                                          1
    for i ← 0 to n − 1 do                          n
        s ← s + X[i]                               n
        A[i] ← s / (i + 1)                         n
    return A                                       1
```

- Algorithm ***prefixAverages2*** runs in $O(n)$ time

Math you need to Review

- Summations
- Logarithms and Exponents
  - **properties of logarithms:**

    $\log_b(xy) = \log_b x + \log_b y$

    $\log_b (x/y) = \log_b x - \log_b y$

    $\log_b x^a = a \log_b x$

    $\log_x a = \log_b a \log_x b$
  - **properties of exponentials**:

    $a^{(b+c)} = a^b a^c$

    $a^{bc} = (a^b)^c$

    $a^b / a^c = a^{(b-c)}$

    $b = a^{\log_a b}$

    $b^c = a^{c*\log_a b}$

Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is $\Omega(g(n))$ if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is $\Theta(g(n))$ if f(n) is asymptotically **equal** to g(n)

Informal coursework

- Please see the module web page:

http://www.cs.nott.ac.uk/~nza/G52ADS

- Tutorials to help with the mathematics involved: is Thursdays at 3 a good time? (I have not booked a room yet).