COMP20010

Data Structures and Algorithms I
06 - Algorithm Analysis II

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Formalities

Course Timetable COMP20010

Week	Topic	Assignment
7	Intro	
	Arrays, ADT's, Generics	
8	Linked Lists	
	Doubly Linked, Circularly Linked	A1
9	Analysis of Algorithms	
10	Recursion	
	Timing, Performance	A2
11	Stacks, Queues, Deques, Priority Queues	
12	Maps, hash functions	A3
	Review	

Algorithm Analysis

Outline

- Algorithm Analysis
- Big-Oh notation
- Examples

Constant Function

The simplest function we can think of is the `constant function` :

$$f(n) = c$$

for some fixed constant c, such as $c = 5$, $c = 27$, or $c = 2^{10}$. For any argument n the constant function assigns the value c. It doesn't matter what n is, $f(n)$ will always be c.

The constant function characterises the number of steps to perform a primitive operation (simple arithmetic, assignment, array access etc) on a computer.

Logarithm function

- log function defined as (constant $b > 0$):

$$f(n) = \log_b n$$

$$x = \log_b n \text{ if and only if } b^x = n$$

- computing the logarithm function exactly for any integer n involves the use of calculus, but we can use an approximation that is good enough without calculus.
- compute the smallest integer greater than or equal to $\log_a n$, since this number is equal to the number of times we can divide n by a until we get a number less than or equal to 1

Logarithm function

Example

$$\log_2 12 = 4 \qquad 12/2/2/2/2 = 0.75 \leq 1$$

This base-2 approximation arises in algorithm analysis, since a common operation in many algorithms is to repeatedly divide an input in half.

Base 2 logarithm is most common in computer science

$$\log n = log_2 n$$

$$\log_b(n) = \frac{\log_d n}{\log_d b}$$

In algorithm analysis we usually leave out the base- the denominator is a constant

Logarithm function

Logarithm rules (Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$)

$$\log_b ac = log_b a + log_b c \tag{1}$$

$$\log_b a/c = log_b a - log_b c \tag{2}$$

$$\log_b a^c = c \log_b a \tag{3}$$

$$\log_b a = (\log_d a)/\log_d b \tag{4}$$

$$b^{\log_d a} = a^{\log_d b} \tag{5}$$

Binary Search

- Given an initial call of BinarySearch(0, n-1, x), what is the worst case running time?
- Worst case is if x is not found. Then count how many iteration until we reach the base case (right>left)

```
1:  function BinarySearch(A, left, right, x)
2:      A an array
3:      left, right indexes into the array
4:      x the value to be found
5:      Output: index of the array at which we will find x
6:      if right > left then
7:          return not found
8:      end if
9:      mid ← floor((right − left)/2) + left
10:     if A[mid] = x then
11:         return mid
12:     end if
13:     if x < A[mid] then
14:         return binarySearch(A, left, mid − 1, x)
15:     else
16:         return binarySearch(A, mid + 1, right, x)
17:     end if
18: end function
```

Binary Search

- after iteration number

  1: there are n/2 items to search in

  2: (n/2)/2 = n/4

  k: $(n/2^k)$

- The last step occurs when
- $1 <= n/2^k$ and $n/2^{k+1} < 1$
- worst case is O(log n)

```
1:  function BinarySearch(A, left, right, x)
2:      A an array
3:      left, right indexes into the array
4:      x the value to be found
5:      Output: index of the array at which we will find x
6:      if right > left then
7:          return not found
8:      end if
9:      mid ← floor((right − left)/2) + left
10:     if A[mid] = x then
11:         return mid
12:     end if
13:     if x < A[mid] then
14:         return binarySearch(A, left, mid − 1, x)
15:     else
16:         return binarySearch(A, mid + 1, right, x)
17:     end if
18: end function
```

Linear Function

- given an input *n* the linear function always assigns the value *n* itself

$$f(n) = n$$

- arises when we need to perform an operation over *n* elements (eg. for loop)
- represents the best we can achieve for any algorithm which requires reading *n* elements into memory, since this requires *n* operations

nlogn function

- assigns to an input *n* the value of *n* times the log (base 2) of *n*

$$f(n) = n \log n$$

- grows a little faster than the linear function and much slower than the quadratic function
- some algorithms can be reduced from quadratic to *n*log*n* with big savings in time

Quadratic function

- for each *n* assigns the product of *n* with itself

$$f(n) = n^2$$

```
for(int i = 0; i < n; ++i) {
    for(int j = 0; j < n; ++j) {
        func();
    }
}
```

- common in algorithm analysis, usually where there are nested loops
- simple nested loops
- loops with changing indices

Quadratic function

- simple nested loops

$$f(n) = n^2$$

```
for(int i = 0; i < n; ++i) {
    for(int j = 0; j < n; ++j) {
        func();
    }
}
```

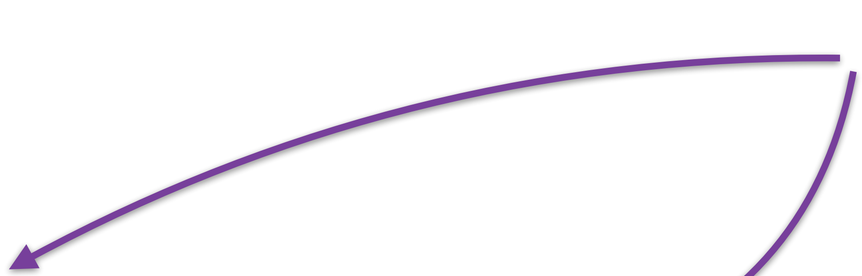
$O(n^2)$

- beware of what happens in func(). If func() has some dependence on n then we need to correct the analysis

Quadratic function

- bubble sort

$$f(n) = n^2$$




```
for(int j = n-1; j > 0; --j) {
    for(int k = 0; k < j; ++k) {
        func();
    }
}
```

(*n*-1) + (*n*-2) + …

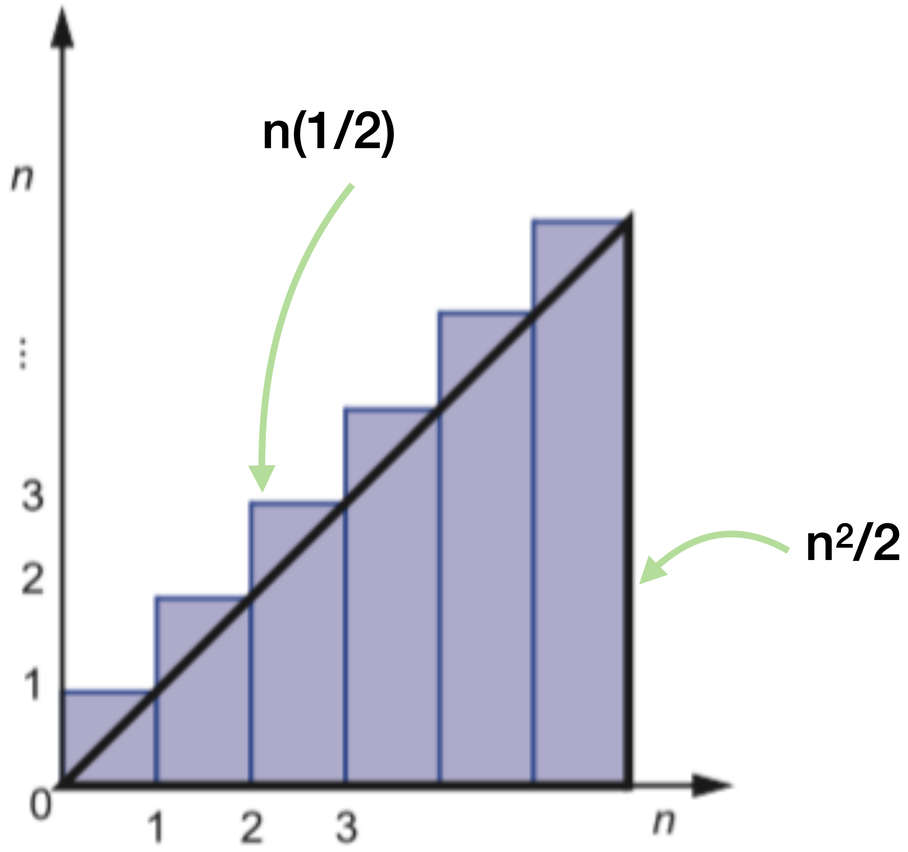
O(n^2)

Quadratic function

```
for(int j = n-1; j > 0; –j) {
    for(int k = 0; k < j; ++k) {
        func();
    }
}
```

$$1 + 2 + \ldots + (n-1) + n = \sum_{i}^{n} i$$

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



Cubic and other polynomial

- assigns to an input value *n* the product of *n* with itself 3 times

$$f(n) = n^3$$

- polynomial is a function of the form:

$$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$$

degree

integer d, highest power of the polynomial

coefficients

constants and a_d!=0

$$f(n) = 2 + 3n + 6n^2$$

$$f(n) = 1 + n^3$$

$$f(n) = 1$$

$$f(n) = n^2$$

Exponential function

- assigns to the input argument *n* the value obtained by multiplying the base *b* by itself *n* times

$$f(n) = b^n$$

- In algorithm analysis, the most common base for the exponential function is *b* = 2. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the *n*th iteration is 2^n.

rules for exponents

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

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

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

Exponential function

- geometric sums

$$f(n) = b^n$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n$$

$$= \frac{a^{n+1} - 1}{a - 1}$$

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$$

Exponential complexity

The binary recursive fibonacci is one of few exponential functions we will encounter

$$
\begin{aligned}
C_0 &= 1 \\
C_1 &= 1 \\
C_2 &= C_1 + C_0 + 1 = 1 + 1 + 1 &&= 3 \\
C_3 &= C_2 + C_1 + 1 = 3 + 1 + 1 &&= 5 \\
C_4 &= C_3 + C_2 + 1 = 5 + 3 + 1 &&= 9 \\
C_5 &= C_4 + C_3 + 1 = 9 + 5 + 1 &&= 15 \\
C_6 &= C_5 + C_4 + 1 = 15 + 9 + 1 &&= 25 \\
C_7 &= C_6 + C_5 + 1 = 25 + 15 + 1 &&= 41 \\
C_8 &= C_7 + C_6 + 1 = 41 + 25 + 1 &&= 67
\end{aligned}
$$

The time taken to compute the nth fibonacci number is the sum of the time taken to compute the previous two:

$$T(n) = T(n-1) + T(n-2)$$

and since: $T(n-1) = T(n-2) + T(n-3)$ then we can say that:

$$T(n) = 2 * T(n-2) + T(n-3)$$

The time taken is doubling, but not every n, instead it doubles every second n.

$$T(n) \sim 2^{n/2}$$

and so we say the growth factor is exponential: $O(2^n)$. You can the

Analysis

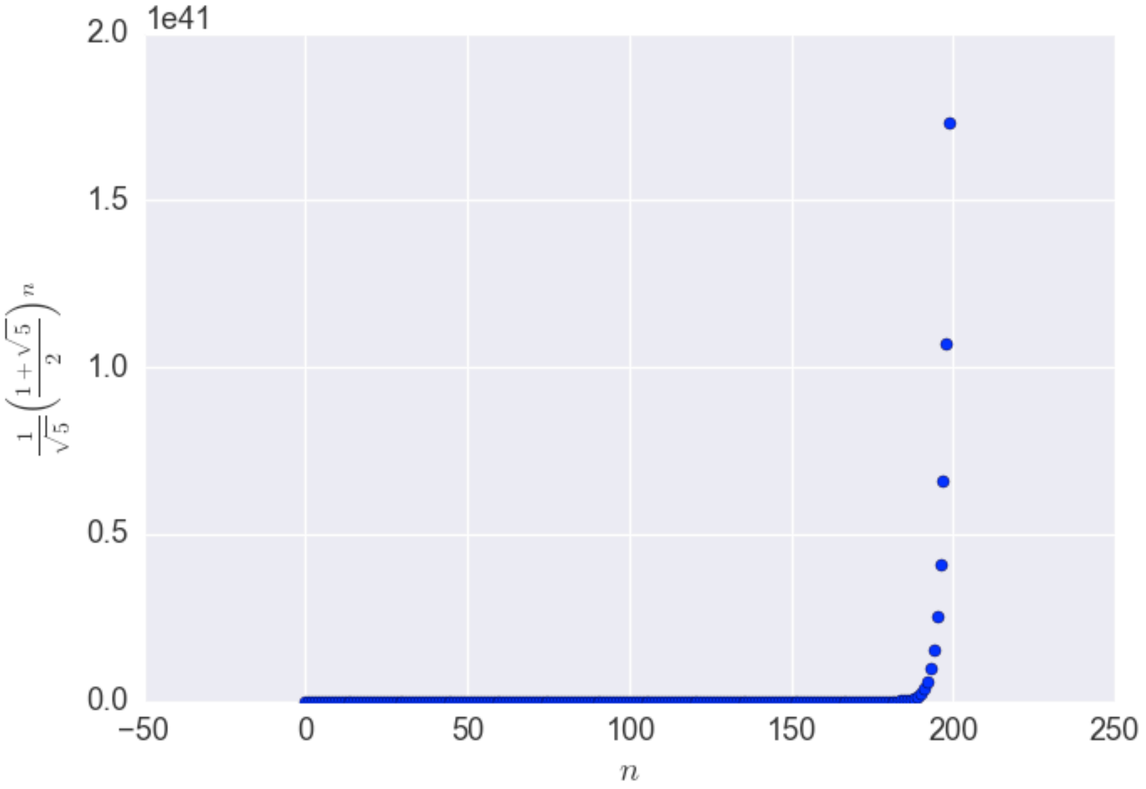
we don’t always need to bother with recursive and iterative solutions and complex analysis

sometime there is an easier solution, but it might be harder to find

nth fibonacci number is the integer part of:

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

takes a millisecond to run in python



TAK

- A recursive function, named after Ikuo Takeuchi
- function is called recursively in the function arguments
- used a benchmark for languages which try to optimise recursion
- very easy to implement, try it out in Java
- what is its running time complexity?

```
1: function TAK(x, y, z)
2:     if y < x then
3:         tak(tak(x − 1, y, z), tak(y − 1, z, x), tak(z − 1, x, y))
4:     else
5:         z
6:     end if
7: end function
```

Analysis

does anyone recognise this algorithm?

```
1:  function M(a[1...p], b[1...p], base)
2:      product ← [1...p + q]
3:      for b_i = 1 doq
4:          for a_i dop
5:              product[a_i + b_i − 1]+ = carry + a[a_i] ∗ b[b_i]
6:              carry = product[a_i + b_i − 1]/base
7:              product[a_i + b_i − 1] = product[a_i + b_i − 1]   mod   base
8:          end for
9:          product[b_i + p]+ = carry
10:     end for
11:     return product
12: end function
```

Multiplication

long multiplication is O(n²)

[2,3,9,5,8,2,3,3] x [0,0,0,0,5,8,3,0]

```
   23958233
×      5830
------------------
    00000000 (=     23,958,233 ×     0)
   71874699  (=     23,958,233 ×    30)
  191665864  (=     23,958,233 ×   800)
+ 119791165  (=     23,958,233 × 5,000)
------------------
 139676498390 (= 139,676,498,390       )
```

```
1:  function MULTIPLY(a[1...p], b[1...p], base)
2:      product ← [1...p + q]
3:      for b_i = 1 doq                                          n
4:          for a_i dop                                          n
5:              product[a_i + b_i − 1] += carry + a[a_i] ∗ b[b_i]
6:              carry = product[a_i + b_i − 1]/base
7:              product[a_i + b_i − 1] = product[a_i + b_i − 1]  mod  base
8:          end for
9:          product[b_i + p] += carry
10:     end for
11:     return product
12: end function                                                 O(n²)
```

Multiplication

- much faster multiplications algorithms have been developed:
- Karatsuba (1960) has complexity:

$$O(n^{\log_2(3)}) \sim O(n^{1.585})$$

- first algorithm faster than long multiplication
- applications for BigIntegers
- also cryptography
- for small n, long multiplication is faster, but there is a crossover point. Python/Java use Karatsuba for integer multiplication.

Comparing Growth rates

- functions we use in order of growth rate

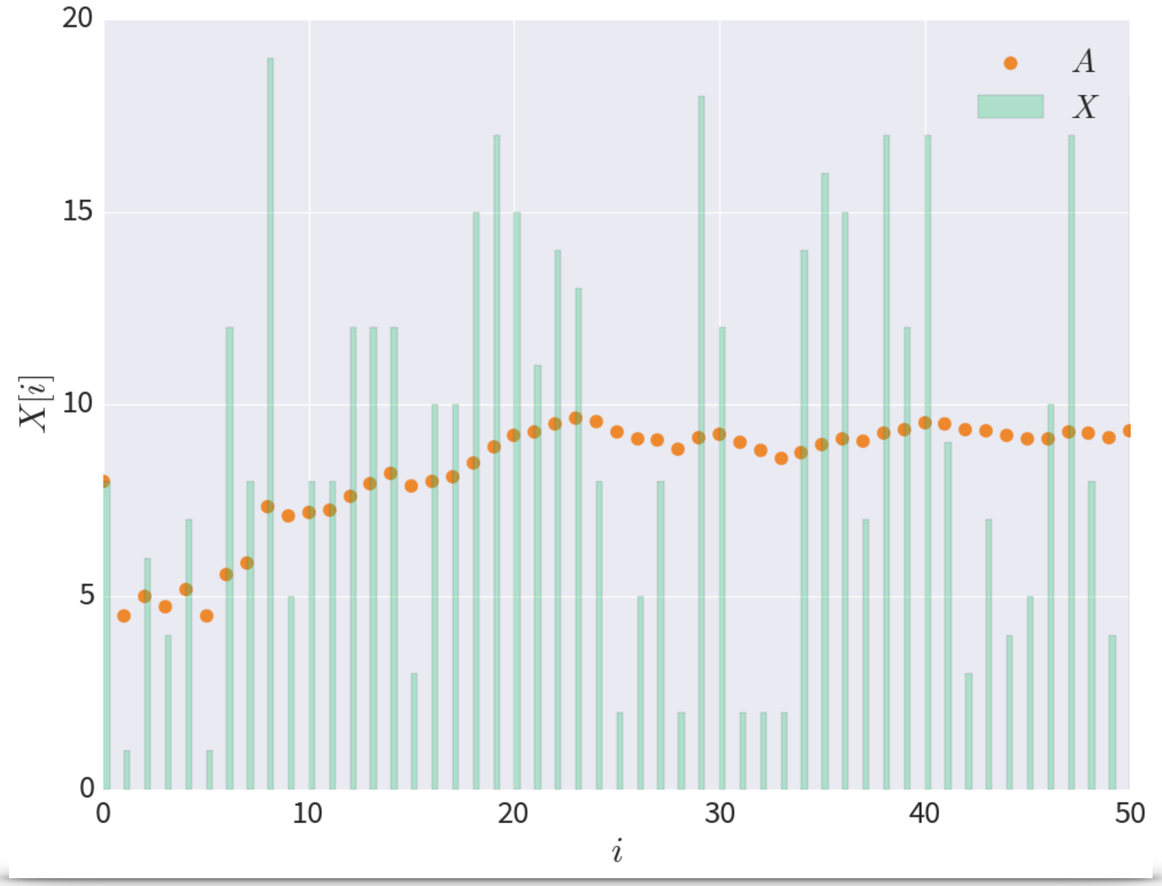
constant	logarithm	linear	n-logn	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Prefix Averages

We 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***

has applications in financial analysis, statistics, time windowing etc



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

Prefix Averages

this is a simple quadratic time algorithm

```
1:  function prefixAverage (X, n)
2:      Input: array X of n integers
3:      Output: array A of prefix averages of X
4:      A ← new array of n integers
5:      for i ← 0 to n − 1 do
6:          s ← X[0]
7:          for j ← 1 to i do
8:              s ← s + X[j]
9:          end for
10:         A[i] ← s/(i + 1)
11:     end for
12:     return A
13: end function
```

Prefix Averages

```
1:  function prefixAverage (X, n)
2:      Input: array X of n integers
3:      Output: array A of prefix averages of X
4:      A ← new array of n integers                          ▷ n
5:      for i ← 0 to n − 1 do                                ▷ n
6:          s ← X[0]                                         ▷ 2n
7:          for j ← 1 to i do
8:              s ← s + X[j]                                 ▷ 1 + 2 + 3 + ... + (n − 1)
9:          end for
10:         A[i] ← s/(i + 1)                                 ▷ n
11:     end for
12:     return A
13: end function                                             ▷ Total: O(n^2)
```

nested loops i, j

inner loop is executed i+1 times: O(n^2)

Prefix Averages

- our simple prefix averages algorithms is O(1 + 2 + … + (n-1) + n)
- $O(n^2)$

$$1 + 2 + \ldots + (n-1) + n = \sum_{i}^{n} i$$

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

Prefix Average

- we notice the two consecutive averages are very similar
- clue to rework the algorithm

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

Prefix Average

- by keeping a running sum we can reduce the growth factor to O(*n*)

```
1:  function prefixAverage (X, n)
2:      Input: array X of n integers
3:      Output: array A of prefix averages of X
4:      A ← new array of n integers                    ▷ n
5:      s ← 0                                          ▷ 1
6:      for i ← 0 to n − 1 do                          ▷ n
7:          s ← s + X[i]                               ▷ 3n
8:          A[i] ← s/(i + 1)                           ▷ 4n
9:      end for
10:     return A
11: end function                                       ▷ Total: O(n)
```

Memoisation

- a technique to speed up algorithms by keeping track of the result of an expensive operation, and returning just the cached result if the same operation is called again
- lookup table

```
1:  function FACTORIAL (n)
2:      Input: n non-negative integer
3:      Output: factorial of n: n!
4:      A ← new array of n integers
5:      s ← 0
6:      if n = 0 then
7:          return 1
8:      end if
9:      return n × Factorial(n − 1)
10: end function
```

Memoisation

- a technique to speed up algorithms by keeping track of the result of an expensive operation, and returning just the cached result if the same operation is called again
- lookup table

```
1:  function FACTORIAL(A, n)
2:      n a non-negativ integer
3:      Output: factorial of n
4:      if n = 0 then
5:          return 1
6:      else if n in lookup table then
7:          return lookuptable[n]
8:      else
9:          x ← n × Factorial(n − 1)
10:         store x in lookup table
11:         return x
12:     end if
13: end function
```

Power Function

raising a number a to an arbitrary nonnegative integer, *n*

$p(a,n) = a^n$

$$\mathrm{pow}(a,n) = \begin{cases} 1 & \text{if } n = 0, \\ a\,\mathrm{pow}(a,(n-1)) & \text{if } n > 0 \end{cases}$$

simple recursive definition

this algorithm is O(n) in the exponent *n*

```
Algorithm Power(a, b):
    Input: Two integers a and b
    Output: The value of a to the power b

    power ← 1
    for k=1 to b do
            power ← power * a

return power
```

Power Function

raising a number *x* to an arbitrary nonnegative integer, *n*

$p(x,n) = x^n$

$$\text{pow}(a,n) = \begin{cases} 1 & \text{if } n = 0, \\ a\text{pow}(a,(n-1)/2)^2 & \text{if } n > 0 \quad \text{is odd} \\ \text{pow}(a,n/2)^2 & \text{if} n > 0 \quad \text{is even} \end{cases}$$

each recursive call of function power(*a*,*n*) divides the exponent, *n*, by two

```
Algorithm power (a, n):
    Input: two integers, a and n
    Output: a^n
    if n == 0
        return 1
    if n is odd then:
        y ← power(a, (n-1)/2)
        return a*y*y
    else
        y ← power(a, n/2)
        return y*y
```

Power Function

raising a number *x* to an arbitrary nonnegative integer, *n*

$p(x,n) = x^n$

$$\mathrm{pow}(a,n) = \begin{cases} 1 & \text{if } n = 0, \\ a\,\mathrm{pow}(a,(n-1)/2)^2 & \text{if } n > 0 \quad \text{is odd} \\ \mathrm{pow}(a,n/2)^2 & \text{if } n > 0 \quad \text{is even} \end{cases}$$

there are O(log*n*) recursive calls, not O(*n*)

That is, by using linear recursion and the squaring technique, we reduce the running time for the computation of the power function from *O*(*n*) to *O*(log *n*), which is a big improvement.

```
Algorithm power (a, n):
    Input: two integers, a and n
    Output: a^n
    if n == 0
        return 1
    if n is odd then:
        y ← power(a, (n-1)/2)
        return a*y*y
    else
        y ← power(a, n/2)
        return y*y
```

Summary

- Computing complexity of various algorithms
- different examples of the functional complexity we encounter in this course
- interpreting pseudocode

Attempt the questions in the practical!