Analysis of Algorithms (Complexity)

❖ Performance Analysis

Program run-time is critical for many applications

- finance, robotics, games, database systems, …

Program efficiency can be investigated in two ways:

- measuring the time for the program to run
- analysing the algorithm on which the program is based

Sometimes, the goal is to compare alternative implementations

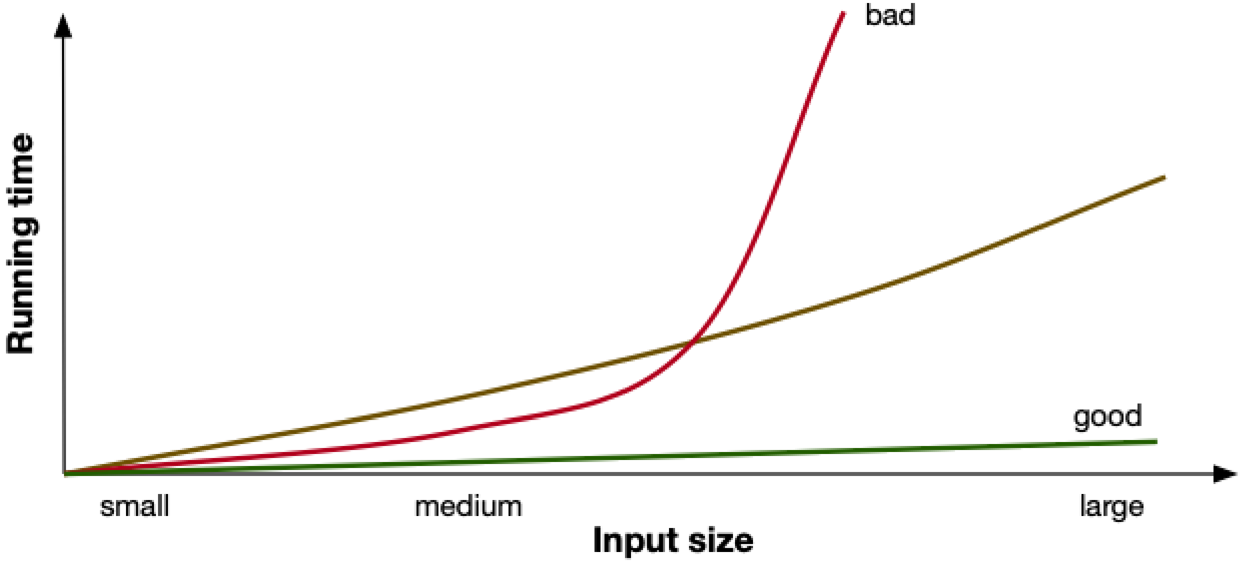
Mostly, the goal is to determine whether ...

- the implemented program is "fast enough"
- a proposed implementation is likely perform well

❖ ... Performance Analysis

Typically: larger input ⇒ longer run time

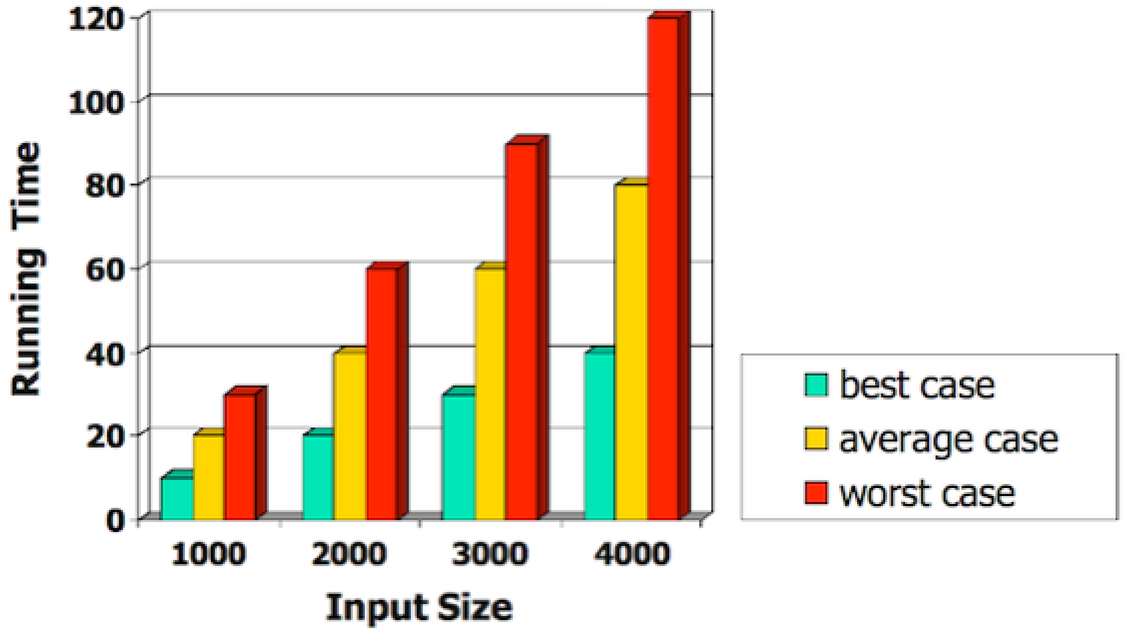
- small inputs ... fast, regardless of algorithm
- medium inputs ... slower, but how much slower?
- large inputs ... slower again, still feasible?



❖ ... Performance Analysis

What to analyse?

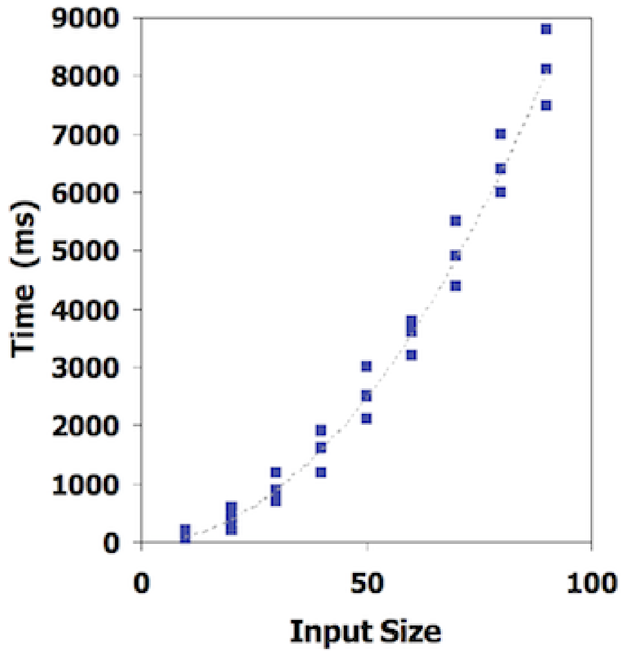
- best-case performance ... not useful unless best case is s-l-o-w
- average-case performance ... difficult; need to know how program used
- worst-case performance ... most important; has observable impacts



❖ Empirical Analysis

Empirical study of *program* performance (measurement)

1. Write program (implement algorithm)
2. Run program with range of data
   (inputs varying in size and composition)
3. Measure the actual running time
4. Plot the results



Measuring run-time on Linux: `time` command (real, user, sys)

❖ ... Empirical Analysis

Limitations:

- requires implementation of algorithm, which may be difficult
- different choice of input data ⇒ different results
- results may not be indicative of running time on other inputs
- timing results affected by run-time enviroment (load)
- in order to compare two *algorithms* ...
  - need "comparable" implementation of each algorithm
  - must use same inputs, same hardware, same O/S, same load

❖ Theoretical Analysis

Formal analysis of *algorithm* performance (complexity)

- uses high-level description of algorithm (pseudocode)
  (no need to write/debug/test an implementation)
- characterises running time as a function of input size, n
- takes into account all possible inputs
- allows us to evaluate the speed of the algorithm
  (independent of the hardware/software environment)

Gives a basis for choosing which algorithm to use in a program.

❖ Pseudocode

Pseudocode is a useful way to describe algorithms

- more structured than English prose
- can capture control structures
- less detailed than a program
- hides program design issues
- high-level ⇒ easy to read/write

❖ ... Pseudocode

Example: **Find** **maximal** **element** in an array

```
arrayMax(A):
|   Input  array A of n integers
|   Output maximum element of A
|
|   currentMax=A[0]
|   for all i=1..n-1 do
|   |   if A[i]>currentMax then
|   |      currentMax=A[i]
|   |   end if
|   end for
|   return currentMax
```

❖ ... Pseudocode

Control flow

- **if** ... **then** ... [**else**] ... **end if**
- **while** .. **do** ... **end while**
  repeat ... **until**
  for [**all**][**each**] .. **do** ... **end for**

Function declaration

- f(arguments):
  Input ...
  Output ...
  ...

Expressions

- = assignment
- = equality testing
- n^2 superscripts and other mathematical formatting allowed
- swap A[i] and A[j] verbal descriptions of *simple* operations allowed

❖ The Abstract RAM Model

RAM = Random Access Machine

- a CPU (central processing unit)
- a potentially unbounded bank of memory cells
  - each of which can hold an arbitrary number, or character
- memory cells are numbered, and accessed via their number
  - accessing any one of them takes CPU time
  - each memory access takes same CPU time

Gives a simple model of computation to use in analyses

❖ Primitive Operations

Every algorithm uses a core set of basic operations

- identifiable in pseudocode
- largely independent of the programming language
- exact definition not important (we will shortly see why)
- assumed to take a constant amount of time in the RAM model

Examples:

- evaluating an expression
- indexing into an array
- calling/returning from a function

❖ Counting Primitive Operations

By inspecting pseudocode ...

- can determine max number of primitive operations executed by algorithm
- as a function of the input size (e.g. #elements in an array)

Example:

```
arrayMax(A):
|   Input   array A of n integers
|   Output maximum element of A
|
|   currentMax=A[0]                    1
|   for all i=1..n-1 do                n+(n-1)
|   |   if A[i]>currentMax then        2(n-1)
|   |      currentMax=A[i]             n-1
|   |   end if
|   end for
|   return currentMax                  1
                                       -------
                               Total   5n-2
```

❖ Estimating Running Times

Algorithm **`arrayMax`** requires ...

- worst case: $5n - 2$ primitive operations
- best case: $4n - 1$ operations (why?)

Define:

- a ... time taken by the fastest primitive operation
- b ... time taken by the slowest primitive operation

Let T(n) be worst-case time of **`arrayMax`**. Then

$$a(5n - 2) \le T(n) \le b(5n - 2)$$

Hence, the running time T(n) is bounded by two linear functions

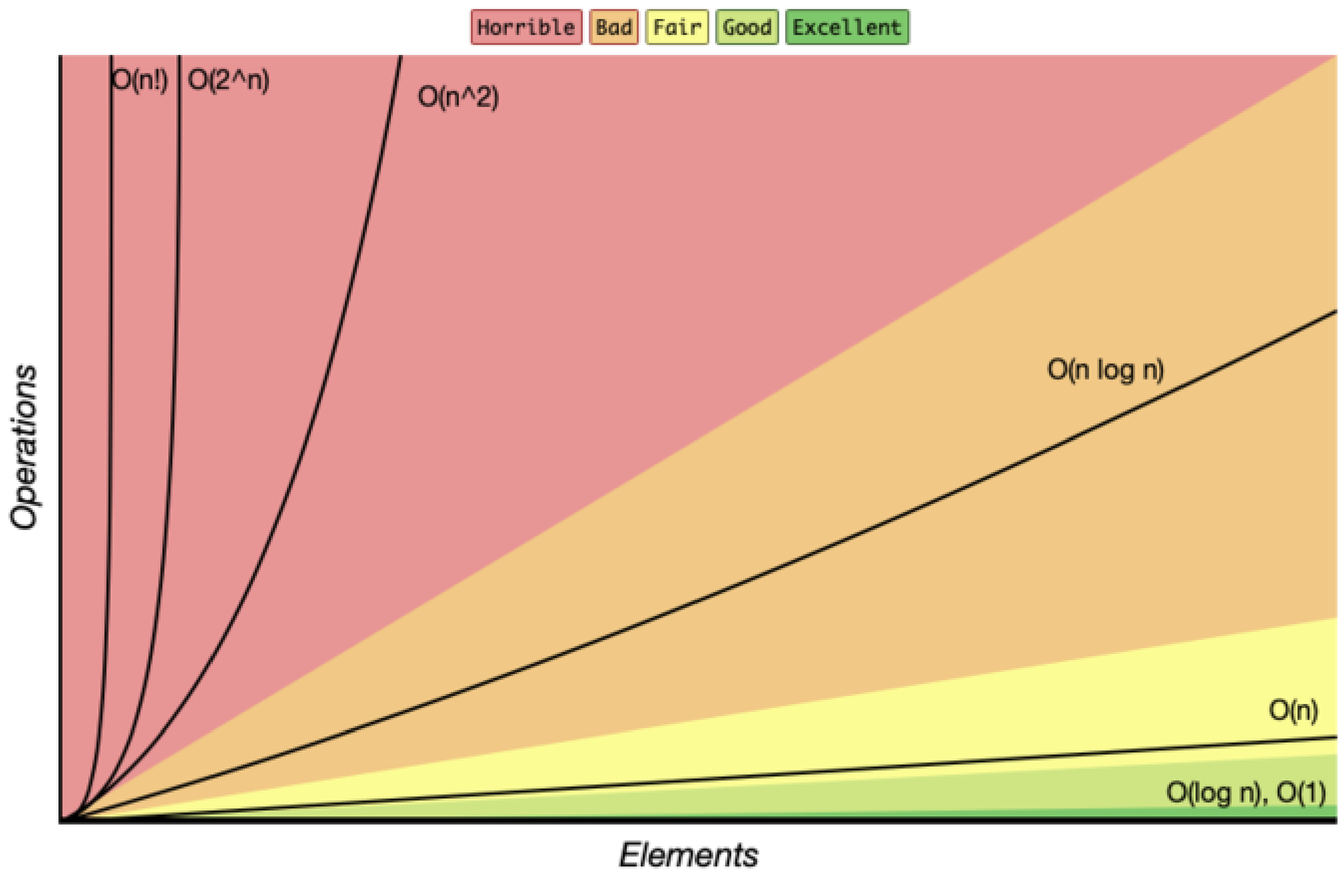
❖ ... Estimating Running Times

Seven commonly encountered functions for algorithm analysis

- Constant $\cong 1$
- Logarithmic $\cong \log n$
- Linear $\cong n$
- N-Log-N $\cong n \log n$
- Quadratic $\cong n^2$
- Cubic $\cong n^3$
- Exponential $\cong 2^n$
- Factorial $\cong$ n!

❖ ... Estimating Running Times

This chart shows various algorithmic growth rates



The lesson: logarithmic complexity is tolerable; exponential is not.

Chart borrowed from: http://bigocheatsheet.com/

❖ ... Estimating Running Times

Growth *rate* is not affected by constant factors or lower-order terms

Examples: $10^2 n + 10^5$ is linear, $10^5 n^2 + 10^8 n$ is quadratic

For **`arrayMax`**, changing the hardware/software environment

- affects T(n) by a constant factor
- but does not alter the growth rate of T(n)

Linear growth rate of T(n)

- is an intrinsic property of the **`arrayMax`** algorithm
- will be the same in any implementation (C, Python, Java, ...)

❖ Example of estimating running times

Determine the number of primitive operations

```
matrixProduct(A,B):
|   Input  n×n matrices A, B
|   Output n×n matrix A·B
|
|   for all i=1..n do
|   |   for all j=1..n do
|   |   |   C[i,j]=0
|   |   |   for all k=1..n do
|   |   |      C[i,j]=C[i,j]+A[i,k]·B[k,j]
|   |   |   end for
|   |   end for
|   end for
|   return C
```

❖ ... Example of estimating running times

```
matrixProduct(A,B):
|  Input  n×n matrices A, B
|  Output n×n matrix A·B
|
|  for all i=1..n do                      2n+1
|  |  for all j=1..n do                   n(2n+1)
|  |  |  C[i,j]=0                         n^2
|  |  |  for all k=1..n do                n^2(2n+1)
|  |  |     C[i,j]=C[i,j]
|  |  |               +A[i,k]·B[k,j]      n^3·5
|  |  |  end for
|  |  end for
|  end for
|  return C                               1
                                          ------------
                                  Total   7n^3+4n^2+3n+2
```

❖ Big-Oh Notation

Given functions f(*n*) and g(*n*), we say that

- f(*n*) is *O(g(n))* (i.e. is *order* g(*n*))

if there are positive constants c and n_0 such that

- $f(n) \le c \cdot g(n) \quad \forall n \ge n_0$

We tend not to use f(*n*) in discussing algorithm complexity.

We quote algorithm complexity as *O(1)* or *O(n)* or *O(n^2)*, etc.

E.g. **`matrixProduct`** has *O(n^3)* complexity.

❖ ... Big-Oh Notation

Example:

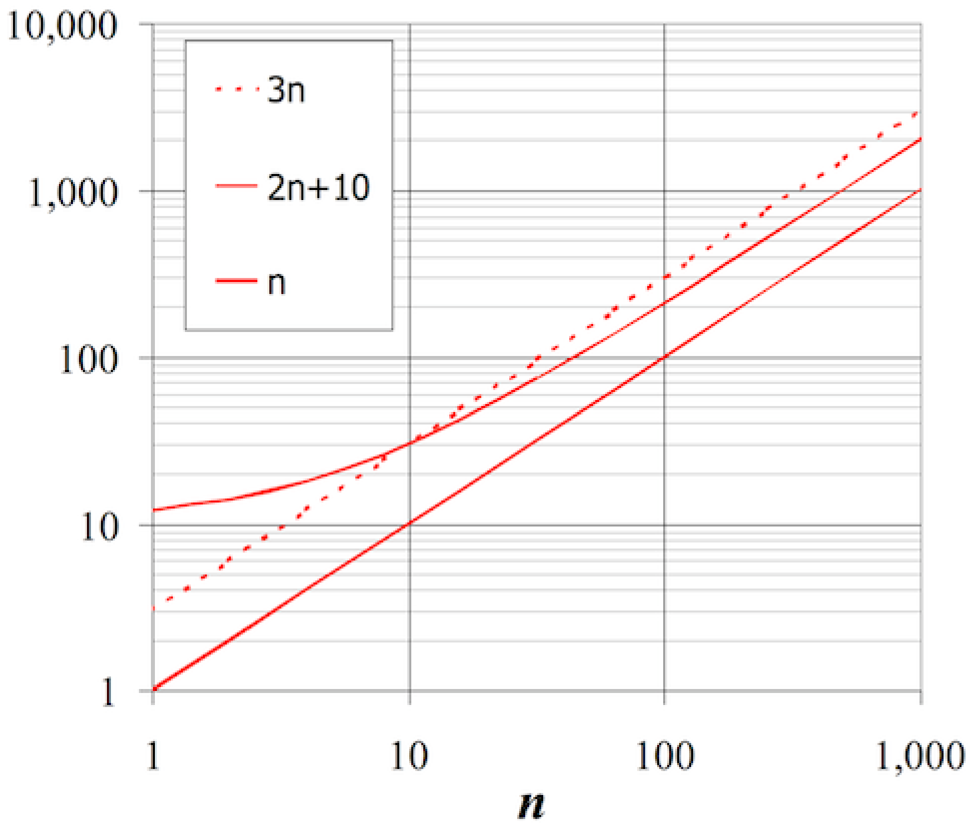
$2n + 10$ is $O(n)$

- $2n+10 \le c \cdot n$

  $\Rightarrow$ $(c-2)n \ge 10$

  $\Rightarrow$ $n \ge 10/(c-2)$

- pick c=3 and n_0=10

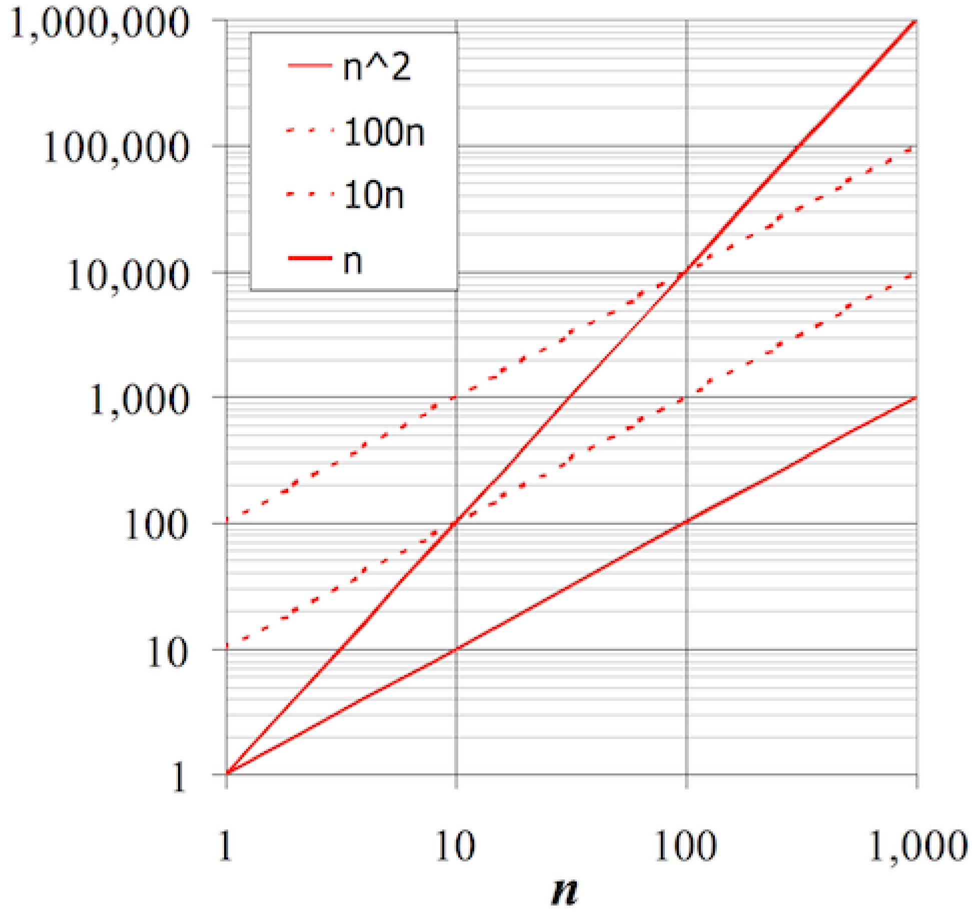


❖ ... Big-Oh Notation

Example:

n^2 is not O(n)

- $n^2 \le c \cdot n$
  $\Rightarrow \; n \le c$
- inequality cannot be satisfied since c must be a constant



❖ Big-Oh and Rate of Growth

Big-Oh gives upper bound on growth rate of a function

- f(n) is O(g(n)) = growth rate of f(n) no more than growth rate of g(n)

Use big-Oh to rank functions according to their rate of growth

	f(n) is O(g(n))	g(n) is O(f(n))
g(n) grows faster	yes	no
f(n) grows faster	no	yes
same order of growth	yes	yes

❖ Big-Oh Rules

How to determine O(*n*) from f(n) ...

- if f(n) is a polynomial of degree d $\Rightarrow$ f(n) is $O(n^d)$
  - lower-order terms are ignored
  - constant factors are ignored
- 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 Analysis of Algorithms

Asymptotic analysis of algorithms

- determines running time in big-Oh notation
- by finding worst-case number of primitive operations
- as a function of input size
- and expressing this function using big-Oh notation

Example:

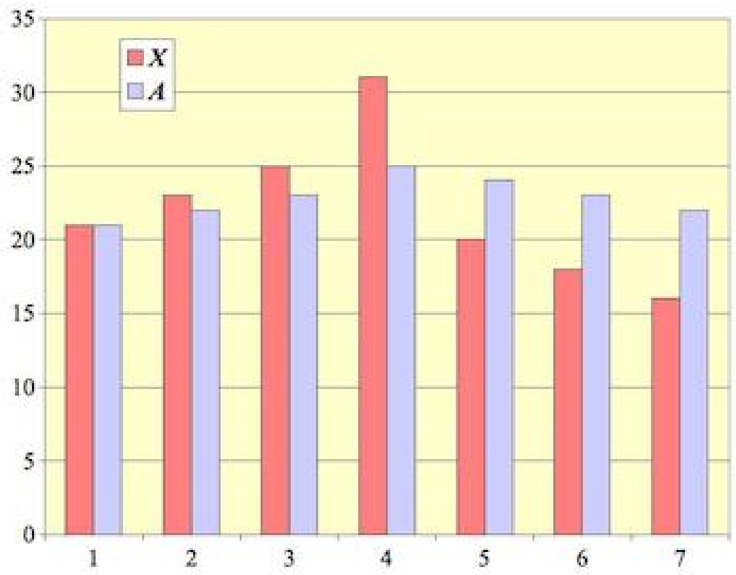
- **`arrayMax`** executes at most 5n – 2 primitive operations
  ⇒ algorithm **`arrayMax`** "runs in O(n) time"

Constant factors and lower-order terms are eventually dropped
⇒ can disregard them when counting primitive operations

❖ Example: Computing Prefix Averages

i-th prefix average of array X is average of first i elements:

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



Note: computing the array A of prefix averages of another array X has applications in e.g. financial analysis

❖ ... Example: Computing Prefix Averages

A quadratic algorithm to compute prefix averages:

```
prefixAverages1(X):
|   Input  array X of n integers
|   Output array A of prefix averages of X
|
|   for all i=0..n-1 do           O(n)
|   |  s=X[0]                     O(n)
|   |  for all j=1..i do          O(n^2)
|   |     s=s+X[j]                O(n^2)
|   |  end for
|   |  A[i]=s/(i+1)               O(n)
|   end for
|   return A                      O(1)
```

Running time cost = $2 \cdot O(n^2) + 3 \cdot O(n) + O(1) = O(n^2)$

⇒ time complexity of algorithm **prefixAverages1** is $O(n^2)$

❖ ... Example: Computing Prefix Averages

The following algorithm computes prefix averages by keeping a running sum:

```
prefixAverages2(X):
|  Input  array X of n integers
|  Output array A of prefix averages of X
|
|  s=0
|  for all i=0..n-1 do          O(n)
|     s=s+X[i]                  O(n)
|     A[i]=s/(i+1)              O(n)
|  end for
|  return A                     O(1)
```

Thus, **prefixAverages2** is O(n)

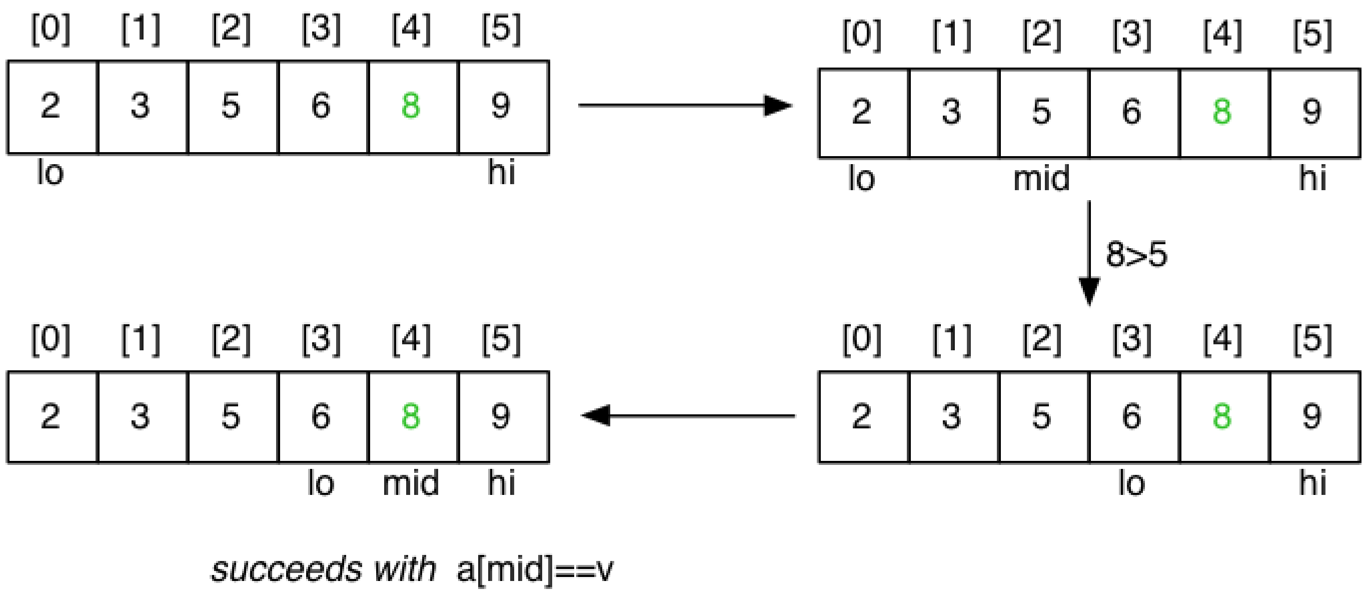
❖ Example: Binary Search

The following recursive algorithm searches for a value in a sorted array:

```
search(v,a,lo,hi):
|  Input  value v
|         array a[lo..hi] of values
|  Output true if v in a[lo..hi]
|         false otherwise
|
|  mid=(lo+hi)/2
|  if lo>hi then return false
|  if a[mid]=v then
|     return true
|  else if a[mid]<v then
|     return search(v,a,mid+1,hi)
|  else
|     return search(v,a,lo,mid-1)
|  end if
```

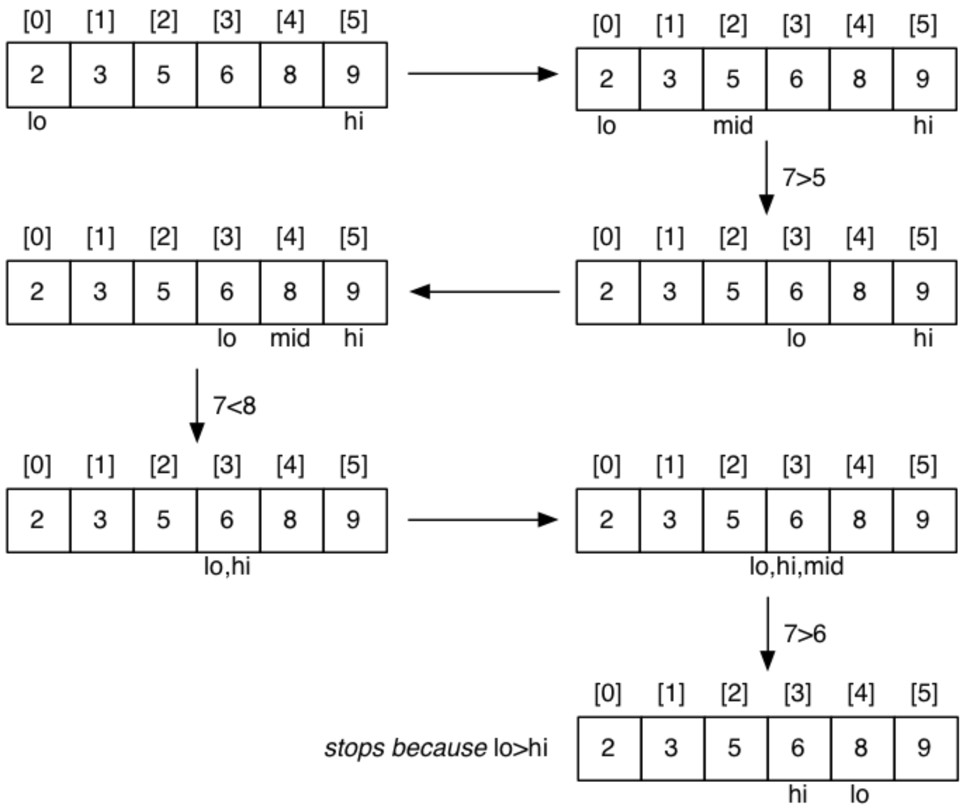
❖ ... Example: Binary Search

Successful search for a value of 8:



❖ ... Example: Binary Search

Unsuccessful search for a value of 7:



❖ ... Example: Binary Search

Cost analysis:

- C_i = #calls to **`search()`** for array of length i
- best case: $C_n = 1$
- for **`a[i..j]`**, **`j<i`** (length=0)
  - $C_0 = 0$
- for **`a[i..j]`**, **`i≤j`** (length=n)
  - $C_n = 1 + C_{n/2} \Rightarrow C_n = \log_2 n$

Thus, binary search is $O(\log_2 n)$ or simply $O(\log n)$

❖ Math Needed for Complexity Analysis

- Summations
- Logarithms
  - $\log_b (xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b x^a = a \log_b x$
  - $\log_b a = \log_x a / \log_x b$
- Exponentials
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c \cdot \log_a b}$

- Proof techniques
- Summation (addition of sequences of numbers)
- Basic probability (for average case analysis, randomised algorithms)

❖ 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 $f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$

big-Theta

- $f(n)$ is $\Theta(g(n))$ if there are constants $c', c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \quad \forall n \geq n_0$

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

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

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

❖ Complexity Classes

Problems in Computer Science …

- some have polynomial worst-case performance (e.g. n^2)
- some have exponential worst-case performance (e.g. 2^n)

Classes of problems:

- P = algorithm can compute answer in polynomial time
- NP = includes problems for which no P algorithm is known

NP stands for "nondeterministic, polynomial time"

❖ ... Complexity Classes

Computer Science jargon for difficulty:

- tractable ... has a polynomial-time algorithm
- intractable ... no tractable algorithm is known
- non-computable ... no algorithm can exist

Programs for intractable problems are only usable for small n

Computational complexity theory deals with degrees of intractability

❖ Generate and Test Algorithms

In scenarios where

- it is simple to test whether a given state is a solution
- it is easy to generate new states (preferably likely solutions)

then a generate and test strategy can be used.

It is necessary that states are generated systematically, so that

- guaranteed to find a solution, or know that none exists

Some randomised algorithms do not require this (more on this later in this course)

❖ ... Generate and Test Algorithms

Simple example: checking whether an integer *n* is prime

- generate/test all possible factors of *n*
- if none of them pass the test ⇒ *n* is prime

Generation is straightforward:

- produce a sequence of all numbers from 2 to *n-1*

Testing is also straightfoward:

- check whether next number divides *n* exactly

❖ ... Generate and Test Algorithms

Function for primality checking:

```
isPrime(n):
|  Input  natural number n
|  Output true if n prime, false otherwise
|
|  for all i=2..n-1 do        // generate
|  |  if n mod i = 0 then     // test
|  |     return false         // i is a divisor => n is not prime
|  |  end if
|  end for
|  return true                // no divisor => n is prime
```

Complexity of **isPrime** is O(n)

Can be optimised: check only numbers between 2 and $\lfloor\sqrt{n}\rfloor$ $\Rightarrow$ O($\sqrt{n}$)

❖ Example: Subset Sum

Problem to solve ...

Is there a subset *S* of these numbers with *sum(S)=1000*?

```
 34,  38,  39,  43,  55,  66,  67,  84,  85,
 91, 101, 117, 128, 138, 165, 168, 169, 182,
184, 186, 234, 238, 241, 276, 279, 288, 386,
387, 388, 389
```

General problem:

- given a set of *n* integers and a target sum *k*
- is there a subset that adds up to exactly *k*?

❖ ... Example: Subset Sum

Generate and test approach:

```
subsetsum(A,n,k):
|  Input  set A of n integers, target sum k
|  Output true if Σ_{b∈B}b=k for some B⊆A, false otherwise
|
|  for each subset S⊆A do
|  |  if sum(S)=k then
|  |     return true
|  |  end if
|  end for
|  return false
```

How many subsets are there of *n* elements?

How could we generate them?

❖ ... Example: Subset Sum

Given: a set of **`n`** distinct integers in an array **`A`** ...

- produce all subsets of these integers

A method to generate subsets:

- represent sets as n bits (e.g. $n=4$, **`0000`**, **`0011`**, **`1111`** etc.)
- bit i represents the i^{th} input number
- if bit i is set to 1, then **`A[i]`** is in the subset
- if bit i is set to 0, then **`A[i]`** is not in the subset
- e.g. if **`A[]=={1,2,3,5}`** then **`0011`** represents **`{1,2}`**

❖ ... Example: Subset Sum

Algorithm:

```
subsetsum1(A,n,k):
|  Input  set A of n integers, target sum k
|  Output true if Σ_b∈B b=k for some B⊆A, false otherwise
|
|  for s=0..2^n-1 do
|  |  if k = Σ_(i^th bit of s is 1) A[i] then
|  |     return true
|  |  end if
|  end for
|  return false
```

Obviously, **subsetsum1** is $O(2^n)$

❖ ... Example: Subset Sum

Alternative approach ...

`subsetsum2(A,n,k)`

- if the n^{th} value A[n-1] is part of a solution ...
  - then the first n-1 values must sum to k – A[n-1]
- if the n^{th} value is not part of a solution ...
  - then the first n-1 values must sum to k
- base cases: k=0 (solved by {}); n=0 (unsolvable if k>0)

❖ ... Example: Subset Sum

Algorithm:

```
subsetsum2(A,n,k):
|  Input  array A, index n, target sum k
|  Output true if Σ_{b∈B}b=k, B=A[0..n-1], false otherwise
|
|  if k=0 then
|     return true    // empty set solves this
|  else if n=0 then
|     return false   // no elements => no sums
|  else
|     return subsetsum2(A,n-1,k-A[n-1])
|                ∨ subsetsum2(A,n-1,k)
|  end if
```

❖ ... Example: Subset Sum

Cost analysis:

- C_i = #calls to **`subsetsum2()`** for array of length i
- for best case, $C_n = C_{n-1}$ (why?)
- for worst case, $C_n = 2 \cdot C_{n-1} \Rightarrow C_n = 2^n$

Thus, **`subsetsum2`** also is $O(2^n)$

Subset Sum is a member of the class of *NP*-complete problems

- intractable … known algorithms have exponential performance
- increase input size by 1 ⇒ double the execution time
- increase input size by 100, ⇒ takes 2^{100} times as long to execute
(note: 2^{100} = = 1,267,650,600,228,229,401,496,703,205,376)

❖ Summary

- Big-Oh notation
- Asymptotic analysis of algorithms
- Examples of algorithms from various complexity classes

  logarithmic, linear, polynomial, exponential complexity

- Suggested reading:
  - Sedgewick, Ch.2.1-2.4,2.6

Here we considered only running time; memory space usage is also important

Produced: 4 Jun 2020