

Analysis of Algorithms

Running Time

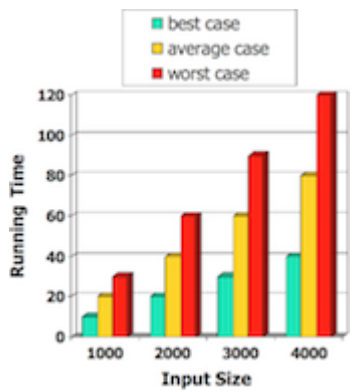
2/64

An **algorithm** is a step-by-step procedure

- for solving a problem
- in a finite amount of time

Most algorithms map input to output

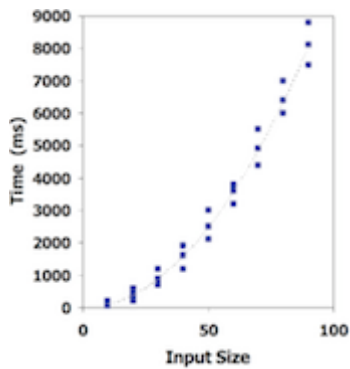
- running time typically grows with input size
- *average time* often difficult to determine
- Focus on *worst case* running time
 - easier to analyse
 - crucial to many applications: finance, robotics, games, ...



Empirical Analysis

3/64

1. Write program that implements an algorithm
2. Run program with inputs of varying size and composition
3. Measure the actual running time
4. Plot the results



... Empirical Analysis

4/64

Limitations:

- requires to implement the algorithm, which may be difficult
- results may not be indicative of running time on other inputs
- same hardware and operating system must be used in order to compare two algorithms

- Uses high-level description of the algorithm instead of implementation ("pseudocode")
 - Characterises running time as a function of the input size, n
 - Takes into account all possible inputs
 - Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
-

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
-

... Pseudocode

- More structured than English prose
 - Less detailed than a program
 - Preferred notation for describing algorithms
 - Hides program design issues
-

Exercise #1: Pseudocode

Formulate the following verbal description in pseudocode:

In the first phase, we iteratively pop all the elements from stack S and enqueue them in queue Q, then dequeue the element from Q and push them back onto S.

As a result, all the elements are now in reversed order on S.

In the second phase, we again pop all the elements from S, but this time we also look for the element x.

By again passing the elements through Q and back onto S, we reverse the reversal, thereby restoring the original order of the elements on S.

Sample solution:

```
while not empty(S) do
    pop e from S, enqueue e into Q
end while
while not empty(Q) do
    dequeue e from Q, push e onto S
end while
found=false
while not empty(S) do
    pop e from S, enqueue e into Q
    if e=x then
        found=true
    end if
end while
while not empty(Q) do
    dequeue e from Q, push e onto S
end while
```

Exercise #2: Pseudocode

Implement the following pseudocode instructions in C

1. A is an array of ints

```
...
swap A[i] and A[j]
...
```

2. head points to beginning of linked list

```
...
swap head and head->next
...
```

3. S is a stack

```
...
swap the top two elements on S
...
```

```
1. int temp = A[i];
   A[i] = A[j];
   A[j] = temp;

2. NodeT *succ = head->next;
   head->next = succ->next;
```

```

succ->next = head;
head = succ;

3. x = StackPop(S);
   y = StackPop(S);
   StackPush(S, x);
   StackPush(S, y);

```

The following pseudocode instruction is problematic. Why?

```

...
swap the two elements at the front of queue Q
...

```

The Abstract RAM Model

13/64

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 accessing any one of them takes CPU time

Primitive Operations

14/64

- Basic computations performed by an algorithm
- 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

15/64

By inspecting the pseudocode ...

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

Example:

```

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

```

```

1
n+(n-1)
2(n-1)
n-1

```

```
|   return currentMax
```

$\frac{1}{5n-2}$
Total

Estimating Running Times
16/64

Algorithm `arrayMax` requires $5n - 2$ primitive operations in the *worst* case

- best* case requires $4n - 1$ 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

$$a \cdot (5n - 2) \leq T(n) \leq b \cdot (5n - 2)$$

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

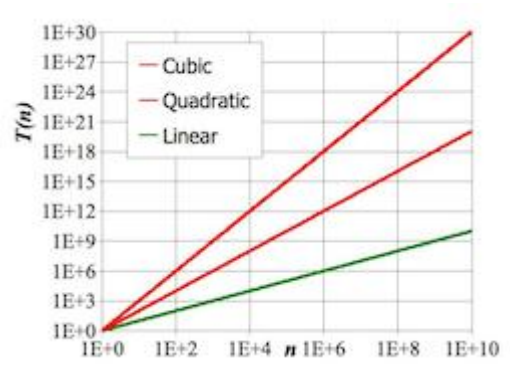
... Estimating Running Times
17/64

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 $\cong 2^n$

... Estimating Running Times
18/64

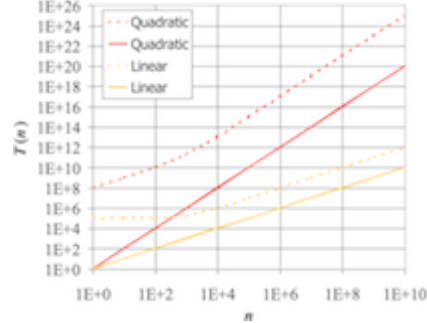
In a log-log chart, the slope of the line corresponds to the growth rate of the function



... Estimating Running Times
19/64

The growth rate is not affected by constant factors or lower-order terms

- Examples:
 - $10^2n + 10^5$ is a linear function
 - $10^5n^2 + 10^8n$ is a quadratic function



... Estimating Running Times

20/64

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 the running time $T(n)$ is an intrinsic property of algorithm `arrayMax`

Exercise #3: Estimating running times

21/64

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
```

```
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
```

	2n+1
	n(2n+1)
	n ²
	n ² (2n+1)
	n ³ ·5
	1

Total	7n ³ +4n ² +3n+2

Big-Oh

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

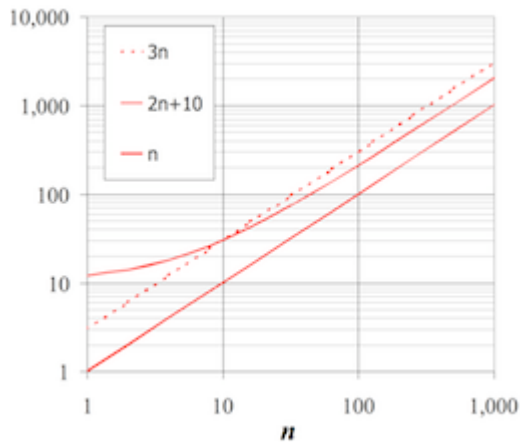
$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

... Big-Oh Notation

25/64

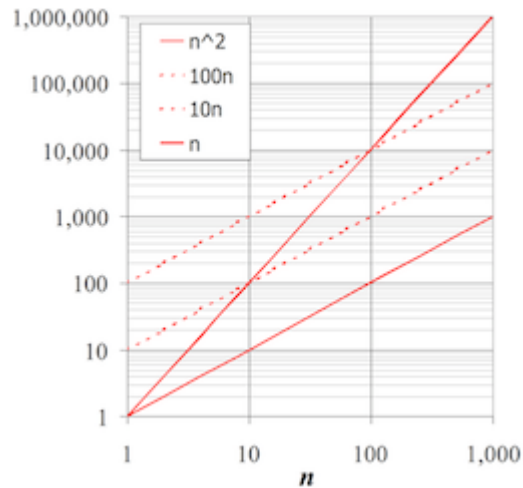
Example: function $2n + 10$ is $O(n)$

- $2n + 10 \leq c \cdot n$
 $\Rightarrow (c - 2)n \geq 10$
 $\Rightarrow n \geq 10 / (c - 2)$
- pick $c = 3$ and $n_0 = 10$



... Big-Oh Notation

26/64



Example: function n^2 is not $O(n)$

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

Exercise #4: Big-Oh

27/64

Show that

1. $7n - 2$ is $O(n)$
2. $3n^3 + 20n^2 + 5$ is $O(n^3)$
3. $3 \cdot \log n + 5$ is $O(\log n)$

1. $7n-2$ is $O(n)$
need $c>0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
 \Rightarrow true for $c=7$ and $n_0=1$
2. $3n^3 + 20n^2 + 5$ is $O(n^3)$
need $c>0$ and $n_0 \geq 1$ such that $3n^3+20n^2+5 \leq c \cdot n^3$ for $n \geq n_0$
 \Rightarrow true for $c=4$ and $n_0=21$
3. $3 \cdot \log n + 5$ is $O(\log n)$
need $c>0$ and $n_0 \geq 1$ such that $3 \cdot \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
 \Rightarrow true for $c=8$ and $n_0=2$

Big-Oh and Rate of Growth

29/64

- Big-Oh notation gives an upper bound on the growth rate of a function
 - "f(n) is $O(g(n))$ " means 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

30/64

- 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 $O(3n)$ "

Exercise #5: Big-Oh

31/64

Show that $\sum_{i=1}^n i$ is $O(n^2)$

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

which is $O(n^2)$

Asymptotic Analysis of Algorithms

33/64

Asymptotic analysis of algorithms determines running time in big-Oh notation:

- find worst-case number of primitive operations as a function of input size
- express this function using big-Oh notation

Example:

- algorithm `arrayMax` executes at most $5n - 2$ primitive operations
 \Rightarrow algorithm `arrayMax` "runs in $O(n)$ time"

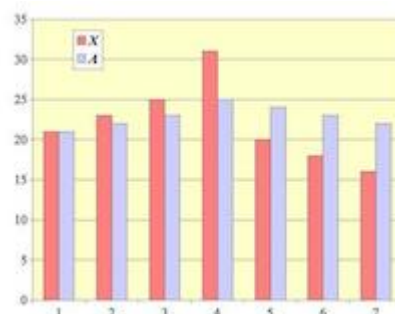
Constant factors and lower-order terms eventually dropped
 \Rightarrow can disregard them when counting primitive operations

Example: Computing Prefix Averages

34/64

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

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



NB. computing the array A of prefix averages of another array X has applications in financial analysis

... Example: Computing Prefix Averages

35/64

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(n²)
|   |   |   s=s+X[j]             O(n²)
|   |   end for
|   |   A[i]=s/(i+1)             O(n)
|   end for
|   return A                     O(1)

```

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

\Rightarrow Time complexity of algorithm `prefixAverages1` is $O(n^2)$

... Example: Computing Prefix Averages

36/64

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

37/64

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

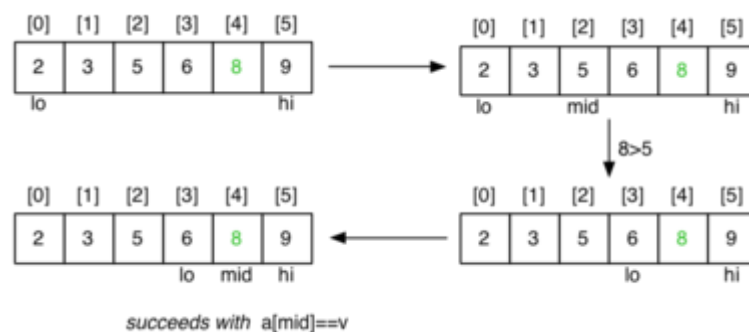
    mid=(lo+hi)/2
    if lo>hi then return false
    if a[mid]=v then
        return true
    else if a[mid]<v then
        return search(v,a,mid+1,hi)
    else
        return search(v,a,lo,mid-1)
    end if

```

... Example: Binary Search

38/64

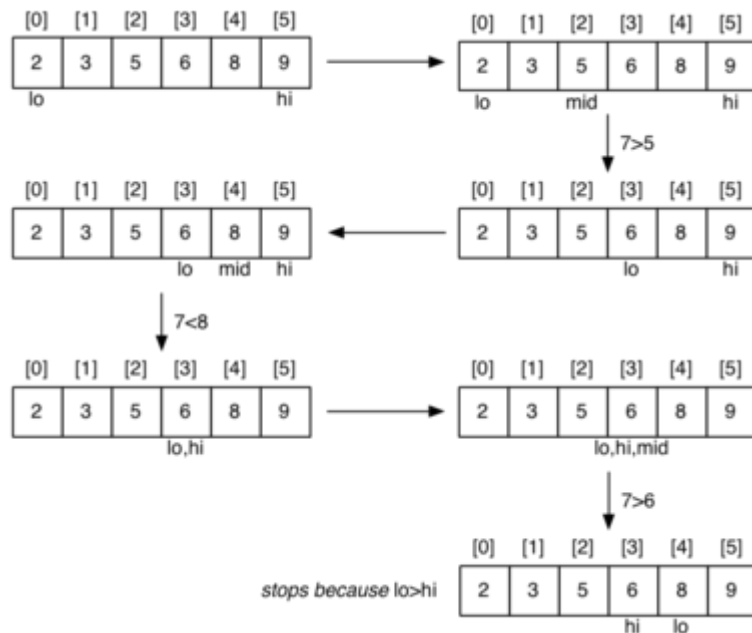
Successful search for a value of 8:



... Example: Binary Search

39/64

Unsuccessful search for a value of 7:



... Example: Binary Search

40/64

Cost analysis:

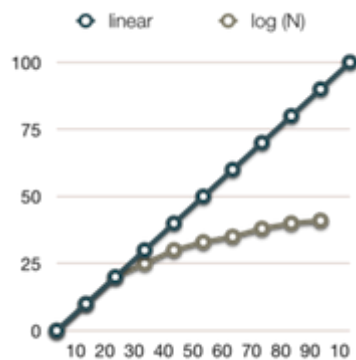
- $C_i = \# \text{calls to } \text{search}() \text{ for array of length } i$
- for best case, $C_n = 1$
- for $a[i..j]$, $j < i$ (length=0)
 - $C_0 = 0$
- for $a[i..j]$, $i \leq j$ (length=n)
 - $C_n = 1 + C_{n/2} \Rightarrow C_n = \log_2 n$

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

... Example: Binary Search

41/64

Why logarithmic complexity is good:



Math Needed for Complexity Analysis

42/64

- 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)
-

Exercise #6: Analysis of Algorithms

43/64

What is the complexity of the following algorithm?

```
splitList(L):
    Input   non-empty linked list L
    Output L split into two halves

    // use slow and fast pointer to traverse L
    slow=head(L), fast=head(L).next
    while fast≠NULL and fast.next≠NULL do
        slow=slow.next, fast=fast.next.next // advance pointers
    end while
    cut L between slow and slow.next
```

Answer: $O(l)$

Exercise #7: Analysis of Algorithms

45/64

What is the complexity of the following algorithm?

```
binaryConversion(n):
    Input   positive integer n
    Output binary representation of n on a stack

    create empty stack S
    while n>0 do
        | push (n mod 2) onto S
        | n=[n/2]
    end while
    return S
```

Assume that creating a stack and pushing an element both are $O(1)$ operations ("constant")

Answer: $O(\log n)$

Relatives of Big-Oh

47/64

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

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

... Relatives of Big-Oh

48/64

- $f(n)$ 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 $g(n)$

... Relatives of Big-Oh

49/64

Examples:

- $\frac{1}{4}n^2$ is $\Omega(n^2)$
 - need $c > 0$ and $n_0 \geq 1$ such that $\frac{1}{4}n^2 \geq c \cdot n^2$ for $n \geq n_0$
 - let $c = \frac{1}{4}$ and $n_0 = 1$
- $\frac{1}{4}n^2$ is $\Omega(n)$
 - need $c > 0$ and $n_0 \geq 1$ such that $\frac{1}{4}n^2 \geq c \cdot n$ for $n \geq n_0$
 - let $c = 1$ and $n_0 = 2$
- $\frac{1}{4}n^2$ is $\Theta(n^2)$
 - since $\frac{1}{4}n^2$ is in $\Omega(n^2)$ and $O(n^2)$

Complexity Classes

50/64

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

Beware: NP stands for "nondeterministic, polynomial time (on a theoretical *Turing Machine*)"

... Complexity Classes

51/64

Computer Science jargon for difficulty:

- tractable ... have a polynomial-time algorithm (useful in practice)
- intractable ... no tractable algorithm is known (feasible only for small n)
- non-computable ... no algorithm can exist

Computational complexity theory deals with different degrees of intractability

Generate and Test Algorithms

Generate and Test

53/64

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 we are guaranteed to find a solution, or know that none exists
 - some **randomised** algorithms do not require this, however (more on this later in this course)

... Generate and Test

54/64

Simple example: checking whether an integer n is prime

- generate/test all possible factors of n
- if none of them pass the test $\Rightarrow n$ is prime

Generation is straightforward:

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

Testing is also straightforward:

- check whether next number divides n exactly

... Generate and Test

55/64

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

56/64

Problem to solve ...

Is there a subset S of these numbers with $\text{sum}(S)=1000$?

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

Generate and test approach:

```
subsetsum(A,k):
|   Input   set A of n integers, target sum k
|   Output true if  $\sum_{b \in B} b = k$  for some  $B \subseteq A$ 
|           false otherwise
|
|   for each subset  $S \subseteq A$  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?

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 $\{1, 2\}$

Algorithm:

```
subsetsum1(A,k):
|   Input   set A of n integers, target sum k
|   Output true if  $\sum_{b \in B} b = k$  for some  $B \subseteq A$ 
|           false otherwise
|
|   for  $s=0 \dots 2^n-1$  do
|   |   if  $k = \sum_{(i^{th} \text{ bit of } s \text{ is } 1)} A[i]$  then
|   |   |   return true
|   |   end if
|   end for
|   return false
```

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

Alternative approach ...

`subsetsum2(A,n,k)`
(returns true if any subset of $A[0..n-1]$ sums to k ; returns false otherwise)

- 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 $k>0$)

```
subsetsum2(A,n,k):
    Input array A, index n, target sum k
    Output true if some subset of A[0..n-1] sums up to k
           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 subsetsum(A,n-1,k-A[n-1]) or subsetsum(A,n-1,k)
    end if
```

... Example: Subset Sum

61/64

Cost analysis:

- $C_i = \text{\#calls to } \text{subsetsum2} () \text{ for array of length } i$
- 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 $O(2^n)$

... Example: Subset Sum

62/64

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

- intractable ... only algorithms with exponential performance are known
 - increase input size by 1, double the execution time
 - increase input size by 100, it takes $2^{100} = 1,267,650,600,228,229,401,496,703,205,376$ times as long to execute
- but if you can find a polynomial algorithm for Subset Sum, then any other *NP*-complete problem becomes *P* ...

Tips for Week 5 Problem Set

63/64

Main theme: *Time complexity analysis*

- Demonstrate your mathematical understanding of Big-Oh
- Count primitive operations to determine time complexity
- Learn how to develop algorithms in pseudocode
 - ... before implementing them

```
prompt$ ./palindrome racecar
yes
prompt$ ./palindrome reviewer
no
```


- Extra challenge for the Challenge Exercise: can you find a *linear time* solution?
-

Summary

64/64

- Big-Oh notation
 - Asymptotic analysis of algorithms
 - Examples of algorithms with logarithmic, linear, polynomial, exponential complexity
-
- Suggested reading:
 - Sedgewick, Ch.2.1-2.4,2.6
-