

Week 01b: Analysis of Algorithms

Analysis of Algorithms

Running Time

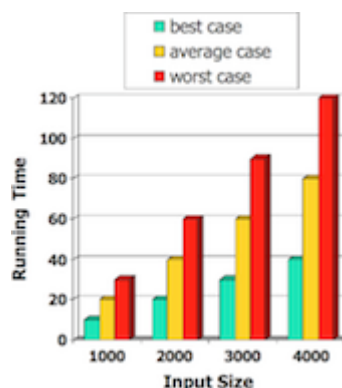
2/87

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

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

Most algorithms map input to output

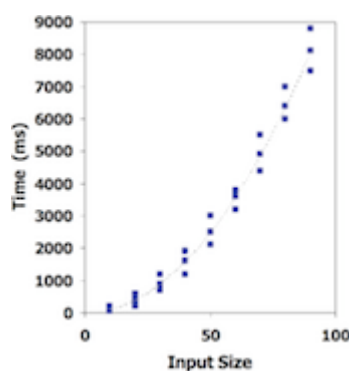
- running time typically grows with input size
- *average time* often difficult to determine
- Focus on *worst case* running time
 - easier to analyse
 - crucial to many applications: finance, robotics, games, ...



Empirical Analysis

3/87

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



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

Theoretical Analysis

4/87

- 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

5/87

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

6/87

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

7/87

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

Exercise #1: Pseudocode

8/87

Formulate the following verbal description in pseudocode:

To reverse the order of the elements on a stack S with the help of a queue:

- 1. In the first phase, pop one element after the other from S and enqueue it in queue Q until the stack is empty.*
- 2. In the second phase, iteratively dequeue all the elements from Q and push them onto the stack.*

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

Sample solution:

```

while S is not empty do
    pop e from S, enqueue e into Q
end while
while Q is not empty do
    dequeue e from Q, push e onto S
end while

```

Exercise #2: Pseudocode

10/87

Implement the following pseudocode instructions in C

1. A is an array of ints

```

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

```

2. S is a stack

```

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

```

```
1. int temp = A[i];  
   A[i] = A[j];  
   A[j] = temp;  
  
2. 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

12/87

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

13/87

- 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

14/87

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
  return currentMax

```

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

Total	$5n-2$

Estimating Running Times

15/87

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

16/87

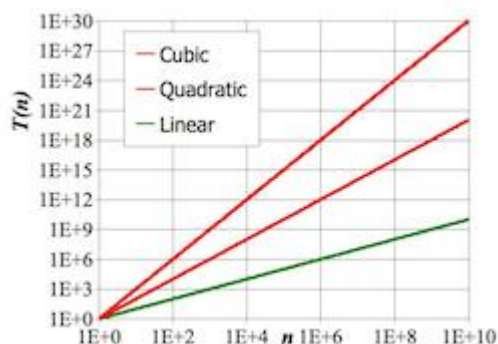
Seven commonly encountered functions for algorithm analysis

- Constant ≈ 1
- Logarithmic $\approx \log n$
- Linear $\approx n$
- N-Log-N $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Exponential $\approx 2^n$

... Estimating Running Times

17/87

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

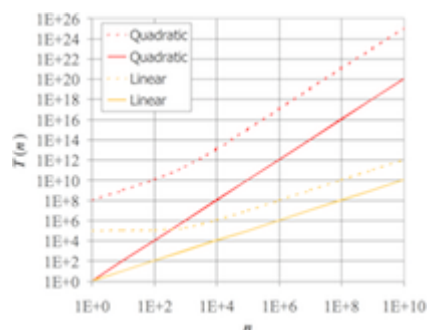


... Estimating Running Times

18/87

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

19/87

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

20/87

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

```

```

|   |   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
	$n^2(2n+1)$
	$n^3 \cdot 4$
	1

Total	$6n^3+4n^2+3n+2$

Big-Oh

Big-Oh Notation

23/87

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

$$f(n) \in O(g(n))$$

if there are positive constants c and n_0 such that

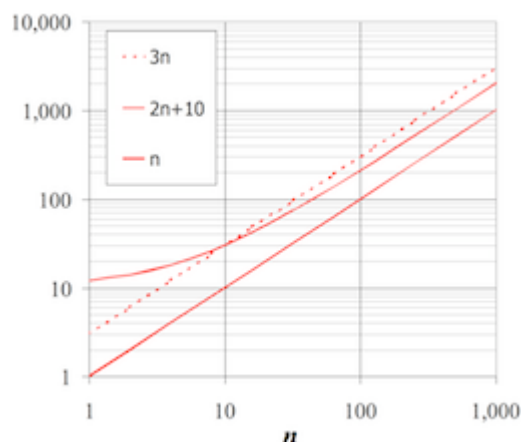
$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

Hence: $O(g(n))$ is the set of all functions that do not grow faster than $g(n)$

... Big-Oh Notation

24/87

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

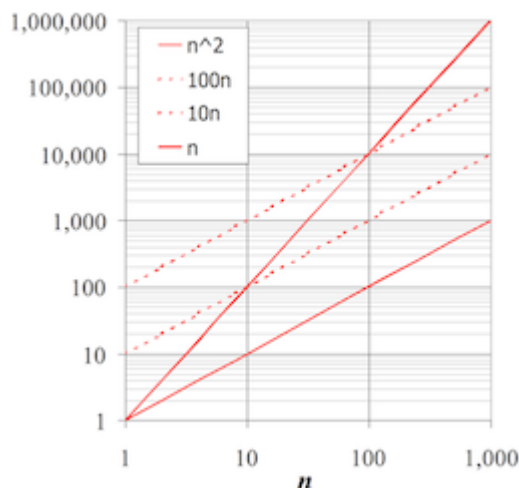


- $2n+10 \leq c \cdot n$
 $\Rightarrow (c-2)n \geq 10$
 $\Rightarrow n \geq 10/(c-2)$
- pick $c=3$ and $n_0=10$

... Big-Oh Notation

25/87

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



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

Exercise #4: Big-Oh

26/87

Show that

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

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

28/87

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

	$f(n) \in O(g(n))$	$g(n) \in O(f(n))$
$g(n)$ grows faster	yes	no
$f(n)$ grows faster	no	yes
same order of growth	yes	yes

Big-Oh Rules

29/87

- 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 $O(n^2)$ "
 - but keep in mind that, $2n$ is in $O(n^2)$, $O(n^3)$, ...
- Use the simplest expression of the class
 - say " $3n + 5$ is $O(n)$ " instead of " $3n + 5$ is $O(3n)$ "

Exercise #5: Big-Oh

30/87

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

32/87

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

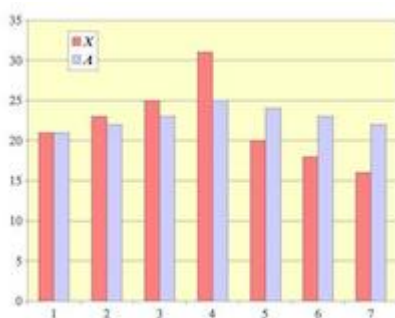
\Rightarrow can disregard them when counting primitive operations

Example: Computing Prefix Averages

33/87

- 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

34/87

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
        s=X[0]
        for all j=1..i do
            s=s+X[j]
        end for
        A[i]=s/(i+1)
    end for
    return A
  
```

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

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

... Example: Computing Prefix Averages

35/87

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

36/87

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

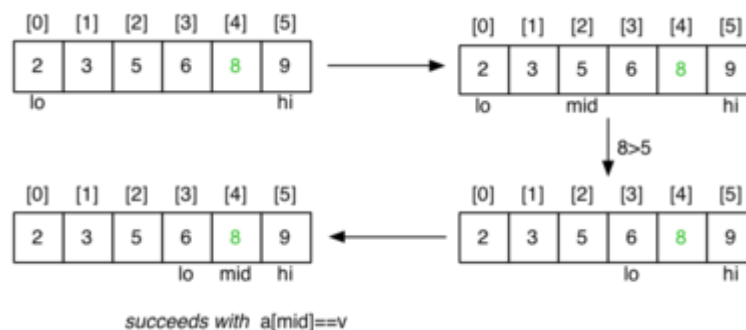
  mid=(lo+hi)/2
  if lo>hi then return false
  if a[mid]=v then
    return true
  else if a[mid]<v then
    return search(v,a,mid+1,hi)
  else
    return search(v,a,lo,mid-1)
  end if

```

... Example: Binary Search

37/87

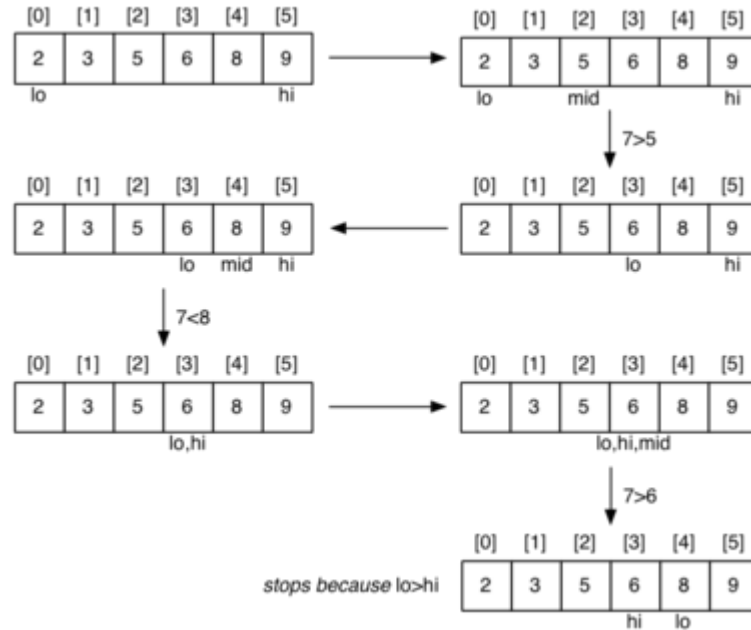
Successful search for a value of 8:



... Example: Binary Search

38/87

Unsuccessful search for a value of 7:



... Example: Binary Search

39/87

Cost analysis:

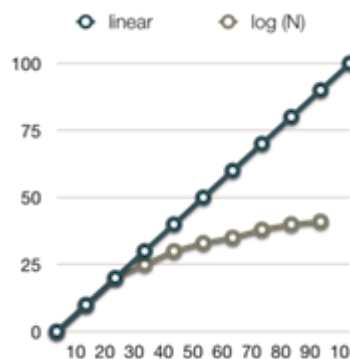
- $C_i = \text{\#calls to search}()$ for array of length i
- for best case, $C_n = 1$
- for $a[i..j]$, $j < i$ (length=0)
 - $C_0 = 0$
- for $a[i..j]$, $i \leq j$ (length=n)
 - $C_n = 1 + C_{n/2} \Rightarrow C_n = \log_2 n$

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

... Example: Binary Search

40/87

Why logarithmic complexity is good:



Math Needed for Complexity Analysis

41/87

- 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

42/87

What is the complexity of the following algorithm?

```
enqueue(Q, Elem):
|   Input   queue Q, element Elem
|   Output  Q with Elem added at the end
|
|   Q.top=Q.top+1
|   for all i=Q.top down to 1 do
|       Q[i]=Q[i-1]
|   end for
|   Q[0]=Elem
|   return Q
```

Answer: $O(|Q|)$

Exercise #7: Analysis of Algorithms

44/87

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

46/87

big-Omega

- $f(n) \in \Omega(g(n))$ 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

- $f(n) \in \Theta(g(n))$ 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

47/87

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

... Relatives of Big-Oh

48/87

Examples:

- $\frac{1}{4}n^2 \in \Omega(n^2)$
 - 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 $n_0 = 1$
- $\frac{1}{4}n^2 \in \Omega(n)$
 - 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 $n_0 = 2$
- $\frac{1}{4}n^2 \in \Theta(n^2)$
 - since $\frac{1}{4}n^2$ belongs to $\Omega(n^2)$ and $O(n^2)$

Complexity Analysis: Arrays vs. Linked Lists

Static/Dynamic Sequences

50/87

Previously we have used an *array* to implement a stack

- fixed size collection of heterogeneous elements
- can be accessed via index or via "moving" pointer

The "fixed size" aspect is a potential problem:

- how big to make the (dynamic) array? (big ... just in case)
- what to do if it fills up?

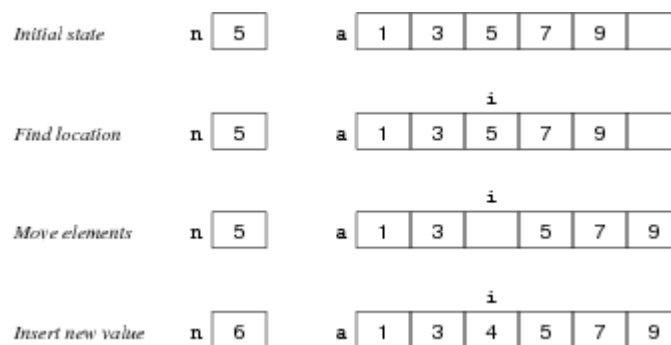
The rigid sequence is another problems:

- inserting/deleting an item in middle of array

... Static/Dynamic Sequences

51/87

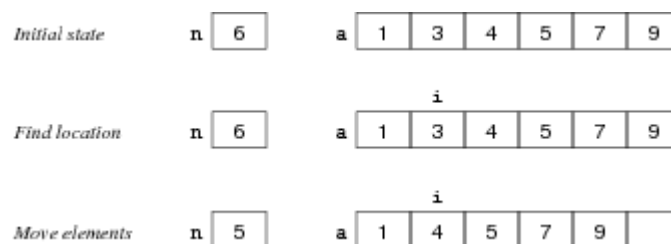
Inserting a value (4) into a sorted array a with n elements:



... Static/Dynamic Sequences

52/87

Deleting a value (3) from a sorted array a with n elements:

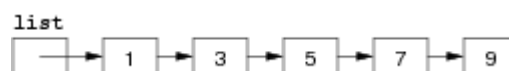


... Static/Dynamic Sequences

53/87

The problems with using arrays can be solved by

- allocating elements individually
- linking them together as a "chain"



Benefits:

- insertion/deletion have minimal effect on list overall
- only use as much space as needed for values

Self-referential Structures

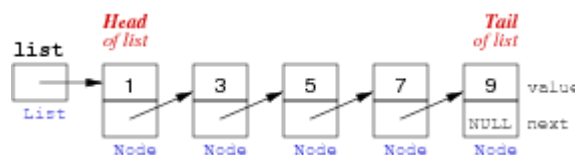
54/87

To realise a "chain of elements", need a *node* containing

- a value
- a link to the next node

To represent a chained (linked) *list* of nodes:

- we need a *pointer* to the first node
- each node contains a pointer to the next node
- the next pointer in the last node is NULL



... Self-referential Structures

55/87

Linked lists are more flexible than arrays:

- values do not have to be adjacent in memory
- values can be rearranged simply by altering pointers
- the number of values can change dynamically
- values can be added or removed in any order

Disadvantages:

- it is not difficult to get pointer manipulations wrong
- each value also requires storage for next pointer

... Self-referential Structures

56/87

Create a new list node:

```

makeNode(v)
| Input   value v
| Output new linked list node with value v
|
| new.value=v           // initialise data
| new.next=NULL        // initialise link to next node
| return new           // return pointer to new node
  
```

Exercise #8: Creating a Linked List

57/87

Write pseudocode to create a linked list of three nodes with values 1, 42 and 9024.


```

mylist=makeNode(1)
mylist.next=makeNode(42)
(mylist.next).next=makeNode(9024)

```

Iteration over Linked Lists

59/87

When manipulating list elements

- typically have pointer `p` to current node
- to access the data in current node: `p.value`
- to get pointer to next node: `p.next`

To iterate over a linked list:

- set `p` to point at first node (head)
- examine node pointed to by `p`
- change `p` to point to next node
- stop when `p` reaches end of list (NULL)

... Iteration over Linked Lists

60/87

Standard method for scanning all elements in a linked list:

```

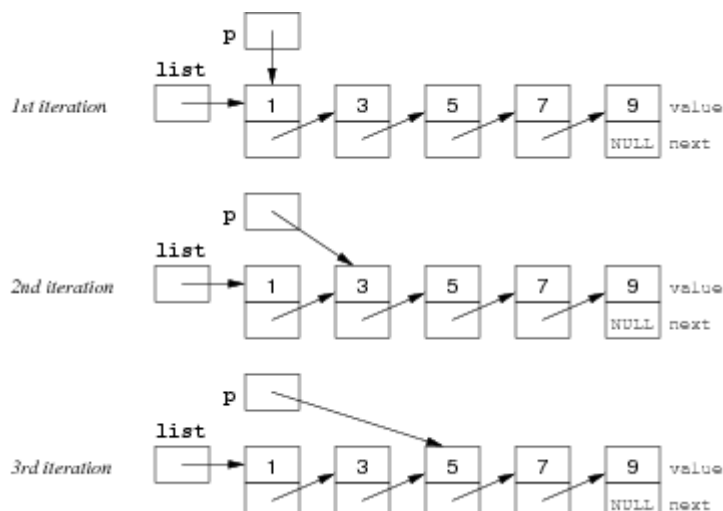
list // pointer to first Node in list
p    // pointer to "current" Node in list

p=list
while p≠NULL do
|   ... do something with p.value ...
|   p=p.next
end while

```

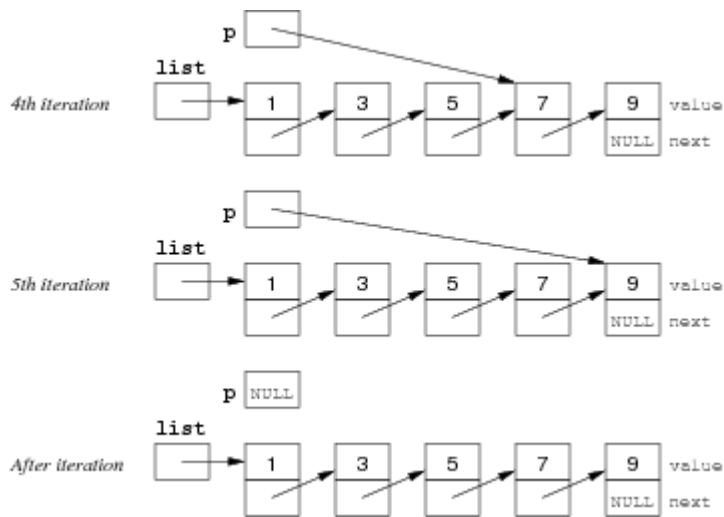
... Iteration over Linked Lists

61/87



... Iteration over Linked Lists

62/87



... Iteration over Linked Lists

63/87

Check if list contains an element:

```

inLL(L,d):
    Input  linked list L, value d
    Output true if d in list, false otherwise

    p=L
    while p≠NULL do
        if p.value=d then    // element found
            return true
        end if
        p=p.next
    end while
    return false            // element not in list
  
```

Time complexity: $O(|L|)$

... Iteration over Linked Lists

64/87

Print all elements:

```

showLL(L):
    Input linked list L

    p=L
    while p≠NULL do
        print p.value
        p=p.next
    end while
  
```

Time complexity: $O(|L|)$

Exercise #9: Traversing a linked list

65/87

What does this code do?

```

1  p=list
2  while p≠NULL do
3      print p.value
4      if p.next≠NULL then
5          p=p.next.next
6      else
7          p=NULL
8      end if
9  end while

```

What is the purpose of the conditional statement in line 4?

Every second list element is printed.

If p happens to be the last element in the list, then $p.next.next$ does not exist.

The if-statement ensures that we do not attempt to assign an undefined value to pointer p in line 5.

Exercise #10: Traversing a linked list

67/87

Rewrite `showLL()` as a recursive function.

```

printLL(L):
|   Input linked list L
|
|   if L≠NULL do
|       print p.value
|       printLL(L.next)
|   end if

```

Modifying a Linked List

69/87

Insert a new element at the beginning:

```

insertLL(L,d):
|   Input  linked list L, value d
|   Output L with d prepended to the list
|
|   new=makeNode(d)    // create new list element
|   new.next=L         // link to beginning of list
|   return new         // new element is new head

```

Time complexity: $O(1)$

... Modifying a Linked List

70/87

Delete the *first* element:

```

deleteHead(L):
|   Input   non-empty linked list L, value d
|   Output L with head deleted
|
|   return L.next    // move to second element

```

Time complexity: $O(1)$

Delete a *specific* element (recursive version):

```

deleteLL(L,d):
|   Input   linked list L
|   Output L with element d deleted
|
|   if L=NULL then                // element not in list
|       return L
|   else if L.value=d then         // d found at front
|       return deleteHead(L)       // delete first element
|   else                           // delete element in tail list
|       L.next=deleteLL(L.next,d)
|   end if
|   return L

```

Time complexity: $O(|L|)$

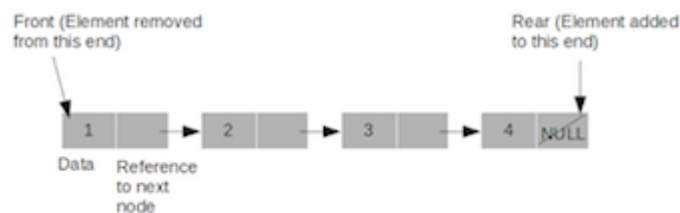
Exercise #11: Implementing a Queue as a Linked List

71/87

Develop a datastructure for a queue based on linked lists such that ...

- enqueueing an element takes constant time
- dequeuing an element takes constant time

Use pointers to both ends



Dequeue from the front ...

```

dequeue(Q):
|   Input   non-empty queue Q
|   Output front element d, dequeued from Q
|
|   d=Q.front.value           // first element in the list
|   Q.front=Q.front.next     // move to second element
|   return d

```

Enqueue at the rear ...

```

enqueue(Q,d):
    Input queue Q

    new=makeNode(d)    // create new list element
    Q.rear.next=new    // add to end of list
    Q.rear=new         // link to new end of list

```

Comparison Array vs. Linked List

73/87

Complexity of operations, n elements

	array	linked list
insert/delete at beginning	$O(n)$	$O(1)$
insert/delete at end	$\cancel{O(n)} O(1)$	$O(1)$ ("doubly-linked" list, with pointer to rear)
insert/delete at middle	$O(n)$	$O(n)$
find an element	$O(n)$ ($O(\log n)$, if array is sorted)	$O(n)$
index a specific element	$O(1)$	$O(n)$

Complexity Classes

Complexity Classes

75/87

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

76/87

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

77/87

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

78/87

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

79/87

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

80/87

Problem to solve ...

Is there a subset S of these numbers with $\sum_{x \in S} x = 1000$?

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

... Example: Subset Sum

81/87

Generate and test approach:

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

82/87

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

83/87

Algorithm:

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

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

... Example: Subset Sum

84/87

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

85/87

Cost analysis:

- $C_i = \text{\#calls to subsetsum2 () for array of length } i$
- for worst case,
 - $C_1 = 2$
 - $C_n = 2 + 2 \cdot C_{n-1} \Rightarrow C_n \approx 2^n$

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

... Example: Subset Sum

86/87

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* ...
-

Summary

87/87

- Big-Oh notation
 - Asymptotic analysis of algorithms
 - Examples of algorithms with logarithmic, linear, polynomial, exponential complexity
 - Linked lists vs. arrays

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

Produced: 6 Jan 2020