Algorithm Analysis

Deepali Londhe

1

1

- Introduction to Algorithms by T.H. Cormen, C.E. Leiserson, and R.L. Rivest, 2nd edition, McGraw-Hill, 2001.
- Fundamentals of Algorithms, by G.Brassard and P.Bratley, Prentice Hall, 1996.
- Computer Algorithms, by Horowitz, Sahni, and Rajasekaran, Computer Science Press, 1996.
- Algorithms, by Sedgewick, Addison-Wesley.
- https://runestone.academy/runestone/books/published/pythonds/AlgorithmAnalysis/WhatIsAlgorithmAnalysis.html#lst-sum1

2

2

Objective: Access a given index:

	Array	Linked List
Access [3]	1 formula	Visits 4 ListNodes
Access [4285]	1 formula	Visits 4,285 ListNodes
Access [1250000]	1 formula	Visits 1,250,000 ListNodes
Based on n pieces of data:	1 formula	Visits up to n ListNodes

3

3

Array vs LL

Task	Array	Linked List
Access Given Index	O(1)	O(n)
Insert at Front	O(n)	O(1)
Find Data location	O(n)	O(n)
Find Data in sorted array	O(log n)	O(n)
Insert After a particular element	O(n)	O(1)
Delete after a particular element	O(n)	O(1)

4

4

Analysis of Algorithms

- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
  - It must be correct
  - It must be finite (in terms of time and size)
  - It must terminate
  - It must be unambiguous
  - Which step is next?
  - It must be space and time efficient
- A program is an instance of an algorithm, written in some specific programming language
  - Program = algorithms + data structures
- Data structures
  - Methods of organizing data

5

5

Algorithms

```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

```
def foo(tom):
    fred = 0
    for bill in range(1,tom+1):
        barney = bill
        fred = fred + barney

    return fred

print(foo(10))
```

- Same
- Poor coding, not used good name of identifiers-poor readability, extra assignment statement during the accumulation step

6

6

Algorithm Efficiency

- There are often many approaches (algorithms) to solve a problem. How do we choose between them?
- At the heart of computer program design are two (sometimes conflicting) goals.
  1. To design an algorithm that is easy to understand, code, debug.
  2. To design an algorithm that makes efficient use of the computer's resources.
- Goal (1) is the concern of Software Engineering
- Goal (2) is the concern of data structures and algorithm analysis

7

7

Analysis of Algorithms

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue
- What is the goal of analysis of algorithms?
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- An important question is: How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:
  - CPU (time) usage
  - memory usage
  - disk usage
  - network usage

8

8

Characteristic operations

- In computing time complexity, one good approach is to count characteristic operations
  - What a "characteristic operation" is depends on the particular problem
  - If searching, it might be comparing two values
  - If sorting an array, it might be:
    - comparing two values
    - swapping the contents of two array locations
    - both of the above
  - Sometimes we just look at how many times the *innermost loop* is executed

9

9

Exact values

- It is sometimes possible, *in assembly language,* to compute *exact* time and space requirements
  - We know exactly how many bytes and how many cycles each machine instruction takes
  - For a problem with a known sequence of steps (factorial, Fibonacci), we can determine how many instructions of each type are required
- However, often the exact sequence of steps cannot be known in advance
  - The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)

10

10

Higher-level languages

- In a higher-level language, we *do not know* how long each operation takes
  - Which is faster, `x < 10` or `x <= 9` ?
  - We don't know exactly what the compiler does with this
  - The compiler probably optimizes the test anyway (replacing the slower version with the faster one)
- In a higher-level language we *cannot* do an exact analysis
  - Our timing analyses will use *major* oversimplifications
  - Nevertheless, we can get some very useful results

11

11

Time and space

- To analyze an algorithm means:
  - developing a formula for predicting *how fast* an algorithm is, based on the size of the input (time complexity), and/or
  - developing a formula for predicting *how much memory* an algorithm requires, based on the size of the input (space complexity)
- Usually time is our biggest concern
  - Most algorithms require a fixed amount of space

12

12

Space Complexity

- Space complexity = The amount of memory required by an algorithm to run to completion
  - [Core dumps = the most often encountered cause is "memory leaks" – the amount of memory required larger than the memory available on a given system]
- Some algorithms may be more efficient if data completely loaded into memory
  - Need to look also at system limitations
  - E.g. Classify 2GB of text in various categories [politics, tourism, sport, natural disasters, etc.] – can I afford to load the entire collection?

13

13

Space Complexity (cont'd)

1. Fixed part: The size required to store certain data/variables, that is **independent** of the size of the problem:
   - e.g. name of the data collection
   - Independent of number, size of input and outputs.
   - This part includes the instruction space (i.e. space for code), space for simple variables and fixed size component variables, space for constants & so on.
2. Variable part: Space needed by component variables, whose size is **dependent** on the size of the problem i.e. on the particular problem instances(input/output):
   - e.g. actual text
   - load 2GB of text VS. load 1MB of text
   - Space needed by reference variables and recursion stack space, this space depends on instance characteristics. Also this included the data structure components like Linked list, heap, trees, graphs etc.

Space(A) = Fixed Components(A) + Variable Components(A)

14

14

Space Complexity-Example

```
Algorithm Sum(number, size)\\ procedure will produce sum of all numbers provided in 'number' list
{
      result=0.0;
      for count = 1 to size do                \\will repeat from 1,2,3,4,....size times
            result= result + number[count];
      return result;
}
```

- Fixed components as 'result', 'count' and 'size' variable there for total space required is three(3) words.
- Variable components is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n'). because this will decide the size of 'number' list and will also drive the for loop. therefore if the space used by size is one word then the total space required by 'number' variable will be 'n'(value stored in variable 'size').
- therefore the space complexity can be written as **Space(Sum) = 3 + n;**

15

15

Space Complexity (cont'd)

- $S(P) = c + S_p$(instance characteristics)
  - c = constant
- Example 1:

```
Algorithm abc (a,b,c)
{
Return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

Space required: S_p (Instance char's) = 0

Space needed by abc is independent of instance characeristics.

16

16

Space Complexity (cont'd)

- Example 2:

```
Algorithm Sum (a, n)
{
    float s = 0.0;
    for  i := 1 to n do
            s+ = a[i];
    return s;
}
```

Space required: $S_{sum}(n) >= (n+3)$

n for a [ ], one each for i, n, s constant space!

17

17

Space Complexity (cont'd)

- Example 3:

```
Algorithm RSum(a,n)
{
 if (n<=0) then return 0.0;
 else return RSum(a,n-1)+a[n];
}
```

Space required: $S_{RSum}(n) >= 3(n+1)$

Recursion stack space includes space for formal para, local variables, return address assume 1 word each

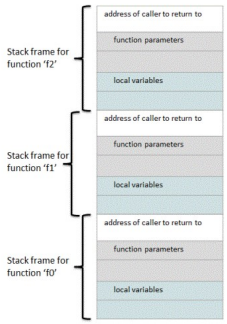
Depth of recursion = n+1

Recursion stack space needed >=3(n+1)

18

18

- To conclude, space complexity of recursive algorithm is proportinal to maximum depth of recursion tree generated. If each function call of recursive algorithm takes O(m) space and if the maximum depth of recursion tree is 'n' then space complexity of recursive algorithm would be O(nm).



Call stack in memory for call sequence f0->f1->f2

19

19

Analyze programs

- Criteria to judge the program
  - Does it do what we want it to do?
  - Does it work correctly according to original specifications of the task?
  - Is there any documentation which describes how to use it and how it works?
  - Are subroutines created in such way that they perform logical sub-functions?
  - Is the code readable?

20

20

Program Analysis

- Performance evaluation can loosely divided into 2 phases
  - Priori estimates
  - Posterior testing
- Priori estimate (Performance analysis)- This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- Posterior testing (performance estimate)-This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.
- For every statement, find
  - The amount of time a single execution will take
  - No. of times it is executed, i.e. frequency count
- The product of these no.s will be total time taken by the statement.
- Frequency count may vary from data set to data set.

21

21

Contd..

- It is impossible to determine exactly how much time it takes to execute command unless we have following information:
  - The m/c we are executing on;
  - The m/c instruction set;
  - Time required for each m/c instruction;
  - Time required by compiler to translate sourrce to m/c language.
- It is difficult to get reliable time bcoz clock limitations, multiprogramming or time sharing environment so concentrate on frequency count.

22

22

Priori Analysis & Posteriori Testing

Priori Analysis
- Algorithm
- Independent of Language
- Hardware Independent
- Time and Space Function

Posteriori Testing
- Program
- Language Dependent
- Hardware dependent
- Watch time and Bytes

23

23

Program(Algorithm) Analysis

The running time of an algorithm is influenced by several factors like:

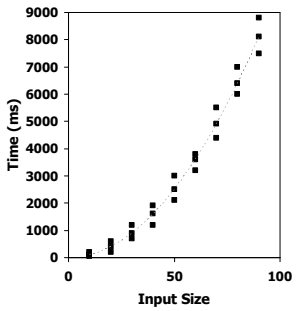
- Speed of the machine running the program
- Language in which the program was written.
- The size of the input
- Organization of the input

24

24

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes 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

Input Size

- Input size (number of elements in the input)
  - size of an array
  - polynomial degree
  - # of elements in a matrix
  - # of bits in the binary representation of the input
  - vertices and edges in a graph

Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$Lower\ Bound \le Running\ Time \le Upper\ Bound$$

- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random

29

29

How do we compare algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

Not good: times are specific to a particular computer !!

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

30

30

Ideal Solution

- Express running time as a function of the input size *n* (i.e., *f(n)*).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

31

31

A Computational Model

- To summarize algorithm runtimes, we can use a computer independent model
  - instructions are executed sequentially
  - count all assignments, comparisons, and increments there is infinite memory
  - every simple instruction takes one unit of time

32

32

Time Complexity

- Time Complexity of an algorithm(basically when converted to program) is the amount of computer time it needs to run to completion.
- The time taken by a program is the sum of the compile time and the run/execution time .
- The compile time is independent of the instance(problem specific) characteristics. following factors effect the time complexity:
  1. Characteristics of compiler used to compile the program.
  2. Computer Machine on which the program is executed and physically clocked.
  3. Multiuser execution system.
  4. Number of program steps.

33

Time Complexity..

- Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 & 4), so for any algorithm 'A' it is provided as:
  - **Time(A) = Fixed Time(A) + Instance Time(A)**
- Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like
  - comments count as zero steps,
  - an assignment statement which does not involve any calls to other algorithm is counted as one step,
  - for iterative statements we consider the steps count only for the control part of the statement etc.

34

Time Complexity...

- Therefore to calculate total number program of program steps we use following procedure. For this we build a table in which we list the total number of steps contributed by each statement. This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed. This procedure is explained using an example.

Statement	Steps per execution	Frequency	Total Steps
Algorithm Sum(number,size)	0	-	0
{	0	-	0
result=0.0;	1	1	1
for count = 1 to size do	1	size+1	size + 1
result= result + number[count];	1	size	size
return result;	1	1	1
}	0	-	0
Total			2size + 3

Time(Sum) = C + (2size +3)

35

Simple Instructions

Count the simple instructions

- assignments have cost of 1
- comparisons have a cost of 1
- let's count all parts of the loop

  for(int j = 0; j < n; j++)
- j=0 has a cost of 1, j<n executes n+1 times,and j++ executes n times for a total cost of 2n+2
- each statement in the repeated part of a loop have have a cost equal to number of iterations

36

Example

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1

	Frequency
arr[0] = 0;	1
arr[1] = 0;	1
arr[2] = 0;	1
...	..
arr[N-1] = 0;	1
----------	
	c_1 x N

37

Algorithm 2

	Frequency
for(i=0; i<N; i++)	N+1
arr[i] = 0;	N

	2N + 1

38

Another Example

- ***Algorithm 3***

	Frequency
sum = 0;	1
for(i=0; i<N; i++)	N+1
for(j=0; j<N; j++)	N(N+1)
sum += arr[i][j];	N^2

	$1 + (N+1) + (N \times (N+1)) + N^2$

39

Asymptotic Analysis

- To compare two algorithms with running times *f(n)* and *g(n),* we need a **rough measure** that characterizes **how fast each function grows.**
- *Hint:* use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**
  (i.e., for large values of *n*)

40

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*
  Cost: cost_of_elephants + cost_of_goldfish
  Cost ~ cost_of_elephants (approximation)
- The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \sim n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

41

41

Rate of Growth

- Imagine two functions:
  f(n) = 1000n g(n) = $n^2 + n$
  - When n is small, which is the bigger function?
  - When n is big, which is the bigger function?
  - We can say: *g(n) grows faster than f(n)*
    - Expressed in differential calculus, the rate of change of these two functions are related as

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

42

42

Rate of Growth, another view

Function growth and weight of terms as a percentage of all terms as n increases for

$$f(n) = n^2 + 80n + 500$$

n	f(n)	n^2	80n	500
10	1,400	100 (7%)	800 (57%)	500 (36%)
100	18,500	10,000 (54%)	8,000 (43%)	500 (3%)
1000	1,0805,000	1,000,000 (93%)	80,000 (7%)	500 (0%)
10000	100,800,500	100,000,000 (99%)	800,000 (1%)	500 (0%)

Conclusion: consider highest order term with the coefficient dropped, also drop all lower order terms

43

43

Definition

- The *asymptotic growth* of an algorithm
  - describes the relative growth of an algorithm as n gets very large
  - With speed and memory increases doubling every two years, the asymptotic efficiency *where n is very large* is the thing to consider
  - There are many sorting algorithm that are "on the order of" n^2 (there are roughly n×n instructions executed)
  - Other algorithms are "on the order of" $n \times \log_2 n$
    - and this is a huge difference when n is very large

44

44

Big O

- Linear search is "on the order of n", which can be written as O(n) to describe the upper bound on the number of operations
- This is called *big O* notation
- Other orders of magnitude—g(n) —you will see:
  - **O(1)** *a.k.a constant (the size of n has no effect)*
  - **O(log n)** *a.k.a logarithmic*
  - **O(n)** *linear time*
  - **O(n log n)** *log linear time*
  - **O(n^2)** *quadratic*
  - **O(n^3)** *cubic*
  - **O(2^n)** *exponential*

45

45

Big-O Definition

- Definition of big-O:
  - f(n) is O(g(n)) if and only if there exist two positive constants c and N such that f(n) <= c*g(n) for all n > N
- Example: $f(n) = n^2 + 2n + 3$ is $O(n^2)$ because when c = 1.5, $n^2 + 2n + 3 \leq 1.5N^2$ when $N \geq 5.1625$: $39.976 \leq 39.977$
- When N = 6, $51 \leq 54$ and when N = 8, $83 \leq 96$

46

46

Big O notation

- When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:
  - Throwing out all but the highest-order term
  - Throwing out all the constants
- If an algorithm takes $12n^3+4n^2+8n+35$ time, we simplify this formula to just n^3
- We say the algorithm requires $O(n^3)$ time
  - We call this Big O notation
  - (More accurately, it's Big Ω, but we'll talk about that later)

47

47

Can we justify Big O notation?

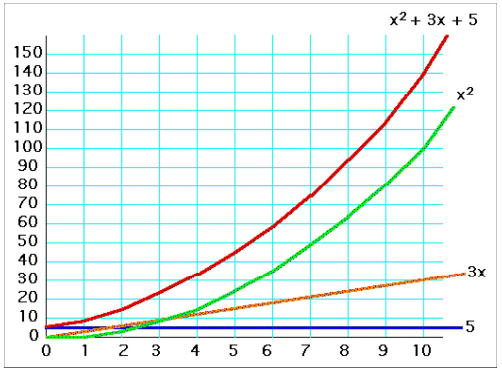
- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - **For sufficiently large problem sizes, the highest-order term swamps all the rest!**
- Consider $R = x^2 + 3x + 5$ as x varies:

x = 0	$x^2 = 0$	3x = 0	5 = 5	R = 5
x = 10	$x^2 = 100$	3x = 30	5 = 5	R = 135
x = 100	$x^2 = 10000$	3x = 300	5 = 5	R = 10,305
x = 1000	$x^2 = 1000000$	3x = 3000	5 = 5	R = 1,003,005
x = 10,000	$x^2 = 10^8$	$3x = 3*10^4$	5 = 5	R = 100,030,005
x = 100,000	$x^2 = 10^{10}$	$3x = 3*10^5$	5 = 5	R = 10,000,300,005

48

48

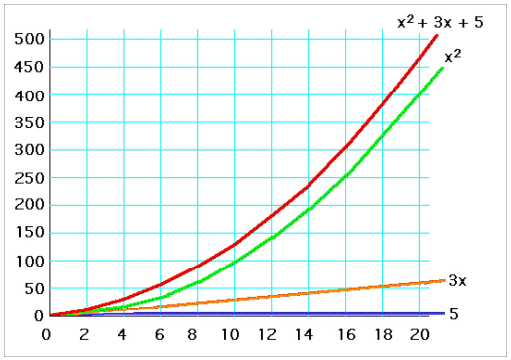
$y = x^2 + 3x + 5$, for x=1..10



49

49

$y = x^2 + 3x + 5$, for x=1..20



50

50

Common time complexities

BETTER

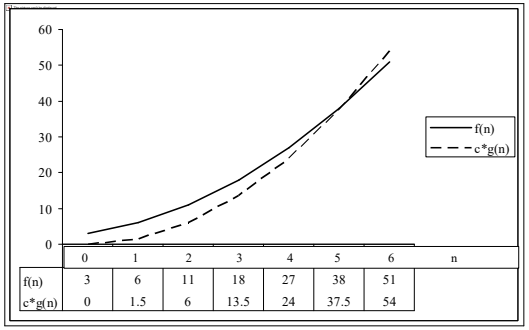
WORSE

- O(1) constant time
- O(log n) log time
- O(n) linear time
- O(n log n) log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

51

51

c*g(n) bigger than f(n) at 5.1625



52

52

n	f(n)	c*g(n)
1	6	1.5
2	11	6
3	18	13.5
4	27	24
5.1625	39.97641	39.97711
5	38	37.5
6	51	54
7	66	73.5
8	83	96
9	102	121.5
10	123	150
11	146	181.5
12	171	216

Another View

53

53

The Big 'O' Notation

To show that f(n) = 4n + 5 = O(n), we need to produce a constant C such that:

$f(n) \leq C * n$ for all n.

If we try C = 4, this doesn't work because 4n + 5 is not less than 4n. We need C to be at least 9 to cover *all* n. If n = 1, C has to be 9, but C can be smaller for greater values of n (if n = 100, C can be 5). Since the chosen C must work for all n, we must use 9:

$4n + 5 \leq 4n + 5n = 9n$

Since we have produced a constant C that works for all n, we can conclude: T(4n + 5) = O(n).

54

54

The Big 'O' Notation

Find the values of 'C' for the big 'O' notations for the following functions:

1) $f(n) = n^2 + 5n - 1$

2) $f(n) = 2n^7 - 6n^5 + 10n^2 - 5$

55

55

The Big 'O' Notation

1: Suppose $f(n) = n^2 + 5n - 1$.

$f(n) = n^2 + 5n - 1 < n^2 + 5n$

$\leq n^2 + 5n^2$ (since $n <= n^2$ for all integers n)

$= 6n^2$ **n>=1**

Therefore, if C = 6, we have shown that **$f(n) = O(n^2)$**.

2: Suppose $f(n) = 2n^7 - 6n^5 + 10n^2 - 5$.

$f(n) < 2n^7 + 10n^2$

$\leq 2n^7 + 10n^7$

$= 12n^7$

Thus, with C = 12 and we have shown that **$f(n) = O(n^7)$**

56

56

Asymptotic Notations

- O notation: asymptotic"less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)

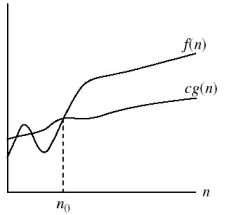
57

57

Asymptotic notations (cont.)

- Ω - *notation*

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



Ω(g(n)) is the set of functions with larger or same order of growth as g(n)

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

58

58

Examples

- $5n^2 = \Omega(n)$

  $\exists\ c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow cn \le 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

- $100n + 5 \ne \Omega(n^2)$

  $\exists\ c, n_0$ such that: $0 \le cn^2 \le 100n + 5$

  $100n + 5 \le 100n + 5n\ (\forall\ n \ge 1) = 105n$

  $cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

  Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

  $\Rightarrow$ contradiction: n cannot be smaller than a constant

- $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

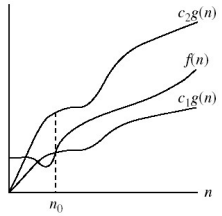
59

59

Asymptotic notations (cont.)

- Θ-*notation*

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



Θ(g(n)) is the set of functions with the same order of growth as g(n)

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

60

60

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \ \forall n \geq 0 \Rightarrow c_2 = \frac{1}{2}$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \ (\forall n \geq 2) = \frac{1}{4} n^2$

    $\Rightarrow c_1 = \frac{1}{4}$

- $n \neq \Theta(n^2)$: $c_1 n^2 \leq n \leq c_2 n^2$

  $\Rightarrow$ only holds for: $n \leq 1/c_1$

61

61

Examples

- $6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

  $\Rightarrow$ only holds for: $n \leq c_2 / 6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

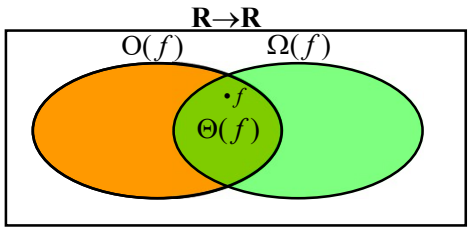
  $\Rightarrow c_2 \geq n/\log n, \ \forall n \geq n_0$ – impossible

62

62

Relations Between Different Sets

- Subset relations between order-of-growth sets.



63

63

Common time complexities

BETTER

- O(1) constant time
- O(log n) log time
- O(n) linear time
- O(n log n) log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

64

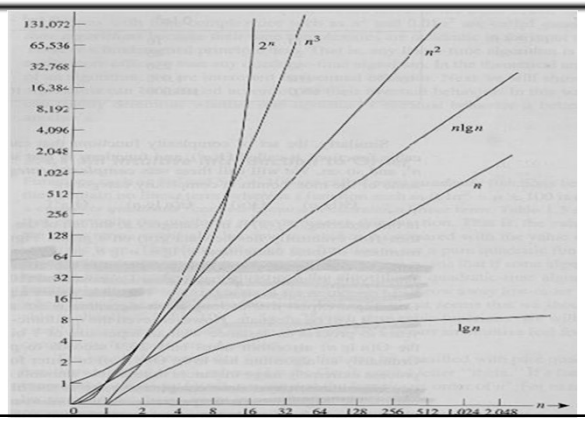
64

Time complexity

The functions shown in the table below are mainly used to express the running time in terms of the input size (n).

Input Size	(1)	log (n)	n*log(n)	n	n^2	n^3	2^n
5	1	3	15	5	25	125	2^5
1000	1	10	10^4	1000	10^6	10^9	2^{1000}
10000	1	13	10^5	10000	10^8	10^{12}	2^{10000}

Common orders of magnitude



Common orders of magnitude

Table 1.4 Execution times for algorithms with the given time complexities

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{11} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	0 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.
†1 ms = 10^{-3} second.

Logarithms and properties

- In algorithm analysis we often use the notation "log n" without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$\lg^k n = (\lg n)^k$

$\lg \lg n = \lg(\lg n)$

$\log x^y = y \log x$

$\log xy = \log x + \log y$

$\log \frac{x}{y} = \log x - \log y$

$\log_b x = \frac{\log_a x}{\log_a b}$

$a^{\log_b x} = x^{\log_b a}$

Searching

- Locating the element in the given list.
- List can be represented using
  - Array
  - Linked list
  - Tree
  - Heap
  - File
- Search key in sequentially by comparing to every record. If found then search successful otherwise unsuccessful.
- This is called sequential search.

69

69

Sequential Search: Algorithm

- Seq_search(a,n,k)
  - Search for key k in array a of n elements

```
i=0;
While (a[i]≠k && i<n)
   i=i+1;
If (i<n)
   print(Number found)
```

70

70

Sequential Search: Analysis

- Seq_search(a,n,k)
  - Search for key k in array a of n elements

	cost
i=0;	1
While (a[i]≠k && i<n)	n+1, n+1
i=i+1;	n
If (i<n)	1
print(Number found)	1

- Total cost
  - 1+2n+2+n+2=3n+5
- Time complexity: O(n)

71

71

Different Cases

- The total cost of sequential search is 3n + 5
  - But is it always exactly 3n + 5 instructions?
  - The last assignment does not always execute
    - But does one assignment really matter?
  - How many times will the loop actually execute?
    - that depends
  - If searchID is found at index 0: _____ iterations
    - best case
  - If searchID is found at index n-1:_____ iterations
    - worst case
  - If searchID is found at index n/2:_____ iterations
    - Average case

72

72

- Sequential search
  - Efficiency
    - Worst case: O(n)
    - Average case: O(n/2) ~ O(n)
    - Best case: O(1)

73

73

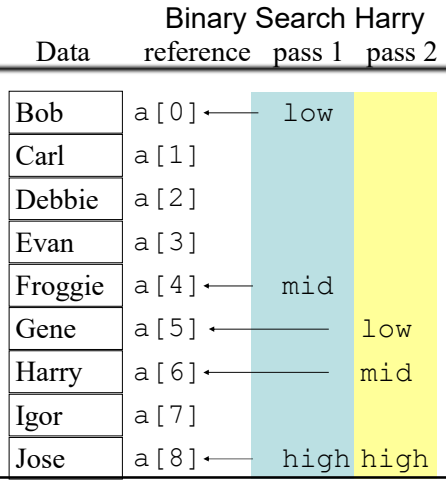
Binary search

- We'll see that binary search can be a more efficient algorithm for searching
- It works only on sorted arrays like this
  - Compare the element in the middle
  - if that's the target, quit and report success
  - if the key is smaller, search the array to the left
  - otherwise search the array to the right
- This process repeats until the target is found or there is nothing left to search
- Each comparison narrows search by half

74

74

Binary Search Harry



75

75

Binary Search algorithm

```
int binary_search(int key, int arr[], int size)
{
    int low = 0, high = size, mid;
    while(low<=high)
    {
        mid = (low + high) / 2;
        if(arr[mid] < Key)
            low = mid + 1;
        else
            if(arr[mid] > val)
                high = mid - 1;
            else
                return mid;
    }
    return -1;
}
```

76

76

How fast is Binary Search?

- Best case: 1
- Worst case: when target is not in the array
- At each pass, the "live" portion of the array is narrowed to half the previous size.
- The series proceeds like this:
  - n , n/2, n/4, n/8, ...
- Each term in the series represents one comparison. How long does it take to get to 1?
  - This will be the number of comparisons

77

77

Binary Search (con.)

1, 2, 4, 8, 16, ... , $k >= n$ or
20, 21, 22, 23, 24, ... , $2c >= n$

- Could start at 1 and double until we get to n

1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
:	:
K>=n	$2^c >= n$

- The length of this series is c+1
- The question is
  - 2 to what power *c* is greater than or equal to n?
    - if n is 8, *c* is 3
    - if n is 1024, *c* is 10
    - if n is 16,777,216, *c* is 24
- Binary search is O(log n) *base 2 assumed*

78

78

Comparing O(n) to O(log n)

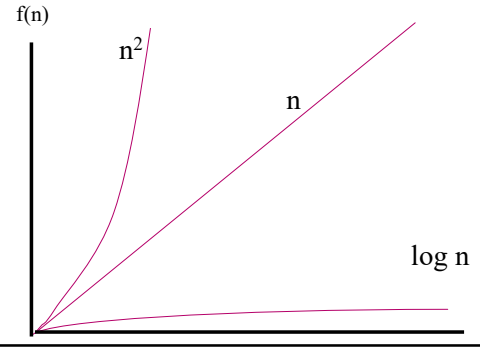
Rates of growth and logarithmic functions

Power of 2	n	$\log_2 n$
2^4	16	4
2^8	128	8
2^{12}	4,096	12
2^{24}	16,777,216	24

79

79

Graph Illustrating Relative Growth n, log n, n^2



80

80

Other logarithm examples

- The guessing game:
  - Guess a number from 1 to 100
    - try the middle, you could be right
    - if it is too high
      - check near middle of 1..49
    - if it is too low
      - check near middle of 51..100
  - Should find the answer in a maximum of 7 tries
    - If 1..250, a maximum of $2^c >= 250$, c == 8
    - If 1..500, a maximum of $2^c >= 500$, c == 9
    - If 1..1000, a maximum of $2^c >= 1000$, c == 10

81

81

Sorting Algorithms

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
- Categories of sorting algorithms
  - An internal sort
    - Requires that the collection of data fit entirely in the computer's main memory
  - An external sort
    - The collection of data will not fit in the computer's main memory all at once but must reside in secondary storage

82

82

Sorting Algorithms and Their Efficiency

- Data items to be sorted can be
  - Integers
  - Character strings
  - Objects
- Sort key
  - The part of a record that determines the sorted order of the entire record within a collection of records

83

83

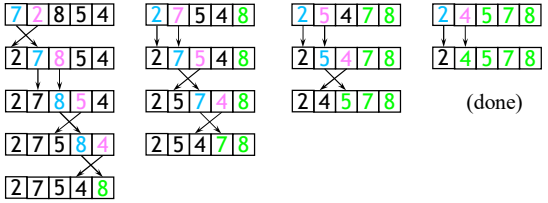
Bubble sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

84

84

Bubble sort(Example)



85

85

Bubble Sort

```
void bubbleSort (int a[ ] , int size)
{
   int i, j, temp;
   for ( i = 0; i < size; i++ )    /* controls passes through the list */
   {
         for ( j = 0; j < size - 1; j++ )   /* performs adjacent comparisons */
         {
            if ( a[ j ] > a[ j+1 ] )   /* determines if a swap should occur */
            {
                  temp = a[ j ];      /* swap is performed */
                  a[ j ] = a[ j + 1 ];
                  a[ j+1 ] = temp;
            }
         }
   }
}
```

86

86

a[]={5,4,3,2,1}
pass 0 4 3 2 1 5
pass 1 3 2 1 4 5
pass 2 2 1 3 4 5
pass 3 1 2 3 4 5
sorted array 1 2 3 4 5

a[]={1,4,3,2,5}
pass 0 1 3 2 4 5
pass 1 1 2 3 4 5
pass 2 1 2 3 4 5
pass 3 1 2 3 4 5
sorted array 1 2 3 4 5

87

87

```
void bubbleSort(int numbers[], int n)
 { int i, j, temp,exch;
   for (i = (n - 1); i >= 0; i- -)
  {
     for (j = 1; j <= i; j++)
    {      exch=0;
           if (numbers[j-1] > numbers[j])
         {
            temp = numbers[j-1];
            numbers[j-1] = numbers[j];
            numbers[j] = temp;
            exch++;
          }
      }
    if (exch==0)
           return;
  }
}
```

88

88

```
void bubbleSort (int a[ ] , int size)
{
   int i, j, temp,last, exch;

last=n;
   for ( i = 0; i < size; i++ )    /* controls passes through the list */
   {
    exch=0;
    for ( j = 0; j < last - 1; j++ )   /* performs adjacent comparisons */
    {
       if ( a[ j ] > a[ j+1 ] )   /* determines if a swap should occur */
      {
          temp = a[ j ];       /* swap is performed */
          a[ j ] = a[ j + 1 ];
          a[ j+1 ] = temp;
           exch++;              }
    }
     if (exch==0)
          return;
    else
              last=last-1;
    }     }
```

89

89

Bubble Sort Analysis

- The outer loop is executed n-1 times (call it n, that's close enough)
- Each time the outer loop is executed, the inner loop is executed
  - Inner loop executes n-1 times at first, linearly dropping to just once
  - On average, inner loop executes about n/2 times for each execution of the outer loop
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is n * n/2 * k, that is, O(n2/2 + k) = O(n2)

90

90

Selection Sort

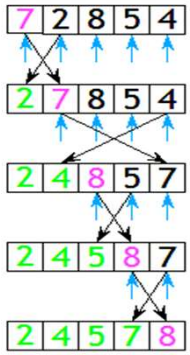
Given an array of length n,

- –Search elements 0 through n-1 and select the smallest
  - •Swap it with the element in location 0
- –Search elements 1 through n-1 and select the smallest
  - •Swap it with the element in location 1
- –Search elements 2 through n-1 and select the smallest
  - •Swap it with the element in location 2
- –Search elements 3 through n-1 and select the smallest
  - •Swap it with the element in location 3
- –Continue in this fashion until there's nothing left to search

91

91

Selection Sort Example



92

92

Selection Sort

```
Algorithm selectionSort(int a[], int n)
Declare int i, j, min, temp
for (i = 0; i < n-1; i++)                          n+1
{       min = i;                                   n
        for (j = i+1; j < n; j++)                  n (n+1)
        {
                if (numbers[j] < numbers[min])     n2
                        min = j;                   n2
         }
        Swap (a[i],a[min])    }                    n
                                                   3n2+4n+1
```


Selection Sort

- Selection sort
  - Strategy
    - Select the largest item and put it in its correct place
    - Select the next largest item and put it in its correct place, etc.

Shaded elements are selected; boldface elements are in order.

A selection sort of an array of five integers

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

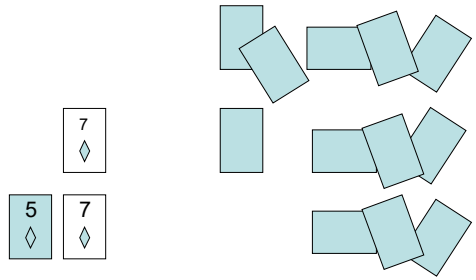
Selection Sort

- Analysis:
  - The outer loop executes n-1 times
  - The inner loop executes about n/2 times on average (from n to 2 times)
  - Work done in the inner loop is constant (swap two array elements)
  - Time required is roughly (n-1)*(n/2)
  - You should recognize this as $O(n^2)$
- Analysis
  - Selection sort is $O(n^2)$
- Advantage of selection sort
  - It does not depend on the initial arrangement of the data
- Disadvantage of selection sort
  - It is only appropriate for small n

Insertion Sort

- Finding the element's proper place
- Making room for the inserted element (by shifting over other elements)
- Inserting the element
- Insertion sort works the same way as arranging your hand when playing cards.
- Out of the pile of unsorted cards that were dealt to you, you pick up a card and place it in your hand in the correct position relative to the cards you're already holding.

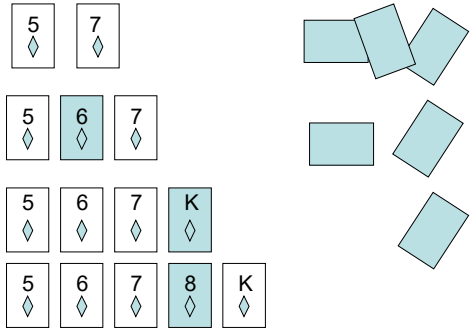
Arranging Your Hand



97

97

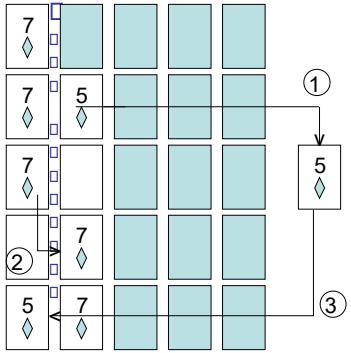
Arranging Your Hand



98

98

Insertion Sort



Unsorted - shaded

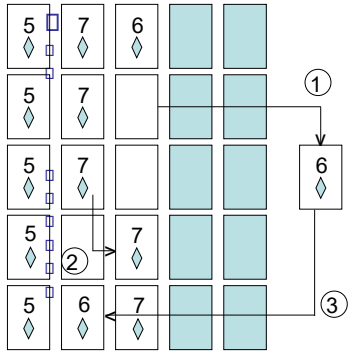
Look at 2nd item - 5.
Compare 5 to 7.
5 is smaller, so move 5 to temp, leaving an empty slot in position 2.
Move 7 into the empty slot, leaving position 1 open.

Move 5 into the open position.

99

99

Insertion Sort (con't)



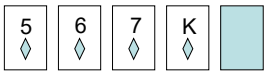
Look at next item - 6.
Compare to 1st - 5.
6 is larger, so leave 5.
Compare to next - 7.
6 is smaller, so move 6 to temp, leaving an empty slot.
Move 7 into the empty slot, leaving position 2 open.

Move 6 to the open 2nd position.

100

100

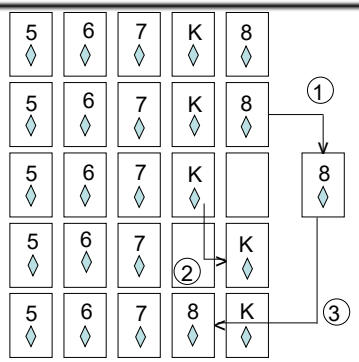
Insertion Sort (con't)



- Look at next item - King.
- Compare to 1st - 5.
- King is larger, so leave 5 where it is.
- Compare to next - 6.
- King is larger, so leave 6 where it is.
- Compare to next - 7.
- King is larger, so leave 7 where it is.

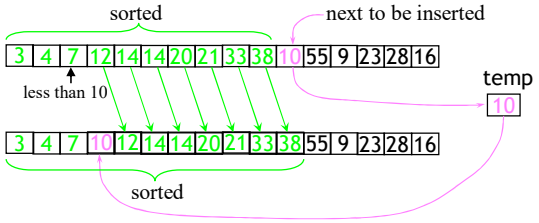
101

Insertion Sort (con't)



102

Insertion Sort



103

Insertion Sort

```
Algorithm insertionSort(int numbers[], int array_size)
Declare int i, j, index;
for (i=1; i < array_size; i++)                  n+1
{ index = numbers[i];                           n
j = i;                                          n
while ((j > 0) && (numbers[j-1] > index))       n(j+1)
{ numbers[j] = numbers[j-1];                    n*j
 j = j - 1; }                                   n*j
numbers[j] = index; } }                         n
                                                3n²+5n+2
```


104

Analysis of insertion sort

- We run once through the outer loop, inserting each of n elements; this is a factor of n
- On average, there are n/2 elements already sorted
  - The inner loop looks at (and moves) half of these
  - This gives a second factor of n/4
- Hence, the time required for an insertion sort of an array of n elements is proportional to $n^2/4$
- Discarding constants, we find that insertion sort is $O(n^2)$

105

105

Summary

- Bubble sort, selection sort, and insertion sort are all $O(n^2)$
- As we will see later, we can do much better than this with somewhat more complicated sorting algorithms
- Within $O(n^2)$,
  - Bubble sort is very slow, and should probably never be used for anything
  - Selection sort is intermediate in speed
  - Insertion sort is usually the fastest of the three--in fact, for small arrays (say, 10 or 15 elements), insertion sort is faster than more complicated sorting algorithms
- Selection sort and insertion sort are "good enough" for small arrays

106

106

Thank You

107

107