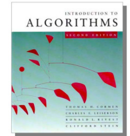
Design and Analysis of Computer Algorithm Lecture: Introduction

Dr. G. L. Prajapati, IET DAVV, Indore INDIA

More Information

- Textbook
  - ***Introduction to Algorithms 2nd*** *, Cormen, Leiserson, Rivest and Stein*, The MIT Press, 2001.



- Others
  - ***Introduction to Design & Analysis Computer Algorithm 3rd,*** Sara Baase, Allen Van Gelder, Adison-Wesley, 2000.
  - ***Algorithms***, Richard Johnsonbaugh, Marcus Schaefer, Prentice Hall, 2004.
  - ***Introduction to The Design and Analysis of Algorithms 2nd Edition,*** Anany Levitin, Adison-Wesley, 2007.

What is Algorithm?

- Algorithm
  - is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
  - is thus a sequence of computational steps that transform the input into the output.
  - is a tool for solving a well - specified computational problem.
  - Any special method of solving a certain kind of problem (Webster Dictionary)

What is a program?

- A program is the expression of an algorithm in a programming language
- a set of instructions which the computer will follow to solve a problem

Where We're Going (1/2)

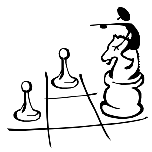
- Learn general approaches to algorithm design
  - Divide and conquer
  - Greedy method
  - Dynamic Programming
  - Basic Search and Traversal Technique
  - Graph Theory
  - Linear Programming
  - Approximation Algorithm
  - NP Problem

Where We're Going(2/2)

- Examine methods of analyzing algorithm correctness and efficiency
  - Recursion equations
  - Lower bound techniques
  - O,Omega and Theta notations for best/worst/average case analysis
- Decide whether some problems have no solution in reasonable time
  - List all permutations of n objects (takes n! steps)
  - Travelling salesman problem
- Investigate memory usage as a different measure of efficiency

Some Application

- Study problems these techniques can be applied to
  - sorting
  - data retrieval
  - network routing
  - Games
  - etc



The study of Algorithm

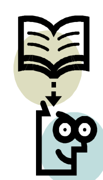
- How to devise algorithms
- How to express algorithms
- How to validate algorithms
- How to analyze algorithms
- How to test a program

Importance of Analyze Algorithm

- Need to recognize limitations of various algorithms for solving a problem
- Need to understand relationship between problem size and running time
  - When is a running program not good enough?
- Need to learn how to analyze an algorithm's running time without coding it
- Need to learn techniques for writing more efficient code
- Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize

Why do we analyze about them?

- understand their behavior, and (Job -- Selection, performance, modify)
- improve them. (Research)



What do we analyze about them?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done (aka complexity)
  - Basic operations to do task
- Amount of space used
  - Memory used

What do we analyze about them?

- Simplicity, clarity
  - Verification and implementation.
- Optimality
  - Is it impossible to do better?

Complexity

❖The complexity of an algorithm is simply the amount of work the algorithm performs to complete its task.

RAM model

❖has one processor
❖executes one instruction at a time
❖each instruction takes "unit time“
❖has fixed-size operands, and
❖has fixed size storage (RAM and disk).

What's more important than performance?

❖Modularity
❖Correctness
❖Maintainability
❖Functionality
❖Robustness
❖User-friendliness
❖Programmer time
❖Simplicity
❖Extensibility
❖Reliability

Why study algorithms and performance?

❖Algorithms help us to understand ***scalability.***
❖Performance often draws the line between what is feasible and what is impossible.
❖Algorithmic mathematics provides a ***language*** *for talking about program behavior.*
❖Performance is the ***currency*** *of computing.*
❖The lessons of program performance generalize to other computing resources.
❖Speed is fun!

Example Of Algorithm

What is the running time of this algorithm?

```
PUZZLE(x)
while x != 1
       if x is even
        then  x = x / 2
        else x = 3x + 1
```

Sample run: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

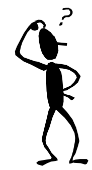
The Selection Problem (1/2)

- Problem: given a group of n numbers, determine the kth largest
- Algorithm 1
  - Store numbers in an array
  - Sort the array in descending order
  - Return the number in position k

The Selection Problem(2/2)

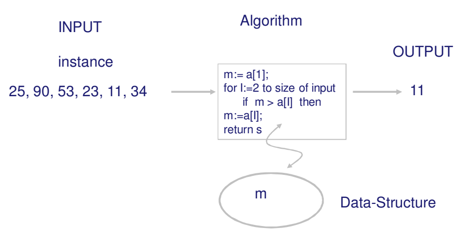
- Algorithm 2
  - Store first k numbers in an array
  - Sort the array in descending order
  - For each remaining number, if the number is larger than the kth number, insert the number in the correct position of the array
  - Return the number in position k

Which algorithm is better?



Example: What is an Algorithm?

Problem: Input is a sequence of integers stored in an array. Output the minimum.



Define Problem

- **Problem**:
  - Description of Input-Output relationship
- **Algorithm**:
  - A sequence of computational step that transform the input into the output.
- **Data Structure:**
  - An organized method of storing and retrieving data.
- **Our task:**
  - Given a problem, design a ***correct*** and ***good*** algorithm that solves it.

Example Algorithm A

Problem: The input is a sequence of integers stored in array. Output the minimum.

Algorithm A

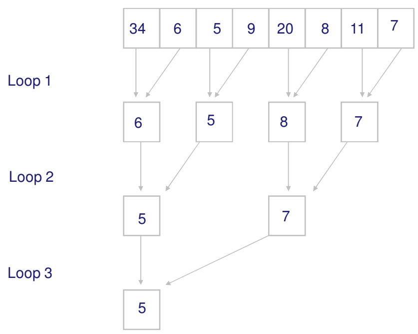
$m \leftarrow a[1]$;
For $i \leftarrow 2$ to size of input;
if $m > a[i]$ then $m \leftarrow a[i]$;
output m.

Example Algorithm B

This algorithm uses two temporary arrays.

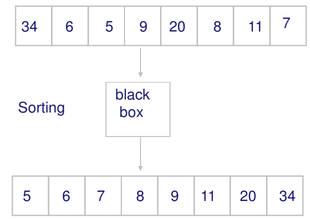
1. copy the input *a* to array *t1;*
   assign *n* ← size of input;
2. While *n* > 1
   For *i* ← 1 to *n* /2
   t2[*i*] ← min (*t1* [2**i*], *t1*[2**i* + 1]);
   copy array *t2* to *t1*;
   n ←*n/2;*
3. Output *t2*[1];

Visualize Algorithm B



Example Algorithm C

Sort the input in increasing order. Return the first element of the sorted data.



Example Algorithm D

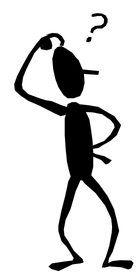
For each element, test whether it is the minimum.

1. $i \leftarrow 0$;
   flag $\leftarrow$ **true**;
2. While *flag*
   - $i \leftarrow i + 1$;
   - *min* $\leftarrow a[i]$;
   - *flag* $\leftarrow$ **false**;
   - for $j \leftarrow 1$ to size of input
     - if *min* $> a[j]$ then *flag* $\leftarrow$ true;
3. Output *min*.

Which algorithm is better?

The algorithms are correct, but which is the best?

- Measure the running time (number of operations needed).
- Measure the amount of memory used.
- Note that the running time of the algorithms increase as the size of the input increases.



What do we need?

Correctness: Whether the algorithm computes the correct solution for **all** instances

Efficiency: Resources needed by the algorithm

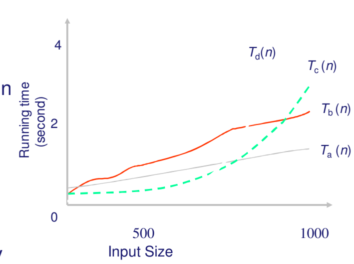
1. Time: Number of steps.
2. Space: amount of memory used.

Measurement "model": Worst case, Average case and Best case.

Time vs. Size of Input

Measurement parameterized by the size of the input.

The algorihtms A,B,C are *implemented* and run in a PC.
Algorithms D is implemented and run in a supercomputer.



Let $T_k(n)$ be the amount of time taken by the Algorithm

Methods of Proof

- Proof by Contradiction
  - Assume a theorem is false; show that this assumption implies a property known to be true is false -- therefore original hypothesis must be true
- Proof by Counterexample
  - Use a concrete example to show an inequality cannot hold
- Mathematical Induction
  - Prove a trivial base case, assume true for k, then show hypothesis is true for k+1
  - Used to prove recursive algorithms

Review: Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0
  - S(n) □ S(n+1) for all n >= k
- Then S(n) is true for all n >= k

Proof By Induction

- Claim:S(n) is true for all n >= k
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

Induction Example: Gaussian Closed Form

- Prove 1 + 2 + 3 + … + n = n(n+1) / 2
  - Basis:
    - If n = 0, then 0 = 0(0+1) / 2
  - Inductive hypothesis:
    - Assume 1 + 2 + 3 + … + n = n(n+1) / 2
  - Step (show true for n+1):

    1 + 2 + … + n + n+1 = (1 + 2 + …+ n) + (n+1)

    = n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2

    = (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1):

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

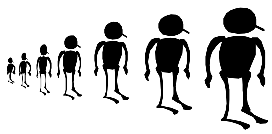
    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Induction

- We've been using *weak induction*
- *Strong induction* also holds
  - Basis: show S(0)
  - Hypothesis: assume S(k) holds for arbitrary k <= n
  - Step: Show S(n+1) follows
- Another variation:
  - Basis: show S(0), S(1)
  - Hypothesis: assume S(n) and S(n+1) are true
  - Step: show S(n+2) follows

Basic Recursion

- ❖Base case: value for which function can be evaluated without recursion
- ❖Two fundamental rules
  - Must always have a base case
  - Each recursive call must be to a case that eventually leads toward a base case



Bad Example of Recursion

Example of non-terminating recursive program (let n=1)

```
int bad(unsigned int n)
{
        if(n == 0)
            return 0;
        else
            return(bad(n/3 + 1) + n - 1);
}
```


Recursion(1/2)

Problem: write an algorithm that will strip digits from an integer and print them out one by one

```
void print_out(int n)
{
    if(n < 10)
        print_digit(n); /*outputs single-digit to terminal*/
    else {
        print_out(n/10); /*print the quotient*/
        print_digit(n%10); /*print the remainder*/
    }
}
```


Recursion(2/2)

Prove by induction that the recursive printing program works:

- basis: If n has one digit, then program is correct
- hypothesis: Print_out works for all numbers of k or fewer digits
- case k+1: k+1 digits can be written as the first k digits followed by the least significant digit

The number expressed by the first k digits is exactly floor(n/10)? which by hypothesis prints correctly; the last digit is n%10; so the (k+1)-digit is printed correctly

By induction, all numbers are correctly printed

Recursion

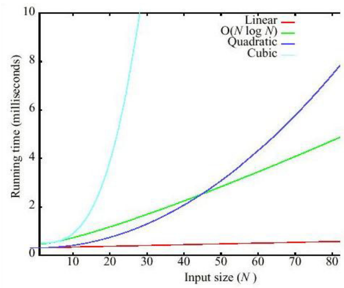
- ❖Don't need to know how recursion is being managed
- ❖Recursion is expensive in terms of space requirement; avoid recursion if simple loop will do
- ❖Last two rules
  - Assume all recursive calls work
  - Do not duplicate work by solving identical problem in separated recursive calls
- ❖Evaluate fib(4) -- use a recursion tree

  fib(n) = fib(n-1) + fib(n-2)

What is Algorithm Analysis?

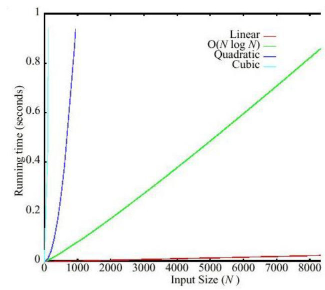
- ❖How to estimate the time required for an algorithm
- ❖Techniques that drastically reduce the running time of an algorithm
- ❖A mathematical framwork that more rigorously describes the running time of an algorithm

Running time for small inputs



Running times for small inputs

Running time for moderate inputs



Running time for moderate inputs

Important Question

- Is it always important to be on the most preferred curve?
- How much better is one curve than another?
- How do we decide which curve a particular algorithm lies on?
- How do we design algorithms that avoid being on the bad curves?

Algorithm Analysis(1/5)

- Measures the efficiency of an algorithm or its implementation as a program as the input size becomes very large
- We evaluate a new algorithm by comparing its performance with that of previous approaches
  - Comparisons are asymtotic analyses of classes of algorithms
- We usually analyze the time required for an algorithm and the space required for a datastructure

Algorithm Analysis (2/5)

- Many criteria affect the running time of an algorithm, including
  - speed of CPU, bus and peripheral hardware
  - design think time, programming time and debugging time
  - language used and coding efficiency of the programmer
  - quality of input (good, bad or average)

Algorithm Analysis (3/5)

- Programs derived from two algorithms for solving the same problem should both be
  - Machine independent
  - Language independent
  - Environment independent (load on the system,...)
  - Amenable to mathematical study
  - Realistic

Algorithm Analysis (4/5)

- In lieu of some standard benchmark conditions under which two programs can be run, we estimate the algorithm's performance based on the number of key and basic operations it requires to process an input of a given size
- For a given input size n we express the time T to run the algorithm as a function T(n)
- Concept of growth rate allows us to compare running time of two algorithms without writing two programs and running them on the same computer

Algorithm Analysis (5/5)

- Formally, let T(A,L,M) be total run time for algorithm A if it were implemented with language L on machine M. Then the complexity class of algorithm A is
  O(T(A,L1,M1) U O(T(A,L2,M2)) U O(T(A,L3,M3)) U ...
- Call the complexity class V; then the complexity of A is said to be f if V = O(f)
- The class of algorithms to which A belongs is said to be of at most linear/quadratic/ etc. growth in best case if the function T_A best(n) is such (the same also for average and worst case).

Asymptotic Performance

- In this course, we care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does O(n) running time mean? O(n^2)? O(n lg n)?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to define this notation more formally and completely

Analysis of Algorithms

- Analysis is performed with respect to a computational model
- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

Input Size

- Time and space complexity
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

Running Time

- ❖Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- ❖We can be more exact if need be

Analysis

- ❖Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- ❖Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate