CS 575

Design and Analysis of Algorithms

KD Kang

1. Introduction

- General Information
  - Course objectives
  - Topics
  - Text
  - Grading

- Syllabus on Blackboard

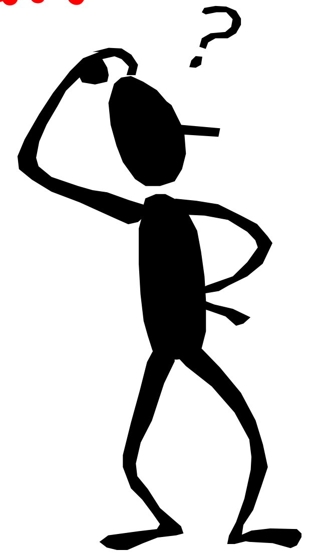
Course Objectives

- Problem Formulation
- Learn key algorithms
- Implement algorithms efficiently and correctly
- Design algorithms using well established methods
- Analyze time and space complexity
- Analyze correctness
- Understand theory of NP completeness

➔ Develop critical thinking skills for problem solving

➔ Prepare for future technical challenges

Are algorithms useful?



- Hardware
- Software
- Economics
- Biomedicine
- Computational geometry (graphics)
- Decision making
- Scheduling …..

"Great algorithms

are the beauty of

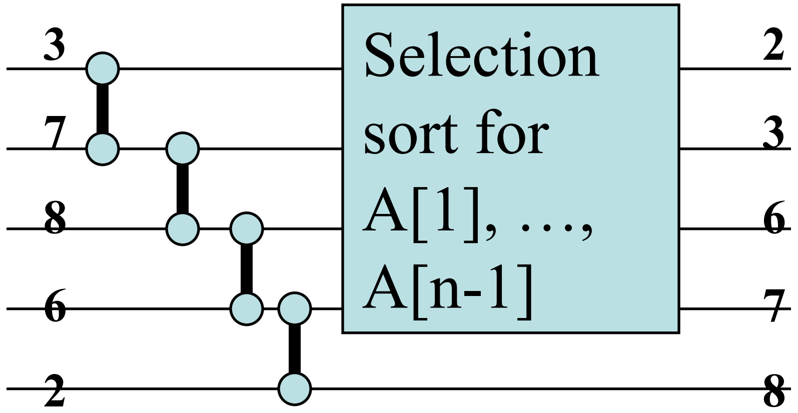
computation"

Software

- Text processing
  - String matching, spell checking, and pretty print,…
- Networks
  - Shortest paths, routing algorithms, minimum spanning trees,…
- Databases
- Compilers

Hardware Design

- VLSI design
- Multiplication
- Search
- Sort networks



Economics

- Transportation problems
  - Shortest paths, traveling salesman, …
- Optimization problems
  - Knapsack, bin packing, …
- Scheduling problems
- Location problems (e.g.,Voronoi Diagram)
- Manufacturing decisions

Technical interview (example)

- Please note this will be a technical interview. It may include questions from one or more of the following areas: coding, algorithms and design, and problem solving. Google takes an academic approach to the interviewing process. This means that we are interested in your thought process and your approach to problem solving as well as your technical abilities.. … It may also be worth refreshing on hash tables, heaps, binary trees, linked lists, depth-first search, recursion…

Math preparation

- ❑ Induction
- ❑ Logarithm
- ❑ Sets
- ❑ Permutation and combination
- ❑ Limits
- ❑ Series
- ❑ Asymptotic growth functions and recurrence
- ❑ Probability theory

- ❑ Math review is uploaded to Blackboard

Chapter 1

- ❑Definition of algorithms
- ❑Sample problems and algorithms
- ❑ Basic math review for yourself
- ❑Analysis
    - ❑Time complexity
    - ❑Notion of Order: big O, small o, Ω, Θ

Basic Concepts

- <u>Example problems</u>
  - Search: Determine whether a number x is in the list S of n integers
  - Sorting: Sort a list S of n numbers in non-decreasing order
  - Matrix multiplication
- <u>Algorithm:</u> A step-by-step procedure for solving a problem.

Importance of Algorithm Efficiency

- Time
- Storage
- Example

  - Sequential search vs. binary search
    Basic operation: comparison
    Number of comparisons grows at different rates

  - n^{th} Fibonacci sequence
    Recursive versus iterative solutions

Example: search strategy

- Sequential search vs. binary search

- Problem: determine whether x is in the sorted array S of n integer keys
- Input: key x, positive integer n, sorted (non-decreasing order) array of keys S indexed from 1 to n
- Output: location of x in S (0 if x is not in S)

Example: search strategy

- ❑Sequential search:

  Basic operation: comparison

```
int seqSearch(int n, const keytype S[], keytype x)
{
  location=1;
  while(location<=n && S[location] != x)
        location++;

  if(location > n) location = 0; // x not found
  return location;
}
```

Example: search strategy

- Binary search:
  Basic operation: comparison

```
Void Binsearch(int n, const keytype S[], keytype x, index& location)
{
  index low, high, mid;

  low = 1; high =n;  location=0;

  while (low <= high && location == 0)
  {
    mid = floor((low+high)/2);

    if (x==S[mid]) location = mid;

    else if (x< S[mid]) high = mid -1;

    else (low = mid +1);
  }
}
```

Example: number of comparisons

- Sequential search:

n	32	128	1024	1,048,576

- Binary search:

lg(n) + 1	6	8	11	21

Eg:

S[1],…., S[16],…., S[24], S[28], S[30], S[31], S[32]

(1st) (2nd) (3rd) (4th) (5th) (6th)

Analysis of Time Complexity

❑ Input size

❑ Basic operation

❑ Time complexity for the size of input, n

- T(n) : Every-case time complexity
- W(n): Worst-case time complexity
- A(n): Average-case time complexity
- B(n): Best-case time complexity

❑ Every-case time complexity examples

- Add array members: T(n) = n-1
- Matrix multiplication: n*n*n

Time Complexity Analysis

- Best Case
  - Smallest amount of time needed to run any instance of a given size
  - Not much practical use
- Worst Case
  - Largest amount of time needed to run any instance of a given size
  - Draconian view but hard to find an effective alternative

Time Complexity Analysis

- Average Case
  - Expected time for an instance of a given size
  - Hard to accurately model real instances by random distributions
  - Algorithm tuned for a specific distribution may suffer poor performance for other inputs

Well known problem

- Problem: Given a map of North America, find the best route from New York to Orlando?
- Many efficient algorithms
- Choose appropriate one (e.g., Floyd's algorithm for shortest paths using dynamic programming)

Another well known problem

- Problem: You are supposed to deliver newspapers to n houses in your town. How can you find the shortest tour from your home to everybody on your list and return to your home?
- One solution to traveling salesperson problem: dynamic programming

Some Examples of Commonly used algorithms

- Search (sequential, binary)
- Sort (mergesort, heapsort, quicksort, etc.)
- Traversal algorithms (breadth, depth, etc.)
- Shortest path (Floyd, Dijkstra)
- Spanning tree (Prim, Kruskal)
- Traveling salesman
- Knapsack
- Bin packing

How to design algorithms? Critical Thinking

- First of all, understand the problem!
- We saw that there are important problems and algorithms to solve them. How did people develop them?
- Clearly, it’s not easy and requires a lot of thoughts.
- Fortunately, there are well-established methods to design algorithms

Algorithm Design Methods

- Divide and conquer
- Dynamic programming
- Greedy
- Backtracking
- Branch and bound
- Linear programming
- Plus, keep thinking!

- Consider different approaches to solving a problem, such as dynamic programming and greedy approaches
- Analyze the merits of each: pros & cons
- Consider different implementations for a chosen approach
- Analyze the merit of the different implementation

Theory of NP completeness

- Important to reason the difficulty of problems
- Many common problems are NP-complete
  - Traveling salesperson, knapsack,...
  - NP: non-deterministic polynomial
- Fast algorithms for solving NP-complete problems probably don’t exist
  - Use approximate algorithms or heuristics

Efficiency

- The efficiency of an algorithm depends on the quantity of resources it requires

- Usually we compare algorithms based on their *time*
  - Sometimes also based on the *memory space* they need.

- The time required by an algorithm depends on the instance *size* and its *data*

Instance Size

- *Formally, size = number of bits needed to represent the instance in a computer.*
- We will usually be much less formal
- For search or sort, the size of each data is c bits or bounded by c bits
- Formally, the size of an input to sort with n records of c bits is nc. Informally, we use just n
- Why do we need the formal definition then?

Number problems

- Examples:
  - Factorial of 10, 10^6, 10^{15}
  - Fibonacci numbers
  - Multiplying, adding, dividing big numbers
- Size of each data is not constant – unbounded
- For these problems we should use the formal definition
- Size for a single number with value x is $O(\lg x)$ bits

Factorial

```
int factorial(int x)
{
int fac = 1;
for(i=2; i<=x; i++)
        fac = fac * i
return fac;
}
```

- Analysis
  - Number of operations: x-1
  - Number bits for value x: s = floor(lg x) +1
  - floor(lg x) = s - 1
  - $n >= 2^{s-1}$
    - → ***Exponential number of operations in terms of s***

Example: Factorial

- Compute factorial of an s bit number x. Its value is:

  $$x! = x * (x\text{-}1) * \ldots * 3 * 2 * 1$$

  - O(n) multiplications where $2^{s - 1} \leq x < 2^s$
  - Exponential time algorithm
- Example:
  - $s = 100$ bits (first bit is 1)
  - $2^{99} \leq x < 2^{100}$
  - $x > 0.6 * 10^{30}$

Fibonacci

```
int fib(int n)
{
  if (n <=1) return n;
  prev=0; cur=1;
  for (i=2; i<=n; i++)
  {
    next = cur + prev;
    prev = cur;
    cur= next;
  }
  return next
}
```

- In this code, x is replace with n. Does it make any difference in terms of time complexity analysis?
- Analysis
  - Number of operations: n-1
  - Number bits for value n: s = floor(lgn) +1
  - floor(lgn) = s - 1
  - n >= 2^{s-1}

    → ***Exponential number of operations in terms of s***

    → Better approach is dynamic programming: Remember the solution to a smaller instance