CS 575

Design and Analysis of Algorithms

Weiying Dai

1. Introduction

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

- Syllabus on myCourses

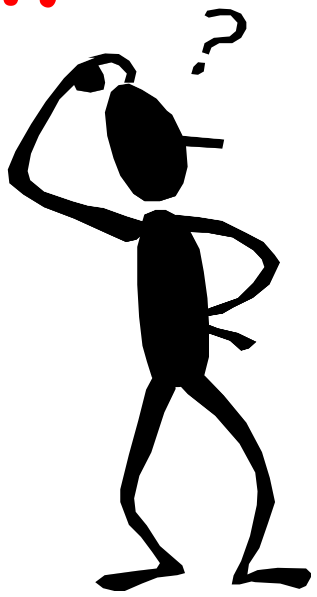
Course Objectives

- 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

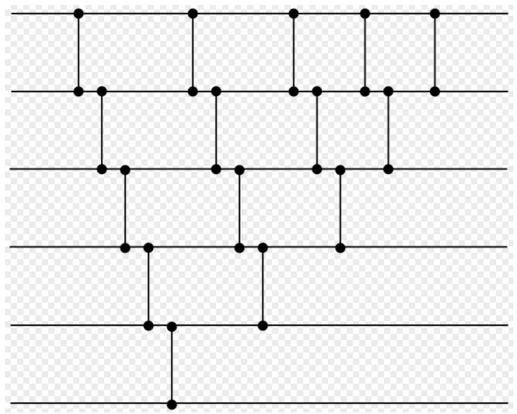
are the beauty of

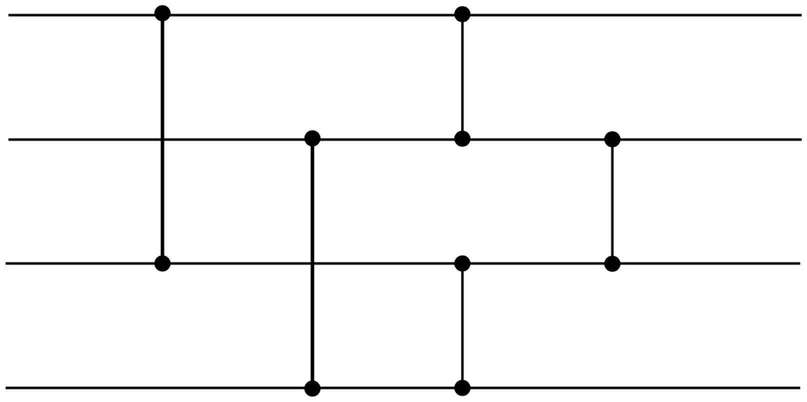
computation”

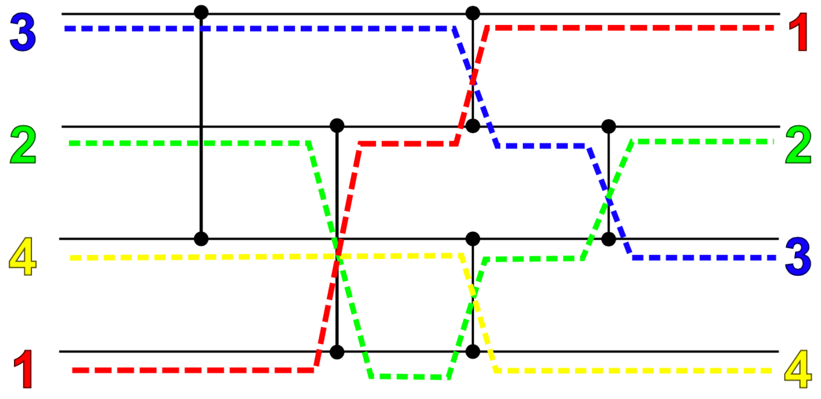
Hardware Design

- VLSI design
- Multiplication
- Search
- Sorting networks (comparators & wires)

Bubble sort for A[1], …, A[n-1]







Software

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

Economics

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

Job Interviews

- Job interviews contain many data structures and algorithm related questions
  - "Describe binary search"
  - "Describe an efficient algorithm that reads a file containing in random order all the numbers between 100 and 2035 except one, and determines the missing number.
    - The algorithm should be <u>linear in time</u> and use only <u>constant space</u>"

Job interview

- 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 they 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…

Programming

- Often you will be asked to program your solutions
  - “Write an efficient program to find the first character of a string that occurs only once”
  - “Write an efficient program to decide whether a linked list contains a cycle”

Math preparation

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

- Math review is uploaded to Brightspace

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

- Example problems
  - Determine whether a number x is in the list S of n integers
  - Sort a list S of n numbers in non-decreasing order
  - Matrix multiplication
- Technique: the approach or methodology used to solve the problem
- Algorithm: A step-by-step procedure for solving a problem.

Importance of Algorithm Efficiency

- ❑ Time: CPU cycles
- ❑ Storage: memory
- ❑ Example

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

  - nth 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;
  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 (x is larger than all the items in an array):

n	32	128	1024	1,048,576	4,294,967,296

- Binary search:

lg(n) + 1	6	8	11	21	33

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

- T(n) example
  - Add array members; Matrix multiplication; Exchange sort

    T(n) = n;   n*n*n   n(n-1)/2

Every-case time complexity

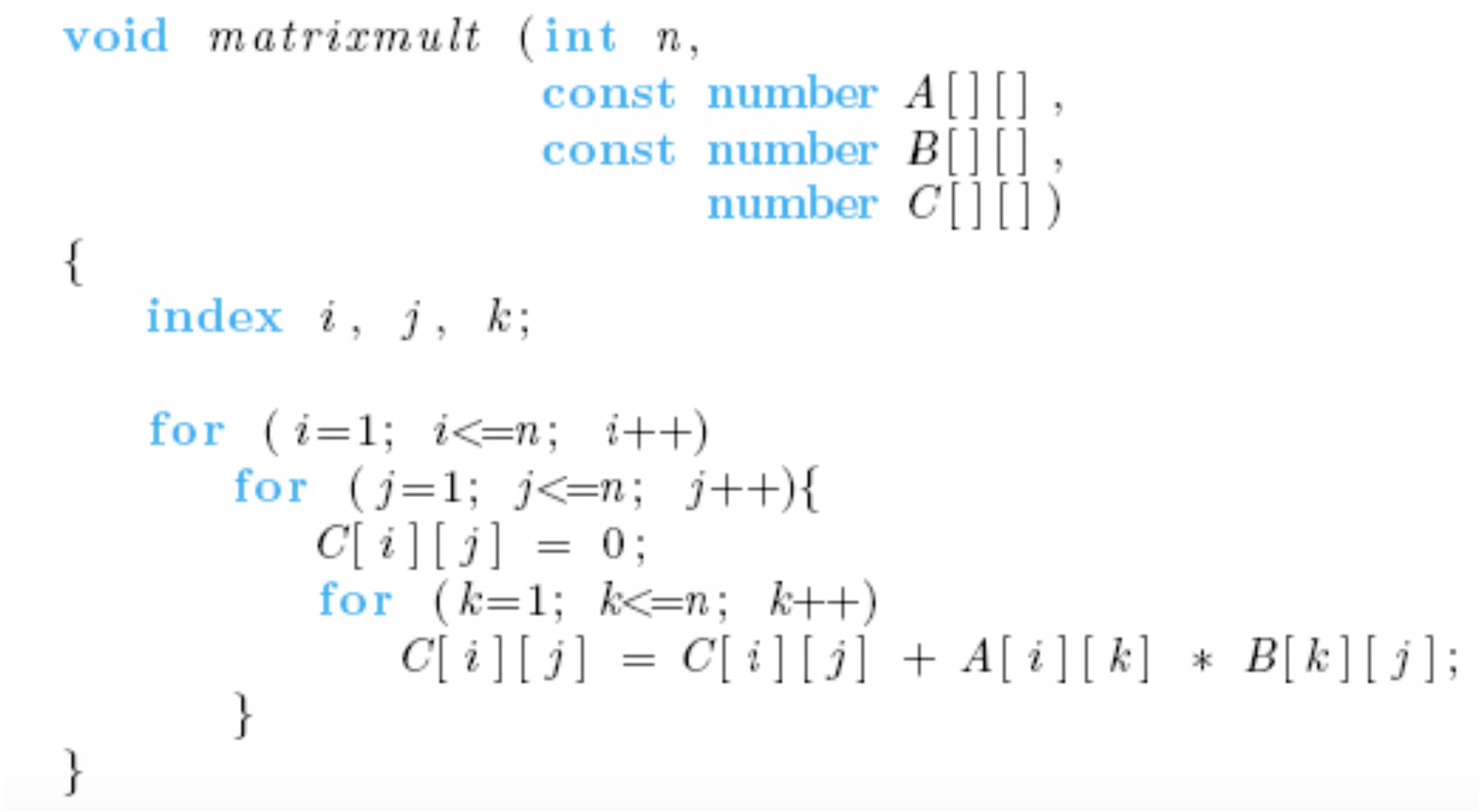
- **Add Array Members (T(n)=?)**
- Problem: Add all the members in the array S of n numbers
- Input: positive integer n, array of numbers S indexed 1~ n.
- Output: sum, the sum of the numbers in S.

```
number sum (int n, const number S[ ])
{
    index i;
    number result;

    result = 0;
    for (i = 1; i <= n; i++)
        result = result + S[i];
    return result;
}
```

Every-case time complexity

- **Matrix Multiplication (T(n)=?)**
- Problem: Determine the product of two $n \times n$ matrices.
- Inputs: a positive integer n, 2D arrays of numbers A and B.
- Outputs: a two-dimensional array of numbers C, containing the product of A and B.

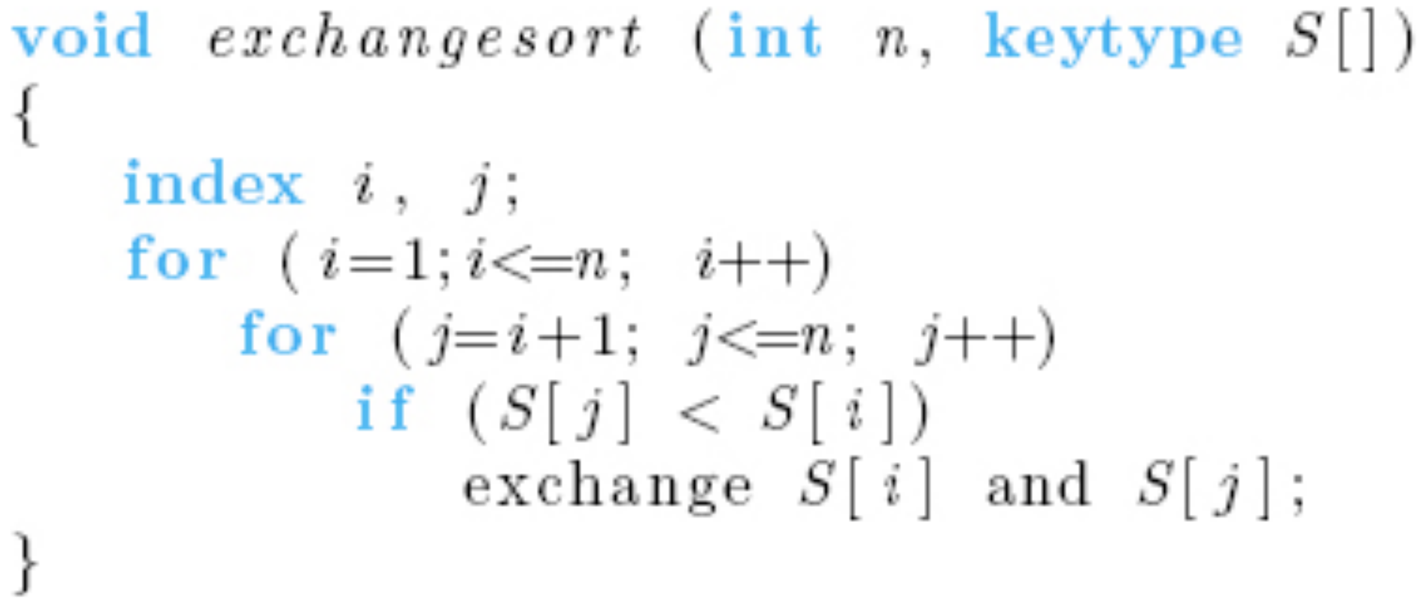


```
void matrixmult (int n,
                 const number A[][],
                 const number B[][],
                       number C[][])
{
   index i, j, k;

   for (i=1; i<=n; i++)
      for (j=1; j<=n; j++){
         C[i][j] = 0;
         for (k=1; k<=n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```

Every-case time complexity

- **Exchange Sort (T(n)=?)**
- Problem: Sort *n* keys in nondecreasing order.
- Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.
- Outputs: the array *S* containing the keys in nondecreasing order.



```
void exchangesort (int n, keytype S[])
{
    index i, j;
    for (i=1;i<=n; i++)
        for (j=i+1; j<=n; j++)
            if (S[j] < S[i])
                exchange S[i] and S[j];
}
```

Worst-case time complexity

- Insertion sort

```
for i = 2 to n
    for (k = i; **k > 1 and a[k] < a[k-1]**;
            k--)
            swap (a[k], a[k-1])
```

→ ***invariant: a[1..i] is sorted***

Worst-case time complexity in terms of number of comparisons:

In the inner "for" loop, for a given i, the comparison is done at most i-1 times

In total: $$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Worst-case time complexity

- Binary search (Recursive)

```
Index Binsearch(index low, index high)
{
   index mid;

   if (low > high)  return 0;

  else
  {
     mid = floor[(low+high)/2];

     if (x == S[mid])  return mid;
     else if (x < S[mid])  return Binsearch(low, mid-1);
     else  return  Binsearch(mid+1, high);
  }
 }
```

Worst-case Time complexity:

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

$\rightarrow$ $W(n) = \lg n + 1$

Time Complexity Analysis

- Best Case
  - Smallest amount of time needed to run any instance of a given size
  - Not much practical use
  - B(n) for sequential search?
- Worst Case
  - Largest amount of time needed to run any instance of a given size
  - Draconian view but hard to find an effective alternative
  - W(n) for sequential search?

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
  - A(n) for sequential search?

Worse case time analysis

- Most commonly used time complexity analysis
- *Because:*
  - Easier to compute than average case
  - Maximum time needed as a function of instance size
  - More useful than the best case

Worst case time analysis

- Drawbacks of comparing algorithms based on their worst case time:

  - An algorithm could be superior on average than another, although the worst case time complexity is not superior.

  - For some algorithms a worst case instance is very unlikely to occur in practice.

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 a few well established methods that we can use to design algorithms. (See the next slide)

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
- 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/input *size* and its *data*
  - Problem: Sort list of records.
  - Instances: (1, 10, 5) and (1,2,3,4, 1000, 27)

Size Examples

- **Search and sort**
  - **Size = *n* number of records in the list. For search ignore search key**
- **Graphs problems**
  - **Size = (|V| + |E|)**
  - **|V|: number of nodes**
  - **|E|: number of edges**
- **Matrix problems**
  - **Size = r*c**
  - **r: number of rows**
  - **c: number of columns**

Instance/input 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 we assume that the size of a record is c or bounded by c number of 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 where a number is expressed using several words
- For these problems we should use the formal definition
- Size for a single number with value n is $O(\lg n)$ bits

Factorial

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

- 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***

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
}
```

- 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