Welcome to IS 2610

Introduction

Course Information

- Lecture:
  - James B D Joshi
  - Mondays: 3:00-5.50 PM
    - One (two) 15 (10) minutes break(s)
  - Office Hours: Wed 1:00-3:00PM/Appointment

- Pre-requisite
  - one programming language
Course material

- Textbook

- References
  - *Fundamentals of Data Structures* by Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed Hardcover
    - March 1992 / 0716782502
  - The C Programming language, Kernigham & Ritchie (Programming)
  - Other material will be posted (URLs for tutorials)

Course outline

- *Introduction to Data Structures and Analysis of Algorithms*
  - Analysis of Algorithms
  - Elementary/Abstract data types
  - Recursion and Trees

- *Sorting Algorithms*
  - Selection, Insertion, Bubble, Shellsort
  - Quicksort
  - Mergesort
  - Heapsort
  - Radix sort

- *Searching*
  - Symbol tables
  - Balanced Trees
  - Hashing
  - Radix Search

- *Graph Algorithms*
Grading

- Quiz 10% (in the beginning of the class; on previous lecture)
- Homework/Programming Assignments 40% (typically every week)
- Midterm 25%
- Comprehensive Final 25%

Course Policy

- Your work MUST be your own
  - Zero tolerance for cheating
  - You get an F for the course if you cheat in anything however small – NO DISCUSSION
- Homework
  - There will be penalty for late assignments (15% each day)
  - Ensure clarity in your answers – no credit will be given for vague answers
  - Homework is primarily the GSA’s responsibility
  - Solutions/theory will be posted on the web
- Check webpage for everything!
  - You are responsible for checking the webpage for updates
Overview

- Algorithm
  - A problem-solving method suitable for implementation as a computer program
- Data structures
  - Objects created to organize data used in computation
- Data structure exist as the by-product or end product of algorithms
  - Understanding data structure is essential to understanding algorithms and hence to problem-solving
  - Simple algorithms can give rise to complicated data-structures
  - Complicated algorithms can use simple data structures

Why study Data Structures (and algorithms)

- Using a computer?
  - Solve computational problems?
  - Want it to go faster?
  - Ability to process more data?
- Technology vs. Performance/cost factor
  - Technology can improve things by a constant factor
  - Good algorithm design can do much better and may be cheaper
  - Supercomputer cannot rescue a bad algorithm
- Data structures and algorithms as a field of study
  - Old enough to have basics known
  - New discoveries
  - Burgeoning application areas
  - Philosophical implications?
Simple example

- Algorithm and data structure to do matrix arithmetic
  - Need a structure to store matrix values
    - Use a two dimensional array: $A[M, N]$
  - Algorithm to find the largest element
    
    ```
    largest = A[0][0];
    for (i=0; i < M; i++)
      for (i=0; i < N; i++)
        if (A[i][j]>largest) then
          largest= A[i][j];
    ```

    How many times does the if statement gets executed?

Another example: Network Connectivity

- Network Connectivity
  - Nodes at grid points
  - Add connections between pairs of nodes
  - Are A and B connected?

![Network Connectivity Diagram]
Network Connectivity

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
<th>Evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>3 4</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>4 9</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8 0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2 3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5 6</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>(2-3-4-9)</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>5 9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7 3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>4 8</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>(5-6)</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>(2-3-4-8-0)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6 1</td>
</tr>
</tbody>
</table>

Union-Find Abstraction

- What are the critical operations needed to support finding connectivity?
  - N objects – *N can be very large*
    - Grid points
  - FIND: test whether two objects are in same set
    - Is A connected to B?
  - UNION: merge two sets
    - Add a connection
- Define Data Structure to store connectivity information and algorithms for UNION and FIND
Quick-Find algorithm

- Data Structure
  - Use an array of integers – one corresponding to each object
  - Initialize $id[i] = i$
  - If p and q are connected they have the same id
- Algorithmic Operations
  - FIND: to check if p and q are connected, check if they have the same id
  - UNION: To merge components containing p and q, change all entries with $id[p]$ to $id[q]$
- Complexity analysis:
  - FIND: takes constant time
  - UNION: takes time proportional to N

for (i = 0; i < N; i++)
  $id[i] = i$;

if ($id[p] == id[q]$)
  // already connected

pid = $id[p]$;
for (i = 0; i < N; i++)
  if ($id[i] == pid$)
    $id[i] = id[q]$;

Quick-find

<table>
<thead>
<tr>
<th>p-q</th>
<th>array entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-4</td>
<td>0 1 2 4 5 6 7 8 9</td>
</tr>
<tr>
<td>4-9</td>
<td>0 1 2 9 9 5 6 7 8 9</td>
</tr>
<tr>
<td>8-0</td>
<td>0 1 2 9 9 5 6 7 0 9</td>
</tr>
<tr>
<td>2-3</td>
<td>0 1 9 9 9 5 6 7 0 9</td>
</tr>
<tr>
<td>5-6</td>
<td>0 1 9 9 9 6 6 7 0 9</td>
</tr>
<tr>
<td>5-9</td>
<td>0 1 9 9 9 9 9 7 0 9</td>
</tr>
<tr>
<td>7-3</td>
<td>0 1 9 9 9 9 9 9 0 9</td>
</tr>
<tr>
<td>4-8</td>
<td>0 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>6-1</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
Complete algorithm

```c
#include <stdio.h>
#define N 10000
main()
{ int i, p, q, t, id[N];
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("d% %d
", &p, &q) == 2
  {
    if (id[p] == id[q]) continue;
    for (pid = id[p], i = 0; i < N; i++)
      if (id[i] == pid) id[i] = id[q];
    printf("s %d
", p, q);
  }
}
```

- **Complexity \(M \times N\)**
  - For each of \(M\) union operations we iterate for loop at \(N\) times

Quick-Union Algorithm

- **Data Structure**
  - Use an array of integers – one corresponding to each object
    - Initialize \(id[i] = i\)
  - If \(p\) and \(q\) are connected they have same root

- **Algorithmic Operations**
  - **FIND**: to check if \(p\) and \(q\) are connected, check if they have the same root
    ```c
    for (i = p; i != id[i]; i = id[i]);
    for (j = q; j != id[j]; j = id[j]);
    if (i == j) // connected
    ```
  - **UNION**: Set the id of the \(p\)'s root to \(q\)'s root

- **Complexity analysis**:
  - **FIND**: takes time proportional to the depth of \(p\) and \(q\) in tree
  - **UNION**: takes constant times
Complete algorithm

```c
#include <stdio.h>
#define N 10000
main()
{ int i, p, q, t, id[N];
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d\n", &p, &q) == 2)
  {
    for (i = p; i != id[i]; i = id[i])
      for (j = q; j != id[j]; j = id[j])
        if (i == j)      // connected
          id[i] = j;
    printf("s %d\n", p, q);
  }
}
```

Quick-Union

<table>
<thead>
<tr>
<th>p-q</th>
<th>array entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-4</td>
<td>0 1 2 4 5 6 7 8 9</td>
</tr>
<tr>
<td>4-9</td>
<td>0 1 2 4 9 5 6 7 8 9</td>
</tr>
<tr>
<td>8-0</td>
<td>0 1 2 4 9 5 6 7 0 9</td>
</tr>
<tr>
<td>2-3</td>
<td>0 1 9 4 9 5 6 7 0 9</td>
</tr>
<tr>
<td>5-6</td>
<td>0 1 9 4 9 6 6 7 0 9</td>
</tr>
<tr>
<td>5-9</td>
<td>0 1 9 4 9 6 9 7 0 9</td>
</tr>
<tr>
<td>7-3</td>
<td>0 1 9 4 9 6 9 9 0 0</td>
</tr>
<tr>
<td>4-8</td>
<td>0 1 9 4 9 6 9 9 0 0</td>
</tr>
<tr>
<td>6-1</td>
<td>1 1 9 4 9 6 9 9 0 0</td>
</tr>
</tbody>
</table>
Complexity of Quick-Union

- Less computation for UNION and more computation for FIND
- Quick-Union does not have to go through the entire array for each input pair as does the Union-find
- Depends on the nature of the input
  - Assume input 1-2, 2-3, 3-4,…
  - Tree formed is linear!
- More improvements:
  - Weighted Quick-Union
  - Weighted Quick-Union with Path Compression

Analysis of algorithm

- Empirical analysis
  - Implement the algorithm
  - Input and other factors
    - Actual data
    - Random data (average-case behavior)
    - Perverse data (worst-case behavior)
  - Run empirical tests
- Mathematical analysis
  - To compare different algorithms
  - To predict performance in a new environment
  - To set values of algorithm parameters
Growth of functions

- Algorithms have a primary parameter $N$ that affects the running time most significantly
  - $N$ typically represents the size of the input—e.g., file size, no. of chars in a string; etc.
- Commonly encountered running times are proportional to the following functions
  - 1: Represents a constant
  - $\log N$: Logarithmic
  - $N$: Linear time
  - $N \log N$: Linearithmic(?)
  - $N^2$: Quadratic
  - $N^3$: Cubic
  - $2^N$: Exponential

Some common functions

<table>
<thead>
<tr>
<th>$\lg N$</th>
<th>$N^{0.5}$</th>
<th>$N$</th>
<th>$N \lg N$</th>
<th>$N (\lg N)^2$</th>
<th>$N^2$</th>
<th>$2^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>10</td>
<td>33</td>
<td>110</td>
<td>100</td>
<td>1042</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>100</td>
<td>664</td>
<td>444</td>
<td>10000</td>
<td>$2^{10} \times 10^9$ $1042^{10}$</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>1000</td>
<td>9966</td>
<td>99317</td>
<td>1000000</td>
<td>?</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>10000</td>
<td>132877</td>
<td>1765633</td>
<td>100000000</td>
<td>?</td>
</tr>
<tr>
<td>17</td>
<td>316</td>
<td>100000</td>
<td>1660964</td>
<td>27588016</td>
<td>10000000000</td>
<td>?</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>1000000</td>
<td>19931569</td>
<td>397267426</td>
<td>1000000000000</td>
<td>?</td>
</tr>
</tbody>
</table>
### Special functions and mathematical notations

- **Floor function**: \( \lfloor x \rfloor \)
  - Largest integer less than or equal to \( x \)
  - e.g., \( \lfloor 5.16 \rfloor = ? \)

- **Ceiling function**: \( \lceil x \rceil \)
  - Smallest integer greater than or equal to \( x \)
  - e.g., \( \lceil 5.16 \rceil = ? \)

- **Fibonacci**: \( F_N = F_{N-1} + F_{N-2} \); with \( F_0 = F_1 = 1 \)
  - Find \( F_2 = ? \) \( F_4 = ? \)

- **Harmonic**: \( H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \)

- **Factorial**: \( N! = N.(N-1)! \)

- **Logarithm**: \( \log_e N = \ln N; \log_2 N = \lg N \)

### Big O-notation – Asymptotic expression

- \( g(N) = O(f(N)) \) (read \( g(N) \) is said to be \( O(f(N)) \)) iff there exist constants \( c_0 \) and \( N_0 \) such that \( 0 = g(N) = c_0 f(N) \) for all \( N > N_0 \)

- Can \( N^2 = O(n) \)?
- Can \( 2^N = O(N^M) \)?
**Big-O Notation**

- **Uses**
  - To bound the error that we make when we ignore small terms in mathematical formulas
  - Allows us to focus on leading terms
  - Example:
    - \( N^2 + 3N + 4 = O(N^2), \) since \( N^2 + 3N + 4 < 2N^2 \) for all \( n > 10 \)
    - \( N^2 + N + N \lg N + \lg N + 1 = O(N^2) \)
  - To bound the error that we make when we ignore parts of a program that contribute a small amount to the total being analyzed
  - To allow us to classify algorithms according to the upper bounds on their total running times

---

**Ω(\(f(n)\)) and Θ(\(f(n)\))**

- \( g(N) = Ω(f(N)) \) (read \( g(N) \) is said to be \( Ω(f(N)) \)) iff there exist constants \( c_0 \) and \( N_0 \) such that \( 0 = g(N) = c_0 f(N) \) for all \( N > N_0 \)

- \( g(N) = Θ(f(N)) \) (read \( g(N) \) is said to be \( Ω(f(N)) \)) iff there exist constants \( c_0, c_1 \) and \( N_0 \) such that \( c_1 f(N) = g(N) = c_1 f(N) \) for all \( N > N_0 \)
Basic Recurrences

- Principle of recursive decomposition
  - decomposition of problems into one or more smaller ones of the same type
  - Use solutions for the sub-problems to get solution of the problem
- Example 1:
  - Loops through a loop and eliminates one item
  - \( C_N = C_{N-1} + N \), for \( N = 2 \) with \( C_1 = 1 \)
  - \( = C_{N-2} + (N-1) + N \)
  - \( = C_{N-3} + (N-2) + (N-1) + N \)
  - \( \ldots \)
  - \( = 1 + 2 + \ldots + (N-2) + (N-1) + N = N(N+1)/2 \)
  - Therefore, \( C_N = O(N^2) \)

Basic Recurrences

- Recurrence relations
  - Captures the dependence of the running time of an algorithm for an input of size \( N \) on its running time for small inputs
- Example 2:
  - formula for recursive programs for that halves the input in one step
    - \( C_N = C_{N/2} + 1 \), for \( N = 2 \) with \( C_1 = 1 \); let \( C_N = \lg N \), and \( N = 2^n \)
      - \( = C_{N/2} + 1 + 1 \)
      - \( = C_{N/4} + 1 + 1 + 1 \)
      - \( \ldots \)
      - \( = C_{N/N} + n = 1 + n \)
  - Therefore, \( C_N = O(n) = O(\lg N) \)
Basic Recurrences

- let $C_N = \lg N$, and $N = 2^n$
- Show that $C_N = N \lg N$ for
  - $C_N = 2^{C_{N/2}} + N$; for $N = 2$ with $C_1 = 0$;