Introduction

Algorithms

- Method for solving problems suitable for computer implementation
  - Generally independent of computer hardware characteristics
  - Possibly suitable for many different programming languages
- Input and output for algorithms
- Problem must be well-specified
  - Old adage – Garbage In Garbage Out (GIGO)
  - Programs always do a specific and well-defined task
- Problem instance
- Algorithm correctness – correct solution for every instance of input
- Incorrect algorithm may not halt
- Pseudocode – No attention paid to data structures
- Data structures
  - Methods of organizing data used in the computations

Algorithms + Data Structures ⇒ Programs

Why study algorithms?

- Know the existing ways of solving problems
- Proper choice of algorithms, depending on efficiency and other constraints
  - Time and space as a function of amount of input data
  - Y2K problem
- Division of larger problems into smaller subproblems that can be easily solved
- Gain an understanding of the fundamental ways of problem-solving in computers, regardless of available hardware and software
  - Similar to the study of basic number manipulation techniques (addition of numbers) when a calculator can do the job faster
- Correct amount of optimization of code
  - Is it worth it to save three milliseconds on a program and spend five hours into the optimization?

Connectivity Problem

- Problem to determine, given a set of pairs of connected nodes, whether there are redundant pair of nodes which may be connected through other nodes (for example, forming a cycle)
Consider the following example:

3-4  3-4
4-9  4-9
8-0  8-0
2-3  2-3
5-6  5-6
2-9  2-3-4-9
5-9  5-9
7-3  7-3
4-8  4-8
5-6  5-6
0-2  0-8-4-3-2
6-1  6-1

The pair of nodes in second column are uniquely connected; the ones missed can be connected to each other through other nodes as shown in third column.

Other variations
- Find out a way of connecting all points in the network to minimize the amount of connecting material used
- Find out a way to connect all points in a network such that you can go from one point to another in minimum amount of time

How can you tell inside of a computer program whether two given points are connected?
- Problem specification point: Find out whether a pair of points is connected, and not all possible ways to connect the pair
- Heuristic: A connectivity algorithm will never give you more than \( n - 1 \) connected pairs for a graph with \( n \) nodes
  * Implication: You can stop looking for new pairs once you have found \( n - 1 \) connected pairs, as the new pairs would invariably be connected
- Input to this algorithm is a graph and the output is a spanning tree for that graph

Union-find algorithms

- Purpose: In the above connectivity problem, we tried to find if a given pair is already connected
  - Each time we get a new pair, we have to find if it is a new connection
  - This information is added (union) to a set of connection for future reference
- First write a simple prototype to check that we understand the problem and have a solution
- First idea:
  - Save all pairs
  - Write a function to pass through them to discover which ones are connected
- Quick-find algorithm
  - Use an array of nodes to save the connectivity
  - Two nodes are connected if and only if their corresponding entries in the array are equal
  - We spend time in building up the array but the find part is just the lookup, hence the name quick-find
Algorithm for quick-find

```plaintext
algorithm quick-find

/* Initialize the array of variables */
1  for i ← 0 to N - 1 do
2     id[i] = i
3  for each pair ⟨p, q⟩
4     if ( id[p] == id[q] ) continue
5       tmp = id[p]
6     for i ← 1 to N do
7         if ( id[i] == tmp )
8             id[i] = id[q]
```

**Property 1** The quick-find algorithm executes at least $mn$ instructions to solve a connectivity problem with $m$ pairs of $n$ objects.

- Quick-union algorithm
  - Complementary to quick-find
  - Based on array just like quick-find but with a different interpretation
  - Each object in the set points to another object in the same set, and membership is determined by following the pointers
  - Objects are in the same set if and only if this process of pointing ends up at the same object; if they are not in the same set, we end up at different objects that point to themselves
  - Union is formed by linking the two sets if they are different (quick-union)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Quick-union is an improvement over quick-find because it removes quick-find’s main liability that the program requires at least $NM$ instructions to process $M$ union operations in $N$ objects
Property 2 For \(m > n\), the quick-union algorithm could take more than \(\frac{mn}{2}\) instructions to solve a connectivity problem with \(m\) pairs of \(n\) objects.

- Weighted quick-union algorithm
  - What if input pairs come as \(1-2, 2-3, 3-4, \ldots\)
  - Find operation for \(n\)th object requires following \(n-1\) pointers
  - Average number of pointers for the first \(n\) objects is
    \[
    \frac{(0 + 1 + 2 + \cdots + (n-1))}{n} = \frac{n-1}{2}
    \]
  - Problem solved by weighted quick-union algorithm
  - Keep track of number of nodes in the tree and always connect the smaller to the larger

Property 3 The weighted quick-union algorithm follows at most \(\log n\) pointers to determine whether two of \(n\) objects are connected.

Selection sort

- Algorithm
  
  ```algorithms
  algorithm selection_sort (a)
  1 for i ← 1 to n-1 do
  2   min ← i
  3 for j ← i+1 to n do
  4     if a[j] < a[min] then
  5       min ← j
  6   exchange(a[i], a[min])
  ```

- Pseudocode convention
  - Indentation for block structure
  - `while`, `for`, `repeat`, `if`, `then`, `else` same as Pascal
  - Comment indicated by `/* ... */`
  - `i ← j ← e`
  - No global variables without explicit declaration
  - Array accessed by index to array name
  - Parameters passed by value

Algorithm analysis

- Predict the resources required by algorithm
  - Space requirements
  - Time requirements
- Model of the machine – RAM
- More data \(\Rightarrow\) more time
  - Input size
    \* Number of items
* Number of bits
* Number of pairs (vertices and edges in graph)
  - Running time
  * Number of primitive operations or “steps” executed
  * Constant amount of time for each line of pseudocode

- Analysis of selection sort

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm selection_sort (a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 for i ← 1 to n-1 do</td>
<td>$c_1$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>2 min ← i</td>
<td>$c_2$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>3 for j ← i+1 to n do</td>
<td>$c_3$</td>
<td>$\sum_{i=1}^{n-1}(n-i)$</td>
</tr>
<tr>
<td>4 if a[j] &lt; a[min] then</td>
<td>$c_4$</td>
<td>$\sum_{i=1}^{n-1}(n-i)$</td>
</tr>
<tr>
<td>5 min ← j</td>
<td>$c_5$</td>
<td>$\sum_{i=1}^{n-1}(n-i)$</td>
</tr>
<tr>
<td>6 exchange(a[i],a[min])</td>
<td>$c_6$</td>
<td>$n-1$</td>
</tr>
</tbody>
</table>

$$T(n) = c_1(n-1) + c_2(n-1) + c_3 \sum_{i=1}^{n-1}(n-i) + c_4 \sum_{i=1}^{n-1}(n-i) + c_5 \sum_{i=1}^{n-1}(n-i) + c_6(n-1)$$

$$= (c_1 + c_2 + c_6)(n-1) + (c_3 + c_4 + c_5) \sum_{i=1}^{n-1}(n-i)$$

$$= (c_1 + c_2 + c_6)(n-1) + (c_3 + c_4 + c_5) \frac{n(n-1)}{2}$$

$$= \frac{(c_3 + c_4 + c_5)}{2} n^2 + (c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6)n - (c_1 + c_2 + c_6)$$

- Quadratic function of $n$

- Worst case analysis
  - Upper bound on running time for any input of size $n$

- Average case analysis
  - All inputs equally likely
  - Often roughly as bad as the worst case

- Order of growth
  - Actual cost of statement is ignored
  - Constants are ignored as they represent the cost of statement
  - Consider only the leading term of expression
  - Worst case running time of selection sort $\Theta(n^2)$
  - Lower order of growth $\Rightarrow$ more efficient algorithm
  - Significant only for large number of inputs

---

Designing Algorithms

- Selection sort based on incremental approach
Select the smallest element in the array and exchange it with the first element
Repeat with the remainder of the array

• Divide-and-conquer

**Divide** the problem into a number of smaller subproblems

**Conquer** the subproblems by solving them recursively

**Combine** the solutions to the subproblems into the solution for the original problem

• Mergesort algorithm

**Divide** the \( n \)-element sequence into two \( n/2 \)-element subsequences

**Conquer** by recursively mergesorting the two subsequences

**Combine** by merging the two sorted subsequences to produce the sorted sequence

```python
def mergesort(a, p, r):
    if p < r:
        q = trunc((p + r) / 2)
        mergesort(a, p, q)
        mergesort(a, q+1, r)
        merge(a, p, q, r)
```

• Analyzing divide-and-conquer algorithms

- Running time described by a recurrence equation
- Based on three steps
  - Problem size \( n \) ⇒ Run time \( T(n) \)
  - Small problem size \( n \leq c \) ⇒ straightforward solution in time \( \Theta(1) \)
  - Divide the problem into \( a \) subproblems; each being \( 1/b \) of the original
    - \( D(n) \) – Time to divide the problem into subproblems
    - \( C(n) \) – Time to combine the solution to original problem
- Recurrence
  \[
  T(n) = \begin{cases} 
  \Theta(1) & \text{if } n \leq c \\
  aT(n/b) + D(n) + C(n) & \text{otherwise}
  \end{cases}
  \]
- Analysis of mergesort