CSCI-GA.1144-001

PAC II

Lecture 7: Algorithms

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Scenario 1: Amazon buying Adventure

- Early April 2011: A scientist at UC-Berkeley logged on to Amazon.com to buy an extra book for his lab.
- He usually pays $35-$40 per copy
- But on that day, he found 2 used copies, one priced at $1,730,045 the other at $2,198,177!!
- He thought it was just a mistake or a joke
- He re-checked the following day and the prices were $2,194,443 and $2,788,233!!
- The escalation continued for two weeks with the price peaking on April 18th at $28,698,655 (+ $3.99 shipping)!!
Scenario 2: Flash Crash (one of several)

- Early on May 6, 2010: stock market was hit by unsettling developments in Greece.... BUT
- At 2:42pm (EST) markets start dropping into a free fall
- At 2:47pm (i.e. 300 seconds later): Dow Jones was down 998.5 points (the largest single day drop in history!)
- Nearly $1 Trillion of wealth fell into the electronic ether!!
- Some share prices crashed to one penny ($0.01) rendering billion-dollar companies worthless!
- Dow Jones recovered 500 points in less than 3 minutes!!
What Happened??

- **Scenario 1:**
  - Algorithms used by Amazon to price books got into price war!
  - One of the seller’s algorithms was programmed to price the book slightly higher than the competitor’s price.
  - The second algorithm, in turn, increased its price!
  - Things didn’t turn to normal until a human being stepped in and overrode the system.

- **Scenario 2:**
  - We don’t know till today!!
  - Explanation 1: Some of the blame was directed to a city money manager whose algorithm sold $4B worth of stock too quickly.
  - Explanation 2: Group of traders who conspired to send things down all at once through a coordinated algorithms
As we put more and more of our world under the control of algorithms, we can lose track of who – or what – is pulling the strings.

from Christopher Steiner’s book “Automate This: How Algorithms Came to Rule our World” .... (from which I got the previous two scenarios too!).
Well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

A tool for solving a well-specified computational problem.

The statement of the problem specifies in general terms the desired input/output relationship.
Problem Statement

- Problem specifications have two parts:
  1. the set of allowed input instances,
  2. the required properties of the algorithm’s output
Questions

• What is the difference between a program and an algorithm?
• Is error handling part of the algorithm? or the HLL program?
• Does your algorithm need to produce just a correct result? or always the best result?
• If computers were infinitely fast and memory was free, would you have any reasons to study algorithms?
Can We Solve Anything With a Computer?

• **Undecidable**
  – Cannot be solved by an algorithm
  – e.g. Halting problem (given a program and inputs for it, decide whether it will run forever or will eventually halt.)

• **Unsolvable**
  – No finite algorithm
  – e.g. Goldbach’s conjecture (Every even number greater than 2 can be written as the sum of two primes.)

• **Intractable**
  – Unreasonable amount of time and resources
“Steps” of an algorithm

• Finite number
• Unambiguous
• very specific
• can be carried out in a finite amount of time in a deterministic way
• Since we can only input, store, process & output data on a computer, the instructions in our algorithms will be limited to these functions
Algorithm Properties

• It must be correct.
• It must be composed of a series of concrete steps.
• There can be no ambiguity as to which step will be performed next.
• It must be composed of a finite number of steps.
• It must terminate.
Algorithm Is Different Than A HLL Program

• In algorithms you do not need to use strict syntax
• You can present an algorithm in pseudocode, flowchart, ...
• Pseudocode is not concerned with issues of software engineering (e.g. error handling, abstraction, modularity, ...)
Pseudocode Algorithm

• **Example**: Write an algorithm to determine a student’s final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.
Pseudocode Algorithm

Pseudocode:

• *Input a set of 4 marks*
• *Calculate their average by summing and dividing by 4*
• *if average is below 50*
  
  Print “FAIL”

  *else*

  Print “PASS”
Pseudocode Algorithm

- Detailed Algorithm

Step 1: Input M1, M2, M3, M4
Step 2: GRADE ← (M1 + M2 + M3 + M4) / 4
Step 3: if (GRADE < 50) then
    Print “FAIL”
else
    Print “PASS”
endif
Flowchart

A Flowchart

– shows logic of an algorithm
– emphasizes individual steps and their interconnections
– e.g. control flow from one action to the next
Flowchart Symbols

Basic

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Use in Flowchart</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oval</td>
<td><img src="image" alt="Oval" /></td>
<td>Denotes the beginning or end of the program</td>
</tr>
<tr>
<td>Parallelogram</td>
<td><img src="image" alt="Parallelogram" /></td>
<td>Denotes an input operation</td>
</tr>
<tr>
<td>Rectangle</td>
<td><img src="image" alt="Rectangle" /></td>
<td>Denotes a process to be carried out e.g. addition, subtraction, division etc.</td>
</tr>
<tr>
<td>Diamond</td>
<td><img src="image" alt="Diamond" /></td>
<td>Denotes a decision (or branch) to be made. The program should continue along one of two routes. (e.g. IF/THEN/ELSE)</td>
</tr>
<tr>
<td>Hybrid</td>
<td><img src="image" alt="Hybrid" /></td>
<td>Denotes an output operation</td>
</tr>
<tr>
<td>Flow line</td>
<td><img src="image" alt="Flow line" /></td>
<td>Denotes the direction of logic flow in the program</td>
</tr>
</tbody>
</table>
Example

Step 1: Input M1, M2, M3, M4
Step 2: GRADE ← (M1 + M2 + M3 + M4)/4
Step 3: if (GRADE < 50) then
   Print “FAIL”
else
   Print “PASS”
endif
Example

Problem: Robot Tour Optimization
Input: A set $S$ of $n$ points in the plane.
Output: What is the shortest cycle tour that visits each point in the set $S$?
The above algorithm is:

• Simple to understand and implement
• Makes sense

And ... WRONG! Does not produce the shortest path!
Example

NearestNeighbor($P$)
Pick and visit an initial point $p_0$ from $P$
$p = p_0$
$i = 0$
While there are still unvisited points
$i = i + 1$
Select $p_i$ to be the closest unvisited point to $p_{i-1}$
Visit $p_i$
Return to $p_0$ from $p_{n-1}$

What can we do?

This is what the above alg. produces

This is the optimal solution.
Example

ClosestPair(P)
   Let n be the number of points in set P.
   For i = 1 to n - 1 do
      $d = \infty$
      For each pair of endpoints (s, t) from distinct vertex chains
         if $\text{dist}(s, t) \leq d$ then $s_m = s$, $t_m = t$, and $d = \text{dist}(s, t)$
      Connect $(s_m, t_m)$ by an edge
   Connect the two endpoints by an edge

This one will produce the optimal solution of the previous example.

But:
Hmmm ...

• Looks like for this problem any algorithm can produce a very bad result for some inputs 😞

• This example we just saw is a classical problem called The Traveling Salesman Problem (TSP)
The traveling salesman must travel to $n$ different towns in his area each month in order to deliver something important. Each town is a different distance away from his town and from each other town. How do you figure out a route that will minimize the distance traveled?
Brute Force?

- Enumerate all possible routes
  - For 10 towns for example there are 10! \((3,628,800)\)
- Choose the shortest.
- This is called **brute force algorithm**.

**OptimalTSP(P)**

\[
d = \infty
\]

For each of the \(n!\) permutations \(P_i\) of point set \(P\)

If \((\text{cost}(P_i) \leq d)\) then \(d = \text{cost}(P_i)\) and \(P_{min} = P_i\)

Return \(P_{min}\)
Is Brute Force a Good Solution?

Take Home Lesson: There is a fundamental difference between *algorithms*, which always produce a correct result, and *heuristics*, which may usually do a good job but without providing any guarantee.
How Do We Judge Algorithms?

- Correctness
- Efficiency
  - Speed
  - Memory
- Algorithm analysis is predicting the resources that the algorithm requires
- Algorithms can be understood and studied in a language and machine-independent manner.
Machine Model → RAM

- Random Access Machine
- Instructions are executed one after the other.
- Basic instructions (arithmetic, logic, data movement) take fixed amount of time
- Memory is infinite
- We need a way that summarizes the behavior of an algorithm executed on RAM
Worst- / average- / best-case

• Worst-case running time of an algorithm
  – The longest running time for any input of size \( n \)
  – An upper bound on the running time for any input
    ⇒ guarantee that the algorithm will never take longer
  – Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  – The worst case can occur fairly often
    • E.g. in searching a database for a particular piece of information

• Best-case running time
  – sort a set of numbers in increasing order; and the data is already in increasing order

• Average-case running time
  – May be difficult to define what “average” means
The Big Oh Notation

- A way of giving an approximation of the amount of computation done by an algorithm given the input size.
- Ignores the difference between multiplicative constants: $f(n) = 2n$ and $g(n) = n$ are identical in Big Oh analysis.
The Big Oh Notation

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus there exists some constant $c$ such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough $n$ (i.e., $n \geq n_0$ for some constant $n_0$).

- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus there exists some constant $c$ such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.

- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants $c_1$ and $c_2$ such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. 
The Big Oh Notation

**Problem:** Is $2^{n+1} = \Theta(2^n)$?
Example

- \( f(n) = 2n + 5 \)
  \( g(n) = n \)
- Consider the condition
  \[ 2n + 5 \leq n \]
  will this condition ever hold? No!
- How about if we multiply a constant by \( n \)?
  \[ 2n + 5 \leq 3n \]
  the condition holds for values of \( n \) greater than or equal to 5
- This means we can select \( c = 3 \) and \( n_0 = 5 \) and \( f(n) \rightarrow O(n) \)
Example (cont’d)

2n+5 is $\mathcal{O}(n)$
Is It Wise to Ignore Constants?

• If two algorithms one is \(O(n^2)\) and the other \(O(\log n)\)
  – one is \(C_1 n^2\) and the other \(C_2 \log n\)
  – What if \(C_2\) is much bigger than \(C_1\)?

<table>
<thead>
<tr>
<th>(n)</th>
<th>(f(n))</th>
<th>(\lg n)</th>
<th>(n)</th>
<th>(n \lg n)</th>
<th>(n^2)</th>
<th>(2^n)</th>
<th>(n!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>0.003</td>
<td>0.01</td>
<td>0.033</td>
<td>0.1</td>
<td>1</td>
<td>3.63 ms</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0.004</td>
<td>0.02</td>
<td>0.086</td>
<td>0.4</td>
<td>1 ms</td>
<td>77.1 years</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>0.005</td>
<td>0.03</td>
<td>0.147</td>
<td>0.9</td>
<td>1 sec</td>
<td>8.4 \times 10^{15}\ yrs</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>0.005</td>
<td>0.04</td>
<td>0.213</td>
<td>1.6</td>
<td>18.3 min</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>0.006</td>
<td>0.05</td>
<td>0.282</td>
<td>2.5</td>
<td>13 days</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>0.007</td>
<td>0.1</td>
<td>0.644</td>
<td>10</td>
<td>4 \times 10^{13}\ yrs</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td>0.010</td>
<td>1.00</td>
<td>9.966</td>
<td>100 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td>0.013</td>
<td>10</td>
<td>130</td>
<td>10 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td>0.017</td>
<td>0.10 ms</td>
<td>1.67 ms</td>
<td>16.7 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td>0.020</td>
<td>1 ms</td>
<td>19.93 ms</td>
<td>1.16 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td>0.023</td>
<td>0.01 sec</td>
<td>0.23 sec</td>
<td>115.7 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000,000</td>
<td></td>
<td>0.027</td>
<td>0.10 sec</td>
<td>2.66 sec</td>
<td>31.7 years</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td></td>
<td>0.030</td>
<td>1 sec</td>
<td>29.90 sec</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Big Oh examples

- $N^2 / 2 - 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- $\sin N = O(1)$; $10 = O(1)$, $10^{10} = O(1)$
- $\log N + N = O(N)$
- $N = O(2^N)$, but $2^N$ is not $O(N)$
Example

- Calculate $\sum_{i=1}^{N} i^3$
  
  ```
  int sum(int n)
  {
      int partialSum;
      partialSum = 0;
      for (int i=1;i<=n;i++)
          partialSum += i*i*i;
      return partialSum;
  }
  ```
  
  - Lines 1 and 4 count for one unit each
  - Line 3: executed $N$ times, each time four units
  - Line 2: (1 for initialization, $N+1$ for all the tests, $N$ for all the increments) total $2N + 2$
  - total cost: $6N + 4 \Rightarrow O(N)$
Sorting

• **Input**: sequence of n numbers
  \(<a_1, a_2, \ldots, a_n>\)

• **Output**: a permutation of the input sequence \(<b_1, b_2, \ldots, b_n>\) such that:
  \(b_1 \leq b_2 \leq \ldots \leq b_n\)
Insertion Sort

- Adding a new element to a sorted list will keep the list sorted if the element is inserted in the correct place.

- A single element list is sorted.

- Inserting a second element in the proper place keeps the list sorted.

- This is repeated until all the elements have been inserted into the sorted part of the list.
**Insertion Sort**

**INSERTION-SORT (A)**

1. for $j = 2$ to length[A]
2. \hspace{1em} key = A[j]
3. \hspace{1em} // Insert A[j] into the sorted sequence A[1...j-1]
4. \hspace{1em} i = j - 1
5. while $i > 0$ and A[i] > key
6. \hspace{1em} A[i+1] = A [i]
7. \hspace{1em} i = i - 1
8. A[i+1] = key

**Source:** “Introduction to Algorithms” 3rd Edition
Insertion Sort

Sorted already

Not yet processed
Algorithm Analysis

• In general, the time taken by an algorithm grows with the size of the input.
• So, it is traditional to describe the running time of a program as a function of the size of its input.
• The running time of an algorithm on a particular input is the number of primitive operations executed.
• We care about the worst-case scenario.
Important note before we start

When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
Analyzing Insertion Sort

**INSERTION-SORT (A)**

1. for $j = 2$ to length[A]
2. key = A[j]
4. $i = j - 1$
5. while $i > 0$ and A[i] > key
7. $i = i - 1$
8. $A[i+1] = key$

---

cost | times
---|---
$c_1$ | $n$
$c_2$ | $n-1$
$c_3 = 0$ | $n-1$
$c_4$ | $n-1$
$c_5 \sum_{j=2}^{n} t_j$ | 
$c_6 \sum_{j=2}^{n} (t_j - 1)$ | 
$c_7 \sum_{j=2}^{n} (t_j - 1)$ | 
$c_8$ | $n-1$

$t_j$ is the number of times the while loop test in step 5 is executed for that value of $j$.

**Source:** “Introduction to Algorithms” 3$^{rd}$ Edition
Analyzing Insertion Sort

Best case: 
A is sorted

$t_j = 1$ in step 5 for all $j$

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1).$$

Worst case: 
A is reverse sorted

$t_j = j$

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \left(\frac{n(n + 1)}{2} - 1\right)$$

$$+ c_6 \left(\frac{n(n - 1)}{2}\right) + c_7 \left(\frac{n(n - 1)}{2}\right) + c_8 (n - 1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n$$

$$- (c_2 + c_4 + c_5 + c_8).$$

$$T(n) = an^2 + bn + n$$

Source: “Introduction to Algorithms” 3rd Edition
How to Design An Algorithm

• Incremental approach: similar to insertion sort

• Divide-and-conquer approach:
  – **Divide**: break the problem into subproblems similar to the original problem but smaller in size
  – **Conquer**: solve the subproblems recursively
  – **Combine**: combine the solutions to create the solution of the original problem
Merge Sort

Sorts the elements of subarray A[p..r].
Initially: p = 1 and r = length[A]

```
MERGE-SORT(A, p, r)
1   if p < r
2       q = ⌊(p + r)/2⌋
3       MERGE-SORT(A, p, q)
4       MERGE-SORT(A, q + 1, r)
5       MERGE(A, p, q, r)
```
Merge Sort

\textbf{MERGE}(A, p, q, r)
1 \quad n_1 = q - p + 1
2 \quad n_2 = r - q
3 \quad \text{let } L[1..n_1 + 1] \text{ and } R[1..n_2 + 1] \text{ be new arrays}
4 \quad \textbf{for } i = 1 \textbf{ to } n_1
5 \quad \quad \quad L[i] = A[p + i - 1]
6 \quad \textbf{for } j = 1 \textbf{ to } n_2
7 \quad \quad \quad R[j] = A[q + j]
8 \quad L[n_1 + 1] = \infty
9 \quad R[n_2 + 1] = \infty
10 \quad i = 1
11 \quad j = 1
12 \quad \textbf{for } k = p \textbf{ to } r
13 \quad \quad \textbf{if } L[i] \leq R[j]
14 \quad \quad \quad A[k] = L[i]
15 \quad \quad \quad i = i + 1
16 \quad \quad \textbf{else } A[k] = R[j]
17 \quad \quad \quad j = j + 1

Source: “Introduction to Algorithms” 3rd Edition
Execution Example

• Partition

7 2 9 4 3 8 6 1

7 2 9 4

3 8 6 1

1 2 3 4 6 7 8 9
Execution Example (cont.)

• Recursive call, partition

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4
```

```
7 | 2 | 9 | 4
```

```
3 8 6 1
```

```
1 | 2 | 3 | 4 | 6 | 7 | 8 | 9
```

```
```
```
Execution Example (cont.)

- Recursive call, partition
Execution Example (cont.)

• Recursive call, base case
Execution Example (cont.)

- Recursive call, base case
Execution Example (cont.)

• Merge

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4
```

```
7 | 2 → 2 7
```

```
7 → 7  2 → 2
```

```
1 2 3 4 6 7 8 9
```
Execution Example (cont.)

- Recursive call, ..., base case, merge
Execution Example (cont.)

• Merge
Execution Example (cont.)

• Recursive call, ..., merge, merge

7 2 9 4 3 8 6 1

7 2 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8

7 2 9 4 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6
Execution Example (cont.)

- **Merge**

```
7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
```

```
7 2 | 9 4 → 2 4 7 9
```

```
3 8 6 1 → 1 3 6 8
```

```
7 | 2 → 2 7
```

```
9 4 → 4 9
```

```
3 8 → 3 8
```

```
6 1 → 1 6
```

```
7 → 7
```

```
2 → 2
```

```
9 → 9
```

```
4 → 4
```

```
3 → 3
```

```
8 → 8
```

```
6 → 6
```

```
1 → 1
```
Analyzing Merge Sort

\[ T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \]

- Calculate the middle of the array
- Recursively solve 2 subproblems each of size \( n/2 \)
- Combine the elements
Analyzing Merge Sort

- $T(n) = \text{divide work} + \text{conquer work} + \text{combine work}$
  
  $= D(n) + 2T(n/2) + C(n)$
  
  $= c + 2T(n/2) + cn$
Analyzing Merge Sort

- \( T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \)
  - \( = D(n) + 2T(n/2) + C(n) \)
  - \( = c + 2T(n/2) + cn \)

Source: "Introduction to Algorithms" 3rd Edition
Bubble Sort

• If we compare pairs of adjacent elements and none are out of order, the list is sorted

• If any are out of order, we must swap them to get an ordered list

• Bubble sort will make passes though the list swapping any adjacent elements that are out of order
Bubble Sort

• After the first pass, we know that the largest element must be in the correct place

• After the second pass, we know that the second largest element must be in the correct place

• Because of this, we can shorten each successive pass of the comparison loop
## Bubble Sort Example

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>7</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>3</td>
<td>8</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>6</td>
<td>13</td>
<td>9</td>
<td>15</td>
</tr>
</tbody>
</table>

1. Starting with the list of numbers:

   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
   | 1 | 3 | 5 | 7 | 4 | 2 | 8 | 6 | 10| 9 | 11| 12| 13| 14| 15| 16|

2. Bubble sort algorithm:

   - Compare elements and swap if out of order.
   - Repeat until no more swaps are needed.

3. After each pass:

   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
   | 1 | 3 | 5 | 7 | 4 | 2 | 8 | 6 | 10| 9 | 11| 12| 13| 14| 15| 16|

4. Final sorted list:

   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
   | 1 | 3 | 5 | 7 | 4 | 2 | 8 | 6 | 10| 9 | 11| 12| 13| 14| 15| 16|
**Bubble Sort Algorithm**

```plaintext
numberOfPairs = N
swappedElements = true
while (swappedElements) do
    numberOfPairs = numberOfPairs - 1
    swappedElements = false
    for i = 1 to numberOfPairs do
        if (A[i] > A[i + 1]) then
            Swap( A[i], A[i + 1] )
            swappedElements = true
        end if
    end for
end while
```
Best-Case Analysis

• If the elements start in sorted order, the for loop will compare the adjacent pairs but not make any changes.

• So the `swappedElements` variable will still be false and the while loop is only done once.

• There are \( N - 1 \) comparisons in the best case.
Worst-Case Analysis

• In the worst case the while loop must be done as many times as possible. This happens when the data set is in the reverse order.

• Each pass of the for loop must make at least one swap of the elements

• The number of comparisons will be:

\[ W(N) = \sum_{i=1}^{N-1} (N - i) = \sum_{k=N-1}^{1} k = \sum_{i=1}^{N-1} i = \frac{(N - 1) \times N}{2} = O(N^2) \]
Quicksort Algorithm

- Another divide-and-conquer algorithm
- Quicksort is usually $O(n \log n)$ but in the worst case slows to $O(n^2)$

Given an array of $n$ elements (e.g., integers):
- If array only contains one element, return
- Else
  - pick one element to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results
Quicksort

• **Divide step:**
  - Pick any element (**pivot** $v$) in $S$
  - Partition $S - \{v\}$ into two disjoint groups
    - $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
    - $S_2 = \{x \in S - \{v\} \mid x \geq v\}$

• **Conquer step:** recursively sort $S_1$ and $S_2$

• **Combine step:** the sorted $S_1$ (by the time returned from recursion), followed by $v$, followed by the sorted $S_2$ (i.e., nothing extra needs to be done)
Example

select pivot

partition
quicksort small

13 0 43 31
26 57

quicksort large

75 81 92
92

0 13 26 31 43 57 65 75 81 92
Pseudo-code

QUICKSORT(A, p, r)
1   if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q - 1)
4       QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1   x = A[r]
2   i = p - 1
3   for j = p to r - 1
4       if A[j] ≤ x
5           i = i + 1
6       exchange A[i] with A[j]
7   exchange A[i + 1] with A[r]
8   return i + 1
More Sorting Algorithms

• Shell sort
• Heap sort
• Radix sort
• Counting sort
• Bucket sort
• ...

Now that we have a sorted array, what is the most efficient way to search an element in it?
Binary Search

• Binary search. **Given** value and sorted array $a[]$, find index $i$ such that $a[i] = value$, or report that no such index exists.

• Ex. Binary search for 33.

![Binary Search Diagram](image)
Binary Search
Binary Search
Binary Search
Binary Search

\begin{center}
\begin{tabular}{ccccccccccccccc}
6 & 13 & 14 & 25 & 33 & 43 & 51 & 53 & 64 & 72 & 84 & 93 & 95 & 96 & 97 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
\uparrow & \uparrow \\
lo & hi \\
\end{tabular}
\end{center}
Binary Search

```
6 13 14 25 33 43 51 53 64 72 84 93 95 96 97
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

↑ ↑ ↑
lo mid hi
Binary Search
Binary Search
Binary Search

lo
hi
mid
Efficiency of binary search

- If \( n \) represents the number of names, the maximum number of searches \( x \) necessary to find a name is the smallest integer that satisfies the inequality \( 2^x > n \).

\[
\begin{align*}
2^x &> n \\
\log (2^x) &> \log n \\
x \log 2 &> \log n
\end{align*}
\]

The maximum number of searches is the smallest integer greater than \( \log n / \log 2 \).
### Efficiency of binary search

<table>
<thead>
<tr>
<th># of elements</th>
<th>Maximum sequential searches necessary</th>
<th>Maximum binary searches necessary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>5,000</td>
<td>5,000</td>
<td>13</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>50,000</td>
<td>50,000</td>
<td>16</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
</tbody>
</table>

With the incredible speed of today’s computers, a binary search becomes necessary only when the number of elements is large.
Don't you think that binary search is related to trees?
Tree Example:
Linux File Structure
Another Tree Example: Compiler Parse Tree

Parse tree for:
\[ x=1 \]
\[ y=2 \]
\[ 3\times(x+y) \]
So ... What is a tree?

• A tree is a **finite set of one or more nodes** such that:
  • There is a specially designated node called the **root**.
  • The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree.
• We call $T_1, \ldots, T_n$ the **subtrees** of the root.
Some Definitions

• The **degree of a node** is the number of subtrees of the node.
• The node with **degree 0 is a leaf or terminal node**.
• A node that has subtrees is the **parent** of the roots of the subtrees.
• The roots of these subtrees are the **children** of the node.
• Children of the same parent are **siblings**.
• The **ancestors** of a node are all the nodes along the path from the root to the node.
• The **level or depth** of a node $n$ is the length of the unique path from the root to $n$. 
A Tree Node

- Every tree node:
  - object - useful information
  - children - pointers to its children nodes
Left Child - Right Sibling
Example: Tree Implementation

```c
struct tnode {
    int key;
    struct tnode* lchild;
    struct tnode* sibling;
};
```

Example of operations:
- Create a tree with three nodes (one root & two children)
- Insert a new node (in tree with root R, as a new child at level L)
- Delete a node (in tree with root R, the first child at level L)
Binary Trees

• A special class of trees: max degree for each node is 2
• Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
Example: Is this a binary tree?
Example of Binary Trees

Skewed Binary Tree

Complete Binary Tree
Maximum Number of Nodes in BT

- The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$, $i \geq 1$ (assuming root is at level 1)
- The maximum number of nodes in a binary tree of depth $k$ is $2^{k-1}$, $k \geq 1$. 
Full BT vs. Complete BT

- A full binary tree of depth $k$ is a binary tree of depth $k$ having $2^k - 1$ nodes, $k \geq 0$ (root is at depth 1).
- A binary tree with $n$ nodes and depth $k$ is complete iff its nodes correspond to the nodes numbered from 1 to $n$ in the full binary tree of depth $k$. 

![Complete binary tree](image1)

![Full binary tree of depth 4](image2)
Binary Tree Representations: Array

- If a complete binary tree with \( n \) nodes is represented sequentially, then for any node with index \( i \), \( 1 \leq i \leq n \), we have:
  - parent(\( i \)) is at \( i/2 \) if \( i \neq 1 \). If \( i=1 \), \( i \) is at the root and has no parent.
  - leftChild(\( i \)) is at \( 2i \) if \( 2i \leq n \). If \( 2i > n \), then \( i \) has no left child.
  - rightChild(\( i \)) is at \( 2i+1 \) if \( 2i + 1 \leq n \). If \( 2i + 1 > n \), then \( i \) has no right child.
Array presentation (aka Sequential presentation)

(1) waste space
(2) insertion/deletion problem
typedef struct tnode *ptnode;
typedef struct tnode {
    int data;
    ptnode left, right;
};
Binary Tree Traversals

- There are six possible combinations of traversal
  - lRr, lrR, Rlr, Rrl, rRl, rlr
- Adopt convention that we traverse **left before right**, only 3 traversals remain:
  - lRr, lrR, Rlr
  - **inorder, postorder, preorder**
Example: Arithmetic Expression Using BT

inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
Inorder Traversal (recursive version)

```c
void inorder(ptnode ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left);
        printf("%d", ptr->data);
        inorder(ptr->right);
    }
}
```

A / B * C * D + E
void preorder(ptnode ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left);
        preorder(ptr->right);
    }
}

+ * * / A B C D E
void postorder(ptnode ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left);
        postorder(ptr->right);
        printf("%d", ptr->data);
    }
}
Euler Tour Traversal

- generic traversal of a binary tree
- the preorder, inorder, and postorder traversals are special cases of the Euler tour traversal
- “walk around” the tree
Good Book
Good Book
Good Book: For Fun!
Conclusions

• We defined what an algorithm is in simple terms.
• Big Oh notation is a convenient way to compare algorithms.
• Sometimes the best solution may not be needed and a good-enough solution is just fine.
• Heuristics are the way to go if we cannot get the exact/best results with reasonable resources.
• You already know stack and queues ... Now you know trees!