Introduction

Analysis of Algorithms
What is Computer Science?

• Computer science is the “science of computation”, i.e., the study of computational problem solving

• The essence of computational problem solving is algorithm design and analysis

• An **algorithm** is a method for solving a problem

• A **data structure** is a means of storing the information that will be used by the algorithm
“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

— Linus Torvalds (creator of Linux)
Programming and Computer Science
Developing an Algorithm

1. Model the problem
2. Find or create an algorithm to solve it
3. Determine if the algorithm meets performance specifications
   • Is the algorithm correct?
   • Is it fast enough?
   • Does it use too much memory?
4. If performance is not met, figure out why
5. Address the performance issues
Maximum Subarray Problem

• Problem: Given an array of $N$ integers, find the subarray of contiguous elements that have the largest sum
  – For example, if array $A$ is [0, -5, -1, 7, -3, 6, 9, -2, -4, 3, -3, 2] the maximum subarray is found at locations $A[3:6]$ and the sum is 19
  – Assume the first element is always 0, so that if all the other array elements are negative, the maximum subarray is just the first element with a sum of 0
Maximum Subarray Problem

- Solution 1: Compute the sum of every possible subarray; for each subarray sum, compare that to a running maximum
  - For example, one possible subarray of
    
    \([0, -5, -1, 7, -3, 6, 9, -2, -4, 3, -3, 2]\) is \([-5, -1, 7, -3]\), with a sum_{1,4} = -2
Maximum Subarray Problem

- Possible code for solution 1:

```java
int sum = 0;
int max = 0;

for (int j=1; j<n; j++) {
    for (int k=j; k<n; k++) {
        sum = 0;
        for (int i=j; i<=k; i++) {
            sum += A[i];
        }
        if (sum > max) max = sum;
    }
}
```
Maximum Subarray Problem

• Efficiency of solution 1:
  – How fast is solution 1?
    • Triply-nested loop: the outer loop iterates \( N \) times, the middle loop iterates at most \( N \) times, and the innermost loop iterates at most \( N \) times, for a total runtime proportional to \( N^3 \)
Maximum Subarray Problem

• Solution 2: Compute a prefix sum \( P \) for each element in the array, then compute the subarray sum using the formula \( \text{sum}_{j,k} = P_k - P_{j-1} \)
  
  – The prefix sum is the accumulated sum from left to right
  – For example, the prefix sum of \([0, -5, -1, 7, -3, 6, 9, -2, -4, 3, -3, 2]\) is
  
  \[
P = [0, -5, -6, 1, -2, 4, 13, 11, 7, 10, 7, 9]
  \]

  – After computing the prefix sum, compute the maximum subarray sum using the above formula
  – For example: \( \text{sum}_{3,6} = P_6 - P_2 = 13 - (-6) = 19 \)
Maximum Subarray Problem

• Possible code for solution 2:

```java
int max = 0;
int sum = 0;

int[] P = new int[n];
P[0] = 0;

for (int i=1; i<n; i++) {
    P[i] = P[i-1] + A[i];
}

for (int j=1; j<n; j++) {
    for (int k=j; k<n; k++) {
        sum = P[k] - P[j-1];
        if (sum > max) max = sum;
    }
}
```
Maximum Subarray Problem

• Efficiency of solution 2:
  – How fast is solution 2?
    • Single loop to compute the prefix sum and doubly-nested loop to compute the summation for a total runtime proportional to $N + N^2$
Maximum Subarray Problem

• Solution 3: Instead of computing a prefix sum, compute a *maximum suffix sum* using the formula $M_t = \max \{0, M_{t-1} + A[t]\}$
  - The maximum suffix sum of $[0, -5, -1, 7, -3, 6, 9, -2, -4, 3, -3, 2]$ is $M = [0, 0, 0, 7, 4, 10, 19, 17, 13, 16, 13, 15]$
  - Then the maximum subarray value is found at $M_6 = 19$
Maximum Subarray Problem

- Possible code for solution 3:

```java
int max = 0;
int[] M = new int[n];
M[0] = 0;

for (int t=1; t<n; t++) {
    M[t] = Math.max(0, M[t-1] + A[t]);
}

for (int t=1; t<n; t++) {
    if (M[t] > max) max = M[t];
}
```
Maximum Subarray Problem

• Efficiency of solution 3:
  – How fast is solution 3?
    • Single loop to construct $M$ and another single loop to find the maximum value in $M$, for a total runtime proportional to $N + N = 2N$
Arbitrage Detection Problem

- Problem: Given a table of exchange rates, is there an opportunity for arbitrage?
  - For example, $1,000 USD ⇒ 741 Euros ⇒ $1,012.206 Canadian ⇒ $1,007.14497 USD
  - $1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>GBP</th>
<th>CHF</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1</td>
<td>0.741</td>
<td>0.657</td>
<td>1.061</td>
<td>1.011</td>
</tr>
<tr>
<td>EUR</td>
<td>1.350</td>
<td>1</td>
<td>0.888</td>
<td>1.433</td>
<td>1.366</td>
</tr>
<tr>
<td>GBP</td>
<td>1.521</td>
<td>1.126</td>
<td>1</td>
<td>1.614</td>
<td>1.538</td>
</tr>
<tr>
<td>CHF</td>
<td>0.943</td>
<td>0.698</td>
<td>0.620</td>
<td>1</td>
<td>0.953</td>
</tr>
<tr>
<td>CAD</td>
<td>0.995</td>
<td>0.732</td>
<td>0.650</td>
<td>1.049</td>
<td>1</td>
</tr>
</tbody>
</table>
Arbitrage Detection Problem

- Solution: Create a currency exchange graph
  - Vertex = currency
  - Edge weight = exchange rate
  - Find a directed cycle whose product of edge weights is > 1

\[
0.741 \times 1.366 \times 0.995 = 1.00714497
\]
Arbitrage Detection Problem

• Model the problem as *negative cycle detection* using a *shortest path algorithm*
  – Replace each edge weight with the *negative log* of the exchange rate
  – This changes multiplication into addition (> 1 turns to < 0)
  – Find a directed cycle whose sum of edge weights is < 0 (a negative cycle) using a shortest path algorithm

\[
\begin{align*}
0.741 \times 1.366 \times 0.995 &= 1.00714497 \\
-ln(0.741) + -ln(1.366) + -ln(0.995) &= 0.2998 - 0.3119 + 0.0050 = -.0071
\end{align*}
\]