Algorithms Analysis

Weiss, sections 5.1 through 5.3
(pages 187–200)
Algorithms

• Algorithm is a set of instructions that the computers follow to solve a problem.
• Algorithm analysis determines the amount of resources (time and memory space) that the algorithms needs to execute.
  – Temporal and spacial complexities
• Build up a mathematical model that captures resource usage.
Why not use clock?

- Computers are very fast to perform computations. For small inputs, algorithms have identical performance.
- To find trends, algorithms need to be tested with many inputs, which can be time-consuming.
- The actual runtime of a program is platform-dependent.

- A mathematical model is an *approximation* of true runtime.
- It is usually preferable to work with mathematical models.
Relevant Operations

• Count the *number* of operations that contribute to the runtime of a program.

```java
public static boolean contains(int[] arr, int k) {
    for(int i = 0; i < arr.length; i++) {
        if(arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

• Operations: initialization, returning values, comparison
  
  ➢ Comparison (arr[i] == k) is a relevant operation. The number of comparison depends on the input size (array length).
Relevant Operations

- There are multiple relevant operations: assignment to `sum` and array access.
- Choose assignment to be the most relevant operation because there are less of them to count.

```java
public static int addAdjacentPairs(int[] arr, int k) {
    int sum = 0;
    for(int i = 0; i < arr.length - 1; i++) {
        sum = sum + arr[i] + arr[i+1];
    }
}
```
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

• Array access is the relevant operation.
• swap has 4 operations (two reads and two writes)

    int[] arr = new int[10];
    /* initialize arr */
    swap(arr, 0, 1);

• The number of array accesses (4) is not dependent on the array size.
Counting Operations

```java
public static sum(int[] arr) {
    int sum = 0;
    for(int i = 0; i < arr.length; i++) {
        sum = sum + arr[i];
    }
}
```

- **Array access** is the relevant operation.
- The number of array accesses: The number of times the for-loop iterates times the number of operations per iteration.
- One operation per iteration. `arr.length` operations
- Express the number of operations as a **mathematical function**: \( T(n) = n \)  
  \( n \) is the size of input array

- For the **swap** function, \( T(n) = 4 \)
Counting Operations

A formula for the number of operations any loop performs:

Total operations = Number of iterations times number of operations per loop

int sum = 0;
for(int i = 0; i < arr.length; i++) {
    for(int j = 0; j < arr.length; j++) {
        sum = sum + i + j;
    }
}

- The relevant operation is addition. Assuming that $n$ is the input size:
  - An iteration of inner loop has 3 additions. \(3n\) additions in total
  - An iteration of outer loop has \(3n + 1\) additions. \((3n + 1) \cdot n = 3n^2 + n\) additions in total
Counting Operations

public static boolean contains(int[] arr, int k) {
    for(int i = 0; i < arr.length; i++) {
        if(arr[i] == k) {
            return true;
        }
    }
    return false;
}

• The relevant operation is array access.
• Total number of operations is dependent on array length and the location of \( k \) in the array.
• Define three cases
  • **Best case**: \( k \) is in the front of array. \( T(n) = 1 \)
  • **Worse case**: \( k \) is the last element or is not in the array. \( T(n) = n \)
  • **Average case**: \( k \) is in the middle of array. \( T(n) = n/2 \)
Growth of Functions

- Programs show identical performance for small inputs.
- How a program running time increases when scaling the input size.
- This study is called the **asymptotic analysis** of algorithms. We use Big-O notation for doing this study.

- Categorize mathematical functions as the input size grows:
  - Constant functions. \( f(x) = 4, \text{ swap} \)
  - Linear functions (lines) \( f(x) = mx + b, \text{ sum (slide 7)} \)
  - Quadratic functions. \( f(x) = ax^2 + bx + c, \text{ doubly-nested loop} \)
  - Cubic functions. \( f(x) = ax^3 + bx^2 + cx + d, \text{ triply-nested loop} \)
  - Exponential functions. \( f(x) = ab^x, \text{ all possible subsets of a set with n elements is } 2^n. \)
  - Logarithmic functions. \( f(x) = \log x, \text{ problems that are successively cut in half until it is solved.} \)

Computer scientists use 2 as the base of logarithms. Most problems are divided into two parts.
Big-O Notation

• Classify the growth behavior of a program using one of the classes of mathematical functions.
• Use **Big-O notation** for doing this classification.
• \( O(f) \): A set of functions that have the same growth behavior as \( f \).
• \( O(f), f(n) = n \): A set of functions with the **linear** growth behavior. \( f_1(n) = 3n, f_2(n) = 1-n \)
• \( g \in O(f) \): \( g \) belongs to a set of functions that \( f \) belongs to. \( f_1 \in O(f), f_2 \in O(f) \)

• Express the complexity of some functions seen in the previous slides:
  - swap: \( T(n) = 4 \rightarrow T \in O(1) \)
  - sum: \( T(n) = n \rightarrow T \in O(n) \)
  - Doubly-nested loop: \( T(n) = 3n^2 + n \rightarrow T \in O(n^2) \)

- Big-O notation provides an **upper bound** on the growth behavior of programs.
Recurrence Relation

• Express complexity for recursive programs.
• Use a type of mathematical function known as \textit{recurrence} relation.

Consider the factorial function

\begin{verbatim}
public static long factorial(int n) {
    if(n == 0) {
        return 1;
    } else {
        return factorial(n-1) * n;
    }
}
\end{verbatim}

• The relevant operation is \textit{multiplication}.
Recurrence Relation

• The recursive defined is dependent on \( n \). There are two cases

• Base case: \( n = 0 \rightarrow T(0) \)
• Recursive case: \( n \neq 0 \rightarrow T(n) \)

\[
T(0) = 0
\]
\[
T(n) = 1 + T(n - 1)
\]

• Given the recurrence, define the complexity of factorial using Big-O
• Use the substitutive method to derive an explicit formulation

\[
T(n) = 1 + T(n - 1) = 1 + (1 + T(n - 2)) = 1 + (1 + (1 + T(n - 3)))
\]

• Rewrite \( T(n) \) in terms of the number of substitutions (\( k \)):

\[
T(n) = k + T(n - k)
\]
Recurrence Relation

• Total number of operations is obtained when reaching the base case.
• When the base case runs, there are $n$ substitutions ($k = n$):

$$T(n) = k + T(n - k) = n + T(n - n) = n + T(0) = n + 0 = n$$

$T \in O(n)$
Binary Search

- Finds a target in a sorted array.
- Compares the target with the middle element.
- Eliminates lower or upper half of the array depending on the outcome of comparison.

```
3 6 7 12 14 19 22 42 45

0 1 2 3 4 5 6 7 8

low mid high

3 6 7 12 14 19 22 42 45

target: 15
```
### Binary Search

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>14</td>
<td>19</td>
<td>22</td>
<td>42</td>
<td>45</td>
</tr>
</tbody>
</table>

#### Target: 15

- **0** low
- **1** mid
- **2** high

#### Target: 15

- **0** low
- **4** mid
- **5** high

#### Target: 15

- **0** high
- **8** low
Binary Search

```c
int binarysearch(int nums[], int size, int value) {
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (nums[mid] < value) {
            low = mid + 1;
        } else if (nums[mid] > value) {
            high = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

Comparison is the relevant operation. The number comparisons is $1.5 \log n$. Asymptotically, binary search takes $O(\log n)$. 