Lecture 4: Best, Worst, and Average Case Complexity
CS204/209: Algorithms (and Scientific Computing)

Wednesday, 30 Sept 2009
In today’s lecture

1. Recall… Computational Complexity
   - Time Complexity

2. Example: Binary Search
   - Best case
   - Worst case
   - Average case

3. Exercise: Linear Search

4. Example: Bubble sort

Most of the content is based on Section 2.3 of Rosen’s *Discrete Mathematics*, and 2.2 of Cormen et al. *Introduction to Algorithms*. 
Recall... Problem 2

Here is a “mutilated chess board”:

Suppose you have 31 dominoes, each the size of two of the squares. Can you cover the remaining 62 squares with the 31 dominoes? If not, can you prove its impossible?
Problem 3

There are three boxes:

- One labelled **BB** that contains 2 black marbles.
- One labelled **WW** that contains 2 white marbles.
- and one labelled **WB** that contains 1 white and 1 black marbles.

Someone has mixed up the labels so that no box has its correct label.

*Devise* an algorithm to find working out contents of the boxes by removing **one marble at a time**.

What is the fewest number of steps needed by your algorithm?
Recall... Computational Complexity

Our goal is to be able to compare algorithms and determine which is the most “efficient”. When we calculate the efficiency of an algorithm, we are essentially trying to quantify how computer resources are required. And the two most important computer resources are time and memory.

Usually this is referred to as the computational complexity of the algorithm, and

- **Time complexity** is an analysis of the amount of time required to solve a problem of a particular size, while the
- **Space complexity** is an analysis of the amount of memory required.

For the most part, we take computational complexity to mean **Time complexity**.
Recall... Computational Complexity  Time Complexity

Time complexity is usually calculated in terms of the number of times certain operations are carried out by an algorithm, e.g.,

- the number of additions or subtractions,
- the number of multiplications,
- the number of **comparisons** made (**if** statements).

Typically, we count the number of times a certain operation, e.g, a comparison, and give a Big-$\mathcal{O}$ estimate.

There are three possible cases of interest:

- **Best-case**: by sheer, dumb luck, what is the easiest problem we might have to solve?
- **Worst-case**: by sheer, bad luck, what is the worst problem we might have to solve?
- **Average-case**: what is the mean complexity over all possible sets of inputs.
Recall... Computational Complexity  Time Complexity

Of these, “best-case” is the easiest and “average-case” is the hardest. “Worst-case” is the most important (why?) and turns out to be \textit{asymptotically similar} to average-case.

We’ll need the following identities, the first of which we proved last week:

\[
\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \quad \sum_{k=1}^{n} 2^{k-1} = 2^n - 1.
\]

**Exercise**

Prove that

\[
\sum_{i=1}^{k} i2^{i-1} = 2^k(k-1)+1.
\]
Example: Binary Search

Recall the **Binary Search** algorithm?

---

**The Binary Search Algorithm**

Find the index of the element $x$ in the **ordered** list
\[ \{a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n\}. \]

1. Split the list into two sublists $a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}$ and $a_{\lfloor n/2 \rfloor} + 1, \ldots, a_n$.

2. If $x < a_{\lfloor n/2 \rfloor}$, then $x$ is in the first sublist. Otherwise it is in $a_{\lfloor n/2 \rfloor} + 1, \ldots, a_n$.

3. Split the new list into two lists, and proceed as above.

4. Stop when you find that $x$ is in a list of length 1.

---

Give the best, worst and average case complexities for this algorithm.
Example: Binary Search

**Best case**

This is easy!
A little harder. We’ll assume that the list is of length $n = 2^k$, and count the number of comparisons required.
Example: Binary Search

This will take is some more thought. We’ll assume that the list is of length \( n = 2^k - 1 \).

We will need to use that

\[
\sum_{i=1}^{k} i2^{i-1} = 2^k(k - 1) + 1.
\]

(Note: this is different from the version given in class:
\[
\sum_{i=1}^{k} i2^{i-1} = 2^k(k - 1) - 1,
\]
which is incorrect).
Exercise: Linear Search

Exercise

1. Describe the Linear Search Algorithm, e.g., by giving a pseudocode description of it,

2. Show how it would try to locate the letter \( R \) in the string \{ 'D', 'E', 'S', 'C', 'R', 'I', 'B', 'E' \}.

3. Provide a best, worst, and average-case complexity analysis for the algorithm, in terms of the number of comparisons required when applied to a list of length \( n \).
Example: Bubble sort

- go through the list \(\{a_1, a_2, \ldots, a_{n-1}, a_n\}\).
- If an adjacent pair of element are out of order, swap them.
- At the end of the 1st pass, the largest element should be at the end.
- Now repeat the process for the list \(\{a_1, a_2, \ldots, a_{n-1}\}\).
- and then for the list \(\{a_1, a_2, \ldots, a_{n-2}\}\).
- Continue until you have sorted the list \(\{a_1, a_2\}\).

Question: What are the best- and worst-case time complexities — in terms of the number of comparisons — for this algorithm?

Solution:
Example: Bubble sort

Exercise

Recall Insertion Sort from Lecture 2. Show that the total number of comparisons required by it for a list of length $n$ is $O(n^2)$. 
Example: Bubble sort

Exercise (Based on Cormen at al, Exer 2.2-2)

The selection sort algorithm works as follows:

- Find the smallest element in the list \( \{a_1, a_2, a_3, \ldots, a_n\} \), and swap it with the element \( a_1 \).
- Find the smallest element in the list, \( \{a_2, a_3, \ldots, a_n\} \) and swap it with the element \( a_2 \).
- ...
- Stop when you have sorted item \( a_{n-1} \).

1. Write a pseudocode description of this algorithm.
2. What loop invariant does this algorithm maintain?
3. Give best- and worst-case estimates for the number of comparisons required.