Alg 2B: Introduction to Algorithms
Lecture 1 of ADS thread
Kyriakos Kalorkoti
School of Informatics
University of Edinburgh

Algorithms and Data Structures thread
Taught by Kyriakos Kalorkoti (KK), IF5.26, kk@inf.ed.ac.uk.

Topics:
1. Algorithms, analysing algorithms, Asymptotic notation (for talking about running-times), Sequential Data Structures, Tree data structures, Hashing, Priority Queues, Advanced sorting.
2. Algorithms for searching graphs, applications to graph problems.

Textbooks
For Algorithms and Data Structures (recommended, not required.):
- [GT] Data Structures and Algorithms in Java, by Goodrich & Tamassia (4th or 3rd ed), Wiley. Gentle textbook, best for this course (doesn’t have WWW stuff).
- Java.
- No Java (or any other programming language).
- Course text for 3rd year Algorithms and Data Structures course.

If you will not take 3rd year ADS, choose [GT], but don’t rush out to buy a book straight away.

Study advice
1. Education is done with you not to you.
2. You are here because you want to learn the subject.
3. Course consists of:
   - Lectures.
   - Tutorials.
   - Practical work (2 assignments only, 1 for each thread).
   - Private study.

Deciding not to take an active part in all of these is deciding to under perform at best and fail at worst. It is not possible to coast along and revise just before the exams (unless failure seems like a good idea).

My promise: If you ask for help I will do my utmost to provide it. But please use the channels above first when appropriate.

Questions from you: Strongly encouraged, during lectures, after lectures or email.

Finally:
- Lectures start at 4.10, keep any eye on the clock and wind down any conversation.
- In lectures either I talk or you talk but not both!
- Laptops, tablets, phones should be put away (unless a medical condition requires the use of an aid).
- If you have any special needs that need my cooperation please speak to me.

Our Ingredients
- Algorithms Step-by-step procedure (a “recipe”) for performing a task.
- Data Structures Systematic way of organising data and making it accessible in certain ways.
- We are interested in the design and analysis of “good” algorithms and data structures.
- Think about very large systems and the need to have them work within acceptable time.
What you have probably seen already

Data Structures
- Arrays, linked lists, stacks, trees.

Algorithm design principles
- Recursive algorithms.

Searching and Sorting Algorithms
- Linear search and Binary search. Insertion sort, selection sort.

Other prerequisites:
- The ability to reason mathematically, spot a bad argument from a mile off.
- Write down a mathematical argument fluently. It should be a pleasure to read.
- See Note 1 for advice on setting out mathematical reasoning.

Evaluating algorithms

- Correctness
- Efficiency w.r.t.
  - running time,
  - space (=amount of memory used),
  - network traffic,
  - number of times secondary storage is accessed.
- Simplicity

Measuring Running time

The running time of a program depends on a number of factors such as:
1. The input.
2. The running time of the algorithm.
3. The quality of the implementation and the quality of the code generated by the compiler.
4. The machine used to execute the program.

We will rarely be concerned with the implementation quality, the code quality or the machine.

A given algorithm can be implemented by many different programs (indeed languages).

Example 1: Linear Search in JAVA

```java
public static int linSearch(int[] A, int k) {
    for (int i = 0; i < A.length; i++)
        if (A[i] == k)
            return i;
    return -1;
}
```

This is Java.
- We want to ignore implementation details, so we map this to pseudocode.

In reality things are the other way round!

Linear Search in Pseudocode

**Input:** Integer array A, integer k being searched.
**Output:** The least index i such that A[i] = k; otherwise -1.

**Algorithm** linSearch(A, k)

1. for i = 0 to A.length - 1 do
2.     if A[i] = k then
3.         return i
4.     return -1

Suppose A = (19, 5, 6, 77, 2, 1, 90, 3, 4, 22, 1, 5, 6) and k = 1.
What happens?

Worst Case Running Time

Assign a size to each possible input.

**Definition**
The (worst-case) running time of an algorithm A is the function
T_A : N → N where T_A(n) is the maximum number of computation steps performed by A on an input of size n.

**Example:** linSearch.
- Suppose the size is the length of the array A.
- Worst-case running time is a linear function of size.

**Note:**
- Implicit assumption that array entries are of bounded size.
- Otherwise we could take sum of all array entry sizes as measure of input size (plus size of k).
Average Running Time

In general worst-case seems overly pessimistic.

Definition
The average running time of an algorithm A is the function \( \text{AVT}_A : \mathbb{N} \rightarrow \mathbb{R} \) where \( \text{AVT}_A(n) \) is the average number of computation steps performed by A on an input of size n.

Problems with average time
- What precisely does average mean? What is meant by an “average” input depends on the application.
- Average time analysis is mathematically very difficult and often infeasible (OK for linSearch).

Example 2: Binary Search
Input: Integer array A in increasing order, integers \( i_1, i_2, k \).
Output: An index \( i \) with \( i_1 \leq i \leq i_2 \) and \( A[i] = k \), if such an \( i \) exists, −1 otherwise.

Algorithm binarySearch(\( A, k, i_1, i_2 \))
1. if \( i_2 < i_1 \) then return −1
2. else
3. \( j \leftarrow \lfloor \frac{i_1 + i_2}{2} \rfloor \)
4. if \( k = A[j] \) then return \( j \)
5. else if \( k < A[j] \) then return \( \text{binarySearch}(A, k, i_1, j - 1) \)
6. else return \( \text{binarySearch}(A, k, j + 1, i_2) \)

Running-time of Binary search
Input array with \( n = i_2 - i_1 + 1 \) (the number of items in the region we search).
- Do at most a constant \( c \) amount of work.
- If \( k \) found done else recurse on array of size about \( n/2 \).
- Do a constant \( c \) amount of work.
- If \( k \) found done else recurse on array of size about \( n/2^2 \).
  :  
- Do a constant \( c \) amount of work.
- If \( k \) found done else recurse on array of size about \( n/2^2 \).
Base case: \( n/2^r = 1 \), i.e., \( r = \log(n) \). Then one more call. Total work done (time) no more than
\[
 c \left( \log(n) + 2 \right).
\]
Better than linSearch?

\[
\begin{align*}
T_{\text{linSearch}}(n) &= 10n + 10, \\
T_{\text{binarySearch}}(n) &= 1000\log(n) + 1000.
\end{align*}
\]

\[
\begin{align*}
&10000 \quad 9000 \quad 8000 \quad 7000 \quad 6000 \quad 5000 \quad 4000 \quad 3000 \quad 2000 \quad 1000 \quad 0  \\
\hline \hline
1000 & 2000 & 3000 & 4000 & 5000 & 6000 & 7000 & 8000 & 9000 & 10000
\end{align*}
\]

\[
\begin{align*}
\text{lg } n \text{ versus } n
\end{align*}
\]

Put \( m = \text{lg } n \).

By definition \( n = 2^m \).

Now:
\[
\begin{align*}
m &\rightarrow m + 1 \quad n \rightarrow 2n \\
m &\rightarrow m + 5 \quad n \rightarrow 32n \\
m &\rightarrow m + 10 \quad n \rightarrow 1024n \\
m &\rightarrow m + c \quad n \rightarrow 2^c n
\end{align*}
\]
Some Statistics

Jan 2008 on a DICE machine.

<table>
<thead>
<tr>
<th>size</th>
<th>wc linS</th>
<th>avc linS</th>
<th>wc binS</th>
<th>avc binS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>100</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>1000</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>10000</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>20000</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>40000</td>
<td>3 ms</td>
<td>1.3 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>60000</td>
<td>3 ms</td>
<td>1.5 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>80000</td>
<td>3 ms</td>
<td>1.5 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>100000</td>
<td>5 ms</td>
<td>2.1 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>200000</td>
<td>7 ms</td>
<td>3.7 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>400000</td>
<td>12 ms</td>
<td>6.9 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>600000</td>
<td>24 ms</td>
<td>11.6 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
<tr>
<td>800000</td>
<td>24 ms</td>
<td>15.6 ms</td>
<td>≤ 1 ms</td>
<td>≤ 1 ms</td>
</tr>
</tbody>
</table>

200 repetitions for each size.

Why not just do experiments?

- Consider sorting arrays of the integers 1, 2, …, 100 held in some order.
- Just take a 1% sample of all possible inputs.
- How many experiments?
- Assume algorithm can sort $10^{10}$ instances per second(!).

How long do we need to wait?

$$99! \approx 9.33262154439441526816923885626670049671596826438$$

$$10^{99} \approx 2.951269209 \times 10^{36}$$ years.

Be seeing you!