Basic Algorithms Lec-1

Abhinav Tamaskar
29 May, 2019

CIMS, NYU
Overview of the course
• Written Assignments
  • Algorithm design
  • Rigorous proofs
  • Complexity Analysis

• Programming assignments
  • Efficient implementations
  • Optimization for speed and memory

• Quizzes
  • 2-3 small problems (open notes)

• Exams
  • Based on the written assignments
  • Open notes
Aim of the course:

- Learn to analyze a problem in the real world and get the optimal solution to the problem.
- New techniques to design and analyze the efficiency of the algorithms.
- Develop the mathematics needed to help prove the correctness of an algorithm.

Learn by solving (think of algorithmic puzzles).
Problem
Given an array of integers $a_1, a_2, \ldots, a_n$, find a peak\(^1\) in the array.

$$
1 \quad 4 \quad 10 \quad 8 \quad 11 \quad 9 \quad 10 \\
\uparrow \quad \uparrow \quad \uparrow
$$

Solution
Naive algorithm: Check each position if it is a peak or not. Number of comparisons done the algorithm: approximately $2 \times n$ comparisons in the worst case...

Can we do faster?

\(^1\)A peak is an element $a_i$ such that $a_i \geq a_{i-1}, a_{i+1}$ (or only one side if, $i = 1$ or $n$)
Use binary search!

Solution

- Choose the middle point of the array and check if it a peak.
- If it is not a peak, then find the larger neighbour and recursively solve that side of the array.
How many comparisons are done?

- For each midpoint chosen, we do 2 comparisons.
- At each step we reduce the size of the array by half.

Let, $T(n)$ denote the number of comparisons done when the array is of size $n$.

When we have $\leq 2$ elements we need to do only 1 comparison, which means $T(2) = 1$.

To get the general formula for $T(n)$ we can calculate the total number of comparisons based on a recurrence relation\(^2\):

$$T(n) = T(n/2) + 2$$

\(^2\)recurrence relation
Solving the recurrence relation by induction:

\[ T(n) = T(n/2) + 2 \]
\[ = T(n/4) + 4 \]
\[ \vdots \]
\[ = T(n/2^k) + 2 \cdot k \quad \text{induction hypothesis} \]
\[ \vdots \]
\[ = T(n/2^{\log n}) + 2 \cdot \log n = 1 + 2 \cdot \log n \]
\[ = O(\log n) \]
Recap of Big-O notation and asymptotic analysis
Definition
Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that $f = O(g)$ if

$$\exists c \in \mathbb{R}_{>0} \text{ and } n_0 \in \mathbb{N} \exists \forall n > n_0, f(n) \leq c \cdot g(n)$$

E.g.

<table>
<thead>
<tr>
<th></th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$1/2 \cdot n$</td>
<td>$n^2/1000$</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$n$</td>
<td></td>
</tr>
</tbody>
</table>
E.g. non-comparable functions

<table>
<thead>
<tr>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
</table>
| $n$ | \[
\begin{cases}
1, & n \text{ is odd} \\
n^2, & n \text{ is even}
\end{cases}
\] |
| $\sin n$ | $\cos n$ |
Definition
Given two functions, as above, we say that $f = \theta(g)$ if $f = O(g)$ and $g = O(f)$

E.g.

<table>
<thead>
<tr>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2$</td>
<td>$n^2/2$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\begin{cases} n + n/2, &amp; n \text{ is odd} \ n - n/2, &amp; n \text{ is even} \end{cases}$</td>
</tr>
</tbody>
</table>
Definition
Given two functions, as above, we say that $f = o(g)$ if

$$\forall \epsilon \in \mathbb{R}_{>0} \exists n_{\epsilon} \in \mathbb{N} \exists \forall n > n_{\epsilon}, f(n) \leq \epsilon \cdot g(n)$$

E.g.$^3$

<table>
<thead>
<tr>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n^2/2$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\begin{cases} e^n - n^2, &amp; n \text{ is odd} \ e^n + n^2, &amp; n \text{ is even} \end{cases}$</td>
</tr>
</tbody>
</table>

$^3o$ is used a lot in cryptography for bounding errors in hashing and other randomized algorithms
Time complexity for real world problems
Number of operations per second in a processor $10^9$ (per core). Nowadays it is $10^{10} - 10^{11}$ (and will probably be more due to Moore’s law but don’t count on it)\textsuperscript{4}.

\textsuperscript{4}instructions per second
Most of the programming assignments will have time limits within 2s - 4s. This would mean depending on the size of the input you can determine the time complexity of the solution which would give full marks.
<table>
<thead>
<tr>
<th>Size of input</th>
<th>Max time complexity feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$2^n$, $3^n$, $4^n$</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$n^2 \sqrt{n}$, $n^3$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$n^2$, $n^2 \sqrt{n}$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$n \log n$, $n \sqrt{n}$</td>
</tr>
<tr>
<td>$\geq 10^6$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
More recursive problems
Problem
Given an integer-matrix, \( M \), of size \( m \times n \), find a peak\(^5\) in this matrix.

E.g.

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
100 & 105 & 1 & 2 & 50 \\
19 & 20 & 19 & 14 & 12 \\
3 & 1 & 2 & 2 & 11 \\
10 & 200 & 3 & 5 & 10 \\
\end{array}
\]

\(^5\) a peak is a position which is larger than all neighbours who share an edge
Naive algorithm takes $O(mn)$, by checking all positions. Faster solution, use binary searching for optimal column.

Solution

- Choose the middle column, $n/2$, and find the maximum of the column (say at position $i$).
- Check which side of $(i, n/2)$ is larger and recursively solve the smaller matrix, to the side of this column.
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>105</td>
<td>1</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>20</td>
<td><strong>19</strong></td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>200</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>100</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>20</td>
<td>19</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>200</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>100</td>
<td>105</td>
<td>1</td>
<td>2</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>19</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>200</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Exercise: Will the above technique work if we replace finding the maximum by finding a column peak? Give a proof or a counterexample.
Counterexample:

\[
\begin{array}{ccc}
11 & 30 & 40 \\
10 & 1 & 11 \\
9 & 2 & 8 \\
\end{array}
\]

If we chose 2 as our column-peak, our algorithm fails, as there is no matrix-peak in the first column.
Proof of correctness, by induction on number of columns:

- **Base case:** 1 column $\rightarrow$ the maximum is a peak.
- **Supposed** that our algorithm is correct for $\#\text{columns} < n$.
- Each time a submatrix is chosen, there is a number in it that is greater than the maximum in the previous column $\rightarrow$ it is greater than all elements in that column. If we follow an increasing path from this number, along its neighbours, we are guaranteed to say inside this submatrix, proving that this submatrix has a peak.
- This submatrix has a smaller number of columns ($n/2$) and our induction hypothesis states that our algorithm finds this peak, hence finding the peak for a matrix with $n$ columns.
Analysis of time complexity:
At each step, before choosing a submatrix, we have to find a maximum for the column, which takes \( m \) comparisons.

\[
T(m, n) = T(m, n/2) + m \\
= T(m, n/4) + 2m \\
= T(m, n/2^k) + km \\
= T(m, n/2^{\log n}) + m \log n = m + m \log n \\
= O(m \log n)
\]