COT 5407: Introduction to Algorithms
Giri NARASIMHAN
Definitions

Abstract Problem: defines a function from any allowable input to a corresponding output

Instance of a Problem: a specific input to abstract problem

Algorithm: well-defined computational procedure that takes an instance of a problem as input and produces the correct output

An Algorithm must halt on every input with correct output.
Sorting

- Input is a sequence of $n$ items that can be compared.
- Output is an ordered list of those $n$ items
  - I.e., a reordering or permutation of the input items such that the items are in sorted order
- Fundamental problem that has received a lot of attention over the years.
- Used in many applications.
- Scores of different algorithms exist.
- Task: To compare algorithms
  - On what bases?
    - Time
    - Space
    - Other
Sorting Algorithms

- Number of Comparisons
- Number of Data Movements
- Additional Space Requirements
Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort
Worst-Case Time Analysis

- **Two Techniques:**
  1. **Counts and Summations:**
     - Count number of steps from pseudocode and add
  2. **Recurrence Relations:**
     - Use invariant, write down recurrence relation and solve it

We will use big-Oh notation to write down time and space complexity (for both worst-case & average-case analyses).

Compute worst possible time of all input instances of length N.
Definition of big-Oh

- We say that
  - $F(n) = O(G(n))$

  If there exists two positive constants, $c$ and $n_0$, such that
  - For all $n \geq n_0$, we have $F(n) \leq c \cdot G(n)$

  Thus, to show that $F(n) = O(G(n))$, you need to find two positive constants that satisfy the condition mentioned above.

  Also, to show that $F(n) \neq O(G(n))$, you need to show that for any value of $c$, there does not exist a positive constant $n_0$ that satisfies the condition mentioned above.
SelectionSort – Worst-case analysis

SelectionSort (array A)
1 \( N \leftarrow \text{length}[A] \)
2 for \( p \leftarrow 1 \) to \( N \)
   do \( \triangleright \) Compute \( j \)
3 \( j \leftarrow p \)
4 for \( m \leftarrow p + 1 \) to \( N \)
   do if \( (A[m] < A[j]) \)
5   then \( j \leftarrow m \)
6 \( \triangleright \) Swap \( A[p] \) and \( A[j] \)
7 \( \text{temp} \leftarrow A[p] \)
8 \( A[p] \leftarrow A[j] \)
9 \( A[j] \leftarrow \text{temp} \)

\( N-p \) comparisons

3 data movements
SelectionSort: Worst-Case Analysis

- Data Movements
  \[ \sum_{p=1}^{N} 3 = 3 \times N = O(N) \]

- Number of Comparisons
  \[
  \begin{align*}
  &= \sum_{p=1}^{N} (N - p) \\
  &= \sum_{p=1}^{N} N - \sum_{p=1}^{N} p \\
  &= (N \times N) - (N)(N + 1)/2 \\
  &= O(N^2)
  \end{align*}
  \]

- Time Complexity = \( O(N^2) \)
SelectionSort – Space Complexity

- Temp Space
  - No extra arrays or data structures
  - O(1)
Invariant for SelectionSort

- An appropriate invariant has a parameter related to the progress of the algorithm (e.g., iteration number)
- An appropriate invariant helps in proving algorithm is correct
- “At the end of iteration p, the p smallest items are in their correct location”
MergeSort

- Divide-and-Conquer Strategy
- Divide array into two sublists of roughly equal length
- Sort each sublist “recursively”
- Merge two sorted lists to get final sorted list
  - Assumption: Merging is faster than sorting from fresh
- Most of the work is done in merging
- Process described using a tree
  - Top-down process: Divide each list into 2 sublists
  - Bottom-up process: Merge two sorted sublists into one sorted sublist
Figure 2.4  The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.
Merge uses an extra array & lots of data movements.
**Assumption:** Array A is sorted from \([p..q]\) and from \([q+1..r]\).

**Space:** Two extra arrays L and R are used.

**Sentinel Items:** Two sentinel items placed in lists L and R.

**Merge:** The smaller of the item in L and item in R is moved to next location in A.

**Time:** \(O(\text{length of lists})\)

---

```plaintext
MERGE(A, p, q, r)
1 n_1 \leftarrow q - p + 1
2 n_2 \leftarrow r - q
3 create arrays L[1..n_1 + 1] and R[1..n_2 + 1]
4 for i \leftarrow 1 to n_1
5 \quad do L[i] \leftarrow A[p + i - 1]
6 for j \leftarrow 1 to n_2
7 \quad do R[j] \leftarrow A[q + j]
8 L[n_1 + 1] \leftarrow \infty
9 R[n_2 + 1] \leftarrow \infty
10 i \leftarrow 1
11 j \leftarrow 1
12 for k \leftarrow p to r
13 \quad do if L[i] \leq R[j]
14 \quad \quad then A[k] \leftarrow L[i]
15 \quad \quad i \leftarrow i + 1
16 \quad else A[k] \leftarrow R[j]
17 \quad j \leftarrow j + 1
```
MergeSort

\[
\text{MERGE-SORT}(A, p, r) \\
1 \quad \text{if } p < r \\
2 \quad \text{then } q \leftarrow \lceil (p + r)/2 \rceil \\
3 \quad \text{MERGE-SORT}(A, p, q) \\
4 \quad \text{MERGE-SORT}(A, q + 1, r) \\
5 \quad \text{MERGE}(A, p, q, r)
\]

Time Complexity Recurrence: \( T(N) = 2T(N/2) + O(N) \)
What is the right invariant for MergeSort?
## Solving Recurrence Relations

<table>
<thead>
<tr>
<th>Recurrence; Cond</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a = b$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a &lt; b$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{log_b a-c})$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{log_b a})$</td>
<td>$T(n) = \Theta(n^{log_b a} \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$; $af(n/b) \leq cf(n)$</td>
<td>$T(n) = \Omega(n^{log_b a} \log n)$</td>
</tr>
</tbody>
</table>
Solving Recurrences: Recursion-tree method

- Substitution method fails when a good guess is not available
- Recursion-tree method works in those cases
  - Write down the recurrence as a tree with recursive calls as the children
  - Expand the children
  - Add up each level
  - Sum up the levels
- Useful for analyzing divide-and-conquer algorithms
- Also useful for generating good guesses to be used by substitution method
Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$.

Part (d) shows $F(n)$, which is progressively expanded in (d) to form the recursion tree. The fully expanded tree in part (c) has $\log n + 1$ levels (i.e., it has height $\log n$, as indicated), and each level contributes a total cost of $cn$. The total cost, therefore, is $cn \log n + cn$, which is $\Theta(n \log n)$. 
Figure 4.1: The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part (a) shows $T(n)$, which is progressively expanded in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).
Figure 4.2  A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$. 

Total: $O(n \log n)$
Solving Recurrences using Master Theorem

Master Theorem:

Let $a, b \geq 1$ be constants, let $f(n)$ be a function, and let $T(n) = aT(n/b) + f(n)$

1. If $f(n) = O(n^{\log_b a - e})$ for some constant $e > 0$, then
   $$ T(n) = \Theta(n^{\log_b a}) $$

2. If $f(n) = \Theta(n^{\log_b a})$, then
   $$ T(n) = \Theta(n^{\log_b a} \log n) $$

3. If $f(n) = \Omega(n^{\log_b a + e})$ for some constant $e > 0$, then
   $$ T(n) = \Theta(f(n)) $$