CSCI-GA.1144-001

PAC II

Lecture 8: Algorithms II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
A Quick Refresh

- We assume we execute our algorithm on RAM.
  - In RAM, instructions are executed one after the other, with no concurrency.
  - RAM model contains instructions available in common computers.
  - Each instruction takes a constant amount of time.
  - We do not attempt to model memory hierarchy.

- We care more about the worst-case scenario.
Sorting

• **Input**: sequence of n numbers
  \[<a_1, a_2, ..., a_n>\]

• **Output**: a permutation of the input sequence \[<b_1, b_2, ..., b_n>\] such that:
  \[b_1 \leq b_2 \leq ... \leq b_n\]
Insertion Sort

• Adding a new element to a sorted list will keep the list sorted if the element is inserted in the correct place

• A single element list is sorted

• Inserting a second element in the proper place keeps the list sorted

• This is repeated until all the elements have been inserted into the sorted part of the list
Insertion Sort

INSERTION-SORT (A)

1 for j = 2 to length[A]
2 key = A[j]
3 // Insert A[j] into the sorted sequence A[1...j-1]
4 i = j - 1
5 while i > 0 and A[i] > key
6 A[i+1] = A[i]
7 i = 1 - 1
8 A[i+1] = key

Source: “Introduction to Algorithms” 3rd Edition
Insertion Sort
Algorithm Analysis

- In general, the time taken by an algorithm grows with the size of the input.
- So, it is traditional to describe the running time of a program as a function of the size of its input.
- The running time of an algorithm on a particular input is the number of primitive operations executed.
- We care about the worst-case scenario.
Analyzing Insertion Sort

**INSERTION-SORT (A)**

1. for \( j = 2 \) to \( \text{length}[A] \)
2. \( \text{key} = A[j] \)
3. // Insert \( A[j] \) into the sorted sequence \( A[1...j-1] \)
4. \( i = j - 1 \)
5. while \( i > 0 \) and \( A[i] > \text{key} \)
7. \( i = i - 1 \)
8. \( A[i+1] = \text{key} \)

\( t_j \) is the number of times the while loop test in step 5 is executed for that value of \( j \).

**Source:** “Introduction to Algorithms” 3rd Edition
Analyzing Insertion Sort

\[ T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1). \]

Best case:
A is sorted
\[ t_j = 1 \text{ in step 5 for all } j \]
\[ T(n) = an+b \]

Worst case:
A is reverse sorted
\[ t_j = j \]
\[ T(n) = \frac{c_5 (n(n+1))}{2} - 1 + \left( \frac{c_6 (n(n-1))}{2} + c_7 \left( \frac{n(n-1)}{2} \right) \right) + c_8(n-1) \]
\[ = \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \]
\[ = an^2 + bn + n \]

Source: “Introduction to Algorithms” 3rd Edition
How to Design An Algorithm

- Incremental approach: similar to insertion sort
- Divide-and-conquer approach:
  - Divide: break the problem into similar subproblems similar to the original problem but smaller in size
  - Conquer: solve the subproblems recursively
  - Combine: combine the solutions to create the solution of the original problem
Merge Sort

Sorts the elements of subarray A[p..r].
Initially: p = 1 and r = length[A]

```
MERGE-SORT(A, p, r)
1  if p < r
2     q = ⌊(p + r)/2⌋
3     MERGE-SORT(A, p, q)
4     MERGE-SORT(A, q + 1, r)
5     MERGE(A, p, q, r)
```
Merge Sort

\[
\text{MERGE}(A, p, q, r) \\
1 \quad n_1 = q - p + 1 \\
2 \quad n_2 = r - q \\
3 \quad \text{let } L[1..n_1 + 1] \text{ and } R[1..n_2 + 1] \text{ be new arrays} \\
4 \quad \text{for } i = 1 \text{ to } n_1 \\
5 \quad \quad L[i] = A[p + i - 1] \\
6 \quad \text{for } j = 1 \text{ to } n_2 \\
7 \quad \quad R[j] = A[q + j] \\
8 \quad L[n_1 + 1] = \infty \\
9 \quad R[n_2 + 1] = \infty \\
10 \quad i = 1 \\
11 \quad j = 1 \\
12 \quad \text{for } k = p \text{ to } r \\
13 \quad \quad \text{if } L[i] \leq R[j] \\
14 \quad \quad \quad A[k] = L[i] \\
15 \quad \quad \quad i = i + 1 \\
16 \quad \quad \text{else } A[k] = R[j] \\
17 \quad \quad \quad j = j + 1
\]

Source: “Introduction to Algorithms” 3rd Edition
Execution Example

• Partition

\[
\begin{array}{c|c|c|c|c}
7 & 2 & 9 & 4 & 3 & 8 & 6 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
\vdots \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
7 & 2 & 9 & 4 & 3 & 8 & 6 & 1 & \vdots \\
\end{array}
\]
Execution Example (cont.)

- Recursive call, partition

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4
```

```
7 | 2 | 9 | 4
```

```
3 | 8 | 6 | 1
```

```
1 | 2 | 3 | 4 | 6 | 7 | 8 | 9
```
Execution Example (cont.)

- Recursive call, partition
Execution Example (cont.)

- Recursive call, base case

```
7 2 9 4 3 8 6 1
7 2 9 4
7 2
7
```

```
1 2 3 4 6 7 8 9
```
Execution Example (cont.)

- Recursive call, base case

```
7 2 9 4 3 8 6 1
```

```
7 2 9 4
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```

```
7 2
```
Execution Example (cont.)

• Merge
Execution Example (cont.)

- Recursive call, ..., base case, merge
Execution Example (cont.)

• Merge

7 2 9 4 | 3 8 6 1

7 2 9 4 → 2 4 7 9

7 2 → 2 7
9 4 → 4 9

7 2 9 4 | 3 8 6 1
Execution Example (cont.)

- Recursive call, ..., merge, merge
Execution Example (cont.)

• Merge

```
7 2 9 4 3 8 6 1 → 1 2 3 4 6 7 8 9
```

```
7 2 9 4 2 4 7 9
3 8 6 1 1 3 6 8
```

```
7 2 2 7
9 4 4 9
3 8 3 8
6 1 1 6
```

```
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1
```
Analyzing Merge Sort

- \( T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \)
  
  \[ T(n) = D(n) + 2T(n/2) + C(n) = c + 2T(n/2) + cn \]
Analyzing Merge Sort

- \( T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \)
  \[ = D(n) + 2T(n/2) + C(n) \]
  \[ = c + 2T(n/2) + cn \]

Source: “Introduction to Algorithms” 3rd Edition

Analysis:
- Total for conquer: \( cn \lg n \)
- \( T(n) = cn \lg n + cn \)
Bubble Sort

• If we compare pairs of adjacent elements and none are out of order, the list is sorted

• If any are out of order, we must swap them to get an ordered list

• Bubble sort will make passes though the list swapping any adjacent elements that are out of order
Bubble Sort

• After the first pass, we know that the largest element must be in the correct place

• After the second pass, we know that the second largest element must be in the correct place

• Because of this, we can shorten each successive pass of the comparison loop
Bubble Sort Example
Bubble Sort Algorithm

numberOfPairs = N
swappedElements = true
while (swappedElements) do
  numberOfPairs = numberOfPairs - 1
  swappedElements = false
  for i = 1 to numberOfPairs do
    if (A[i] > A[i + 1]) then
      Swap( A[i], A[i + 1] )
      swappedElements = true
    end if
  end for
end while
Best-Case Analysis

• If the elements start in sorted order, the *for* loop will compare the adjacent pairs but not make any changes

• So the `swappedElements` variable will still be false and the *while* loop is only done once

• There are $N-1$ comparisons in the best case
Worst-Case Analysis

- In the worst case the while loop must be done as many times as possible. This happens when the data set is in the reverse order.

- Each pass of the for loop must make at least one swap of the elements.

- The number of comparisons will be:

$$W(N) = \sum_{i=1}^{N-1} (N - i) = \sum_{k=1}^{N-1} k = \sum_{i=1}^{N-1} i = \frac{(N-1)*N}{2} = O(N^2)$$
Quicksort Algorithm

- Another divide-and-conquer algorithm
- Quicksort is usually $O(n \log n)$ but in the worst case slows to $O(n^2)$

Given an array of $n$ elements (e.g., integers):
- If array only contains one element, return
- Else
  - pick one element to use as pivot.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results
Quicksort

• **Divide step:**
  – Pick any element (*pivot*) $v$ in $S$
  – Partition $S - \{v\}$ into two disjoint groups
    $S1 = \{x \in S - \{v\} \mid x \leq v\}$
    $S2 = \{x \in S - \{v\} \mid x \geq v\}$

• **Conquer step:** recursively sort $S1$ and $S2$

• **Combine step:** the sorted $S1$ (by the time returned from recursion), followed by $v$, followed by the sorted $S2$ (i.e., nothing extra needs to be done)
Example
Pseudo-code

QUICKSORT($A, p, r$)
1 \textbf{if} $p < r$
2 \quad $q = \text{PARTITION}(A, p, r)$
3 \quad \text{QUICKSORT}(A, p, q - 1)$
4 \quad \text{QUICKSORT}(A, q + 1, r)$

PARTITION($A, p, r$)
1 \quad $x = A[r]$
2 \quad $i = p - 1$
3 \quad \textbf{for} $j = p$ \textbf{to} $r - 1$
4 \quad \quad \textbf{if} $A[j] \leq x$
5 \quad \quad \quad $i = i + 1$
6 \quad \quad \text{exchange} $A[i]$ \text{ with } $A[j]$
7 \quad \text{exchange} $A[i + 1]$ \text{ with } $A[r]$
8 \quad \textbf{return} $i + 1$
More Sorting Algorithms

- Shell sort
- Heap sort
- Radix sort
Binary Search

• Binary search. Given value and sorted array $a[]$, find index $i$ such that $a[i] = value$, or report that no such index exists.

• Invariant. Algorithm maintains $a[lo] \leq value \leq a[hi]$.

• Ex. Binary search for 33.
Binary Search
Binary Search

```
<table>
<thead>
<tr>
<th>6</th>
<th>13</th>
<th>14</th>
<th>25</th>
<th>33</th>
<th>43</th>
<th>51</th>
<th>53</th>
<th>54</th>
<th>72</th>
<th>84</th>
<th>93</th>
<th>95</th>
<th>96</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

↑
lo

↑
hi
```
Binary Search
Binary Search

lo  hi
Binary Search
Binary Search

lo

hi
Binary Search
Binary Search
Efficiency of binary search

• If $n$ represents the number of names, the maximum number of searches $x$ necessary to find a name is the smallest integer that satisfies the inequality $2^x > n$.

\[
2^x > n \\
\log (2^x) > \log n \\
x \log 2 > \log n
\]

The maximum number of searches is the smallest integer greater than $\log n / \log 2$. 

## Efficiency of binary search

<table>
<thead>
<tr>
<th># of elements</th>
<th>Maximum sequential searches necessary</th>
<th>Maximum binary searches necessary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>5,000</td>
<td>5,000</td>
<td>13</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>50,000</td>
<td>50,000</td>
<td>16</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
</tbody>
</table>

With the incredible speed of today’s computers, a binary search becomes necessary only when the number of elements is large.
Conclusions

• In this lecture, we have seen examples of basic algorithms used in many applications and compared their complexities.

• Heuristics are the way to go if we cannot get the exact/best results with reasonable resources.