CSE 373: Data Structures and Algorithms

Sorting and recurrence analysis techniques

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...
Sorting problem statement

Given n comparable elements, rearrange them in an increasing order.

Input
- An array $A$ that contains n elements
- Each element has a key $k$ and an associated data
- Keys support a comparison function (e.g., keys implement a Comparable interface)

Expected output
- An output array $A$ such that for any $i$ and $j$,
  - $A[i] \leq A[j]$ if $i < j$  (increasing order)
- Array $A$ can also have elements in reverse order (decreasing order)
Desired properties in a sorting algorithm

Stable
- In the output, equal elements (i.e., elements with equal keys) appear in their original order

In-place
- Algorithm uses a constant additional space, $O(1)$ extra space

Adaptive
- Performs better when input is almost sorted or nearly sorted
  - (Likely different big-O for best-case and worst-case)

Fast. $O(n \log n)$

No algorithm has all of these properties. So choice of algorithm depends on the situation.
Sorting algorithms – High-level view

- $O(n^2)$
  - Insertion sort
  - Selection sort
  - Quick sort (worst)

- $O(n \log n)$
  - Merge sort
  - Heap sort
  - Quick sort (avg)

- $\Omega(n \log n)$ -- lower bound on comparison sorts

- $O(n)$ – non-comparison sorts
  - Bucket sort (avg)
A framework to think about sorting algos

Some questions to consider when analyzing a sorting algorithm.

Loop/step invariant: ________________________________
What is the state of the data during each step while sorting

Runtime:  Worst __________      Average __________       Best __________

Input:        Worst ______________________      Best ______________________
Stable ____________________           In-place ____________________  Adaptive ____________________

Operations:  Comparisons __________________________ Moves
> or < or approx. equal

Data structure __________________ Which data structure is better suited for this algo
Insertion sort

**Idea:** At step $i$, insert the $i^{th}$ element in the correct position among the first $i$ elements.

![Insertion sort visualization](https://visualgo.net/en/sorting)

**Loop/step invariant:** Subarray $0 \ldots i-1$ is sorted

**Runtime:**
- Worst $O(n^2)$
- Average $O(n^2)$
- Best $O(n)$

**Input:**
- Worst Reverse sorted
- Best Sorted

**Stable:** Yes (if careful)

**In-place:** Yes

**Adaptive:** Yes

**Operations:**
- Comparisons depends on case
- Moves

**Data structure:** Linked list (ideal to insert elements in a list)
Selection sort

**Idea:** At step \(i\), find the smallest element among the not-yet-sorted elements (\(i \ldots n\)) and swap it with the element at \(i\).

**Loop/step invariant:**

- 0 \(\ldots\) \(i-1\) is sorted and contains the smallest element in array

**Runtime:**
- Worst \(\Theta(n^2)\)
- Average \(\Theta(n^2)\)
- Best \(\Theta(n^2)\)

**Input:**
- Worst
- Best

**Stable:** Yes (if careful)
- In-place: Yes
- Adaptive: No

**Operations:**
- Comparisons \(\Rightarrow\) Moves

**Data structure:** Array. To get min, could use heap \(\Rightarrow\) heap sort
Heap sort

Idea: $buildHeap$ with all $n$ elements

```plaintext
for i = 0 to n do
    A[i] = removeMin()
end for
```

Loop/step invariant: ____________________________

Runtime: Worst $O(n \log n)$, Average _________, Best _________

Input: Worst ______________, Best ______________

Stable __________, In-place No (But Can be), Adaptive ______________

Operations: Comparisons ______________, Moves

Data structure ______________
In-place heap sort

Idea:

1. Treat initial array as a heap

2. When you call removeMin(), that frees up a slot towards the end in the array. Put the extract min element there.

3. More specifically, when you remove the \( i^{th} \) element, put it at \( A[n - i] \)

4. This gives you a reverse sorted array. But easy to fix in-place.
Design technique: Divide-and-conquer

Very important technique in algorithm to attack problems

Three steps:
1. Divide: Split the original problem into smaller parts
2. Conquer: Solve individual parts independently (think recursion)
3. Combine: Put together individual solved parts to produce an overall solution

Merge sort and Quick sort are classic examples of sorting algorithms that use this technique
Merge sort

To sort a given array,

Divide: Split the input array into two halves

Conquer: Sort each half independently

Combine: Merge the two sorted halves into one sorted whole (HW3 Problem 6!)

function mergeSort(A)
    if A.length == 1 then
        return A;
    else
        mid = A.length / 2
        firstHalf = mergeSort(new [0, ... mid])
        secondHalf = mergeSort(new [mid+1, ... ])
        return merge(firstHalf, secondHalf)
    end if
end function

Visualization: https://visualgo.net/en/sorting
Merge sort

Split array in the middle
Sort the two halves
Merge them together

\[ T(n) = \begin{cases} 
  c_1 & \text{if } n \leq 1 \\
  2T\left(\frac{n}{2}\right) + c_2 n & \text{otherwise} 
\end{cases} \]
Review: Unfolding (technique 1)

\[ T(n) = 2T(n/2) + c_2n \]

\[ = 2 \left( 2T \left( \frac{n}{2 \cdot 2} \right) + c_2 \frac{n}{2} \right) + c_2n \]

\[ = 2^2 T \left( \frac{n}{2 \cdot 2} \right) + c_2n + c_2n \]

\[ = 2^2 \left( 2T \left( \frac{n}{2^3} \right) + c_2 \frac{n}{2^2} \right) + c_2n + c_2n \]

\[ = 2^3 T \left( \frac{n}{2^3} \right) + c_2n + c_2n + c_2n \]

\[ = \ldots \]

\[ = 2^{\log n} T(1) + c_2n + c_2n + \cdots + c_2n \]

\[ \text{about } \log(n) \text{ times} \]

\[ = c_1n + c_2n \log n \]

\[ T(n) = \begin{cases} 
  c_1 & \text{if } n \leq 1 \\
  2T \left( \frac{n}{2} \right) + c_2n & \text{otherwise} 
\end{cases} \]
Technique 2: Tree method

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases} \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of Nodes at level</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last recursive level:
Technique 2: Tree method

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases} \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of Nodes at level</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last recursive level:
Technique 2: Tree method

\[ T(n) = \begin{cases} 
  1 & \text{if } n = 1 \\
  2T(n/2) + n & \text{otherwise}
\end{cases} \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of Nodes at level</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last recursive level:
Technique 2: Tree method

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases} \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of Nodes at level</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last recursive level:
Technique 2: Tree method

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of Nodes at level</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \frac{n}{2} )</td>
<td>( \frac{n}{2} )</td>
<td>( n )</td>
</tr>
<tr>
<td>1</td>
<td>( \frac{n}{4} )</td>
<td>( \frac{n}{4} )</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>2</td>
<td>( \frac{n}{8} )</td>
<td>( \frac{n}{8} )</td>
<td>( \frac{n}{4} )</td>
</tr>
<tr>
<td>( i )</td>
<td>( \frac{n}{2^i} )</td>
<td>( \frac{n}{2^i} )</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>base</td>
<td>( n )</td>
<td>1</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Last recursive level: \( \log n - 1 \)

\[ T(n) = \text{base case} + \sum_{i=0}^{\log n-1} \frac{n}{2^i} \]

\[ \log n \quad \text{base} \]

\[ \log n + \sum_{i=0}^{\log n-1} \frac{n}{2^i} = n + n \log n \]
Technique 2: Tree method

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of Nodes at level</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(n/2)</td>
<td>(n)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>(n/4)</td>
<td>(n)</td>
</tr>
<tr>
<td>(i)</td>
<td>(2^i)</td>
<td>(n/2^i)</td>
<td>(n)</td>
</tr>
<tr>
<td>base</td>
<td>(2^{\log_2 n})</td>
<td>1</td>
<td>(n)</td>
</tr>
</tbody>
</table>

Last recursive level: \(\log n - 1\)

Combining it all together...

\[ T(n) = n + \sum_{i=0}^{\log_2 n - 1} n \]
Tree Method Practice

\[ T \left( \frac{n}{4} \right) + T \left( \frac{n}{4} \right) + T \left( \frac{n}{4} \right) + cn^2 \]

<table>
<thead>
<tr>
<th>Level (i)</th>
<th>Number of Nodes</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( \alpha \times cn^2 )</td>
<td>( cn^2 )</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>( \alpha \times c(n/16)^2 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>( \alpha \times c(n/16)^2 )</td>
<td></td>
</tr>
<tr>
<td>( i )</td>
<td>( 3^i )</td>
<td>( \alpha \times c(n/16)^2 )</td>
<td>( \alpha(3n/16)^i )</td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td>( 4 )</td>
</tr>
</tbody>
</table>

\( \alpha \) is some constant factor.

Level (i) Table:

\[ \sum_{i=0}^{\infty} \alpha \times \left( \frac{3}{4} \right)^i = 4 \cdot \frac{1}{1 - 3/4} = 8 \]

\( \sum_{i=0}^{\infty} \alpha \times \left( \frac{3}{4} \right)^i \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \frac{n}{16} \]

\[ \frac{\alpha n}{16} \]

\[ \sum_{i=0}^{\infty} \left( \frac{3}{4} \right)^i \cdot \left( \frac{n}{16} \right)^i \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \left( \frac{n}{16} \right)^i \]

\[ \frac{\alpha n}{16} \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \left( \frac{n}{16} \right)^i \]

\[ \frac{\alpha n}{16} \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \left( \frac{n}{16} \right)^i \]

\[ \frac{\alpha n}{16} \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \left( \frac{n}{16} \right)^i \]

\[ \frac{\alpha n}{16} \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \left( \frac{n}{16} \right)^i \]

\[ \frac{\alpha n}{16} \cdot \left( \frac{n}{16} \right)^i = 4 \cdot \frac{1}{1 - 3/4} \cdot \left( \frac{n}{16} \right)^i \]
Tree Method Practice

\[ T \left( \frac{n}{4} \right) + T \left( \frac{n}{4} \right) + T \left( \frac{n}{4} \right) + cn^2 \]

<table>
<thead>
<tr>
<th>Level (i)</th>
<th>Number of Nodes</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( cn^2 )</td>
<td>( cn^2 )</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>( c \left( \frac{n}{4} \right)^2 )</td>
<td>( \frac{3}{16} cn^2 )</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>( c \left( \frac{n}{16} \right)^2 )</td>
<td>( \frac{9}{256} cn^2 )</td>
</tr>
<tr>
<td>( i )</td>
<td>( 3^i )</td>
<td>( c \left( \frac{n}{4^i} \right)^2 )</td>
<td>( \left( \frac{3}{16} \right)^i cn^2 )</td>
</tr>
<tr>
<td>base</td>
<td>( 3^{\log_4 n} )</td>
<td>4</td>
<td>( 4 \cdot 3^{\log_4 n} )</td>
</tr>
</tbody>
</table>

Last recursive level: \( \log_4 n - 1 \)

Combining it all together...

\[ T(n) = 4n^{\log_4 3} + \sum_{i=0}^{\log_4 n - 1} \left( \frac{3}{16} \right)^i cn^2 \]
**Technique 3: Master Theorem**

Given a recurrence of the following form:

\[
T(n) = \begin{cases} 
  d & \text{when } n = 1 \\
  aT\left(\frac{n}{b}\right) + nc & \text{otherwise}
\end{cases}
\]

Where \(a\), \(b\), and \(c\) are constants, then \(T(n)\) has the following asymptotic bounds:

- If \(\log_b a < c\) then \(T(n) \in \Theta(n^c)\)
- If \(\log_b a = c\) then \(T(n) \in \Theta(n^c \log_2 n)\)
- If \(\log_b a > c\) then \(T(n) \in \Theta(n^{\log_b a})\)
Apply Master Theorem

Given a recurrence of the form:

\[ T(n) = \begin{cases} 
  d \text{ when } n = 1 \\
  aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} 
\end{cases} \]

- If \( \log_b a < c \) then \( T(n) \in \Theta(n^c) \)
- If \( \log_b a = c \) then \( T(n) \in \Theta(n^c \log n) \)
- If \( \log_b a > c \) then \( T(n) \in \Theta(n^{\log_b a}) \)

\[ T(n) = \begin{cases} 
  1 \text{ when } n \leq 1 \\
  2T\left(\frac{n}{2}\right) + n \text{ otherwise} 
\end{cases} \]

- \( a = 2 \)
- \( b = 2 \)
- \( c = 1 \)
- \( d = 1 \)

\( \log_b a = c \Rightarrow \log_2 2 = 1 \)

\[ T(n) \in \Theta(n^c \log n) \Rightarrow \Theta(n^1 \log n) \]
Reflecting on Master Theorem

Given a recurrence of the form:

\[
T(n) = \begin{cases} 
d \text{ when } n = 1 \\ 
aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} 
\end{cases}
\]

- If \( \log_b a < c \) then \( T(n) \in \Theta(n^c) \)
- If \( \log_b a = c \) then \( T(n) \in \Theta(n^c \log_2 n) \)
- If \( \log_b a > c \) then \( T(n) \in \Theta(n^{\log_b a}) \)

**height** \( \approx \log_b a \)

**branchWork** \( \approx n^c \log_b a \)

**leafWork** \( \approx d(n^{\log_b a}) \)

The \( \log_b a < c \) case
- Recursive case conquers work more quickly than it divides work
- Most work happens near “top” of tree
- Non recursive work in recursive case dominates growth, \( n^c \) term

The \( \log_b a = c \) case
- Work is equally distributed across call stack (throughout the “tree”)
- Overall work is approximately work at top level x height

The \( \log_b a > c \) case
- Recursive case divides work faster than it conquers work
- Most work happens near “bottom” of tree
- Leaf work dominates branch work
Recurrence analysis techniques

1. Unfolding method
   - more of a brute force method
   - Tedious but works

2. Tree methods
   - more scratch work but less error prone

3. Master theorem
   - quick, but applicable only to certain type of recurrences
   - does not give a closed form (gives big-Theta)