CS 162
Intro to Programming II

Sorting II
MergeSort

- Suppose you have 2 sorted arrays
- How do you merge them into a sorted array?
  - Just look at the first element in both arrays
  - Put the smaller element into the new array.
  - Then look at the next element in the array that the smaller element came from and compare with the current element in the other array
  - Repeat...
Merging

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>23</td>
</tr>
</tbody>
</table>
The complexity of merge sort is $O(n)$
MergeSort

• But wait a second...you aren’t given two already sorted arrays

• Recursion to the rescue!
  – Keep dividing the array in half (until the subarray size is 1)
  – A subarray of size 1 is sorted
  – Merge the sorted halves together
MergeSort Illustrated
MergeSort Illustrated

1 2 2 3 4 5 6 7

2 4 5 7

2 5 4 7

5 2 4 7

1 2 3 6

1 3 2 6

1 3 2 6
Complexity

• Each recursive call operates on an array that is half the size of the original array

• How many level’s to get to arrays size of 1? About O(log n)
Complexity

- Each recursive call operates on an array that is half the size of the original array.

- Each time you do a sort() you do a merge which is O(n). Each level is O(n).
Complexity

$O(n)$

$O(n)$

$O(n)$

For $O(\log n)$ levels

Total complexity is $O(n \log n)$

Best and worst case
Quicksort

• Maybe the most commonly used algorithm
• Quicksort is also a divide and conquer algorithm
• Advantage over mergesort: no temporary arrays to sort and merge the partial results
Quicksort

Has two steps:

1. Partition the data around a special element called the pivot

2. Recursively sort each partition
Quicksort

Pick the first element (ie. 5) as the pivot. Everything < 5 goes in the left partition, everything >= 5 goes in the right partition

Before partitioning

| 5 | 3 | 2 | 6 | 4 | 1 | 3 | 7 |

After partitioning

| 3 | 3 | 2 | 1 | 4 | 6 | 5 | 7 |

Left

Right
Quicksort

• After partitioning

3 3 2 1 4 6 5 7

Left Right

To continue you apply the algorithm to each partition:

1. Find the pivot
2. Partition that partition
Quicksort

\[
\text{int partition(int a[], int from, int to)} \\
\{ \\
\quad \text{int pivot} = a[\text{from}]; \\
\quad \text{int } i = \text{from} - 1; \\
\quad \text{int } j = \text{to} + 1; \\
\quad \text{while } (i < j) \\
\quad \quad \{ \\
\quad \quad \quad i++; \text{ while } (a[i] < \text{pivot}) i++; \\
\quad \quad \quad j--; \text{ while } (a[j] > \text{pivot}) j--; \\
\quad \quad \quad \text{if} (i < j) \text{ swap}(i, j); \\
\quad \quad \} \\
\quad \text{return } j; \\
\} \\
\]
Quicksort

/**
 * Code snippet for Quicksort
 * from and to specify the range of the array that
 * you want to sort. For example, from = 0 and to =
 * array size - 1 will sort the entire array.
 */
void quicksort(int a[], int from, int to)
{
    if (from >= to) return;
    int p = partition(a, from, to);
    quicksort(a, from, p);
    quicksort(a, p+1, to);
}
Complexity

Best Case-
  scrambled  \(O(n \log n)\)

Worst Case-
  already sorted  \(O(n^2)\)

Average Case-
  randomly picked pivot  \(O(n^2)\)