Today
  • Merge Sort

Assignments
  • HW6 due today
  • HW7 out (due Tues)
Merge Sort

The basic idea (note: recursive “divide and conquer” approach)

• divide list into two halves
• sort each half (recursive step)
• merge the sorted halves into sorted list

Split Step:

\[
\begin{array}{cccc}
29 & 10 & 14 & 13 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
29 & 10 & 14 & 13 \\
\end{array}
\]

Sort Step (via more splitting/merging):

\[
\begin{array}{cc}
29 & 10 \\
14 & 13 \\
\end{array}
\Rightarrow
\begin{array}{cc}
10 & 29 \\
13 & 14 \\
\end{array}
\]

Merge Step:

\[
\begin{array}{cc}
10 & 29 \\
13 & 14 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
10 & 13 & 14 & 29 \\
\end{array}
\]
Initial list \((n=4)\)

\[
\begin{array}{c}
29 \\
10 \\
14 \\
13 \\
\end{array}
\quad \text{Split}
\]

\[
\begin{array}{c}
29 \\
10 \\
\end{array}
\quad \begin{array}{c}
14 \\
13 \\
\end{array}
\]

(1-element lists are sorted by definition)

\[
\begin{array}{c}
29 \\
10 \\
\end{array}
\quad \begin{array}{c}
14 \\
13 \\
\end{array}
\]

\[
\begin{array}{c}
10 \\
29 \\
\end{array}
\quad \begin{array}{c}
13 \\
14 \\
\end{array}
\]

Merge

\[
\begin{array}{c}
10 \\
13 \\
14 \\
29 \\
\end{array}
\quad \text{Merge}
\]
The algorithm (psuedocode):

Note: start, mid, and end are array indexes

MergeSort(T array[], int start, int end)
1. if start < end
2. mid = (start + end) / 2
3. mergesort(array, start, mid) # recursive step
4. mergesort(array, mid + 1, end) # recursive step
5. merge(array, start, mid, end) # merge sorted sublists

Merge(T array[], int start, int mid, int end)
1. T tmp[(end - start) + 1] # tmp array size n
2. first1 := start
3. first2 := mid + 1
4. i := 0
5. while first1 <= mid and first2 <= end do # merge into tmp
6. if array[first1] < array[first2] then
7. tmp[i++] = array[first1++]
8. else
9. tmp[i++] = array[first2++]
10. while first1 <= mid do # copy rest
11. tmp[i++] = array[first1++]
12. while first2 <= mid do # copy rest
13. tmp[i++] = array[first2++]
14. for i = 0 to (end - start) do # copy to array
15. array[start + i] = tmp[i]
What is the time complexity of the merge step?

- Assume number of elements to be merged is \( n \)
  - \( n = \) length of first half + length of second half

- The merge step requires:
  - \( n - 1 \) comparisons (worst case is interleaved values, e.g., \([a, c, e] \ [b, d, f]\))
  - \( n \) moves from original array into the temporary array
  - \( n \) moves from temporary array into original array

- Thus merge is \( O(n) \)
  - since \( n - 1 \) comparisons and \( 2n \) moves

What is the time complexity of mergesort?

- Like binary search, we halve the list roughly \( \log_2 n \)
  - thus, we make \( O(\log n) \) recursive calls

- In each recursive call merge costs \( O(n) \) (moves/comparisons)

- This means mergesort is \( O(n \log n) \)!
Observations on mergesort

- Fastest sorting algorithm we’ve seen yet
- There isn’t a best vs worst case for mergesort (same work regardless)
- To be fast, we are making a trade-off for space
  - we don’t sort the array “in place”
  - instead use a temporary array
  - which effectively doubles the space needed

<table>
<thead>
<tr>
<th>Sorting Alg</th>
<th>Best Case</th>
<th>Avg Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort for HW6

- We'll do merge sort “in place” by splicing and reattaching
- Linked lists are nice for this ...

We'll use a helper function for the merge sort implementation:

```c
Node* merge_sort(Node* left, int len)
```

This function will do **both** the “halving” and merging

- **left** is the left-most node in the list to consider
- **len** is the number of nodes to consider

**The general (high-level) algorithm**

```c
Node* merge_sort(Node* left, int len)
1. if len is 1 return left
2. find mid length
3. left := merge_sort(left, mid_len)
4. right := merge_sort(right, len - mid_len)
5. merge the two lists left and right
6. return first node of merged list
```

```c
void merge_sort()
1. head := merge_sort(head, length)
2. update tail ptr
```