CPSC 223
Algorithms & Data Abstract Structures

Lecture 12:
More on Sorting – Mergesort and Quicksort

Today …
• Continuing with Sorting
• Mergesort [Sect. 9.2, pp. 466-472]
• Quicksort [Sect. 9.2, pp. 472-484]
Mergesort

Mergesort [von Neumann, 1945]

- A “divide and conquer” approach
- The basic idea:
  - Divide the list into two halves
  - Sort each half
  - Merge the sorted halves

  ... Merging is fast when the two sublists are sorted

Initial list: \( n = 4 \)

<table>
<thead>
<tr>
<th>Split</th>
<th>Sort</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 10 14 13</td>
<td>29 10 14 13</td>
<td>10 29 13 14</td>
</tr>
<tr>
<td>29 10 14 13</td>
<td>10 29 13 14</td>
<td>10 13 14 29</td>
</tr>
</tbody>
</table>
Mergesort

• Given that mergesort merges two sorted lists to create a (larger) sorted list

• How do we sort the two sublists?
  – Using mergesort!
  – Divide the list into two halves, sort each half (using mergesort), and then merge the sorted halves

• Mergesort is defined using recursion
Mergesort (based on textbook)

void Mergesort(Entry theArray[], int n, int first, int last)
{
    if(first < last) {
        int mid = (first + last) / 2;
        Mergesort(theArray, n, first, mid);
        Mergesort(theArray, n, mid + 1, last);
        Merge(theArray, n, first, mid, last);
    }
}

Mergesort (based on textbook)

void Merge(Entry theArray[], int n, int first, int mid, int last)
{
    Entry tmpArray[n];
    int first1 = first, first2 = mid + 1, i = first1;
    for(; first1 <= mid && first2 <= last; i++) {
        if(theArray[first1] < theArray[first2])
            tmpArray[i] = theArray[first1++];
        else
            tmpArray[i] = theArray[first2++];
    }
    for(; first1 <= mid; i++, first1++)    // copy remaining from first half
        tmpArray[i] = theArray[first1];
    for(; first2 <= last; i++, first2++)    // copy remaining from sec. half
        tmpArray[i] = theArray[first2];
    for(int j = first; j <= last; j++)    // copy tmpArray to the array
        theArray[j] = tmpArray[j];
}
Mergesort

The *merge* step

- Assume number of elements to be merged is $n$
  - $n = \text{length of first half} + \text{length of second half}$

- The merge step requires
  - $n - 1$ comparisons (worst case)
  - $n$ moves from the original array to the temp array
  - $n$ moves from the temp array to the original array

- So merge costs $3n - 1$

- Which is $O(n)$

Mergesort

- How expensive is mergesort?
  - Each time we call mergesort twice (halving the list)
  - Assume $n$ items in the list
  - The recursion goes approximately $\log_2 n$ *levels deep*
  - At each level the merges cost a total of $O(n)$

- So mergesort is $O(n \log n)$ !!!
Mergesort

Why $\log_2 n$ levels of recursion?

- Each level adds \textit{twice} the number of sublists
- Each sublist is \textit{half} the size of the previous ones
- We stop “expanding” when sublists are of size 1

\begin{itemize}
\item Lets say $n = 8$
  \begin{itemize}
  \item The 1st level results in sublists of size $n / 2^1$
  \item The 2nd level results in sublists of size $n / 2^2$
  \item The 3rd level results in sublists of size $n / 2^3$
  \end{itemize}
\item We stop the recursion when $n / 2^r = 1$
  \begin{itemize}
  \item This means that $n = 2^r$
  \item So the number of levels (merges) is $\log_2 n = r$
  \end{itemize}
\end{itemize}

Mergesort

Mergesort is a \textit{“fast”} sort

\begin{itemize}
\item But it comes at a \textit{cost} ...
  \begin{itemize}
  \item Merging uses temporary space (tmpArray)
  \item We’re using more space to obtain time efficiency
  \end{itemize}
\end{itemize}
## Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Average 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>Mergesort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treesort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Quicksort
Quicksort

- Another “divide and conquer” approach
- The basic idea:
  - Pick a “pivot” element in the list
  - Put values smaller than the pivot on the left
  - Put values larger than the pivot on the right
  - Put pivot value in its final sorted position
  - Repeat on the left and right sublists

Initial list: \((n=4)\)

<table>
<thead>
<tr>
<th>Pick pivot</th>
<th>Partition and Place Pivot</th>
<th>Repeat on Sublists</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 10 32 13</td>
<td>29 10 13 32</td>
<td>13 10 29 32</td>
</tr>
<tr>
<td>29 10 32 13</td>
<td>13 10 29 32</td>
<td>10 13 29 32</td>
</tr>
</tbody>
</table>

Quicksort

Initial list \((n=4)\)

\(n = 2, n = 1\)
\(n = 1, n = 0\)
Quicksort (based on textbook)

```c
void Quicksort(Entry theArray[], int first, int last)
{
    if(first < last) {
        int pivotIndex = Partition(theArray, first, last);
        // quicksort first half
        Quicksort(theArray, first, pivotIndex - 1);
        // quicksort second half
        Quicksort(theArray, pivotIndex + 1, last);
    }
}
```

Note we do not include the pivot element in recursive calls to Quicksort!

Quicksort (based on textbook)

```c
int Partition(Entry theArray[], int first, int last)
{
    Entry pivot = theArray[first];    // pivot value
    int lastP1 = first;     // last index of first partition
    for(int i = first + 1; i <= last; i++) {
        if(theArray[i] < pivot) {
            lastP1++;
            Swap(theArray[i], theArray[lastP1]);
        }
    }
    Swap(theArray[first], theArray[lastP1]);
    return lastP1;
}
```
Quicksort

Partitioning step (where the work is done)

• **Worst case**
  – When pivot is smallest (largest) element
  – We end up with only one partition
  – This partition is of size \( n - 1 \)

• In general, the partition step has \( O(n) \) comparisons and swaps

• We do this partitioning \( n - 1 \) times in the worst case

• Therefore quicksort is \( O(n^2) \) in the worst case

Quicksort

Quicksort is **much better in practice**

– Each recursive step on avg. partitions array into equal halves
– So we have approx. \( \log_2 n \) recursion “levels” ... \( O(\log n) \)
– At each level we have \( O(n) \) comparisons and swaps
– So quicksort is \( O(n\log n) \) in the average (best) case

Note that compared to Mergesort …

– Quicksort does not have a “merge” step
– Each level has a length \( n-1 \) array
– Often makes quicksort fast in practice
– Also can spend some time picking a “good” pivot
**Comparison of Sorting Algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Average 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>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treesort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>