CSC148H Week 10

Ilir Dema, Michael Miljanovic

Summer 2021

Sorting Efficiency

How does the time to sort a list of n elements vary with n? It depends on the sorting algorithm that we use!

- Bubble sort: $O(n^2)$
- Selection sort: $O(n^2)$
- Insertion sort: $O(n^2)$
- Merge sort: ?
- Quick sort: ?
- Radix sort: ?

Merge Sort

- Merge sort works by
  - Recursively sorting the first half of the list
  - Recursively sorting the second half of the list
  - Merging the two halves into a newly sorted list
- The merging requires an auxiliary list (not required in quick sort)

Quick Sort

- Quick sort works by
  - Choosing a pivot value
  - Splitting (partitioning) the list into the part smaller than or equal to the pivot and the part greater than the pivot
  - Recursively sorting the first part of the list
  - Recursively sorting the second part of the list
  - Recombining the two parts into a single list

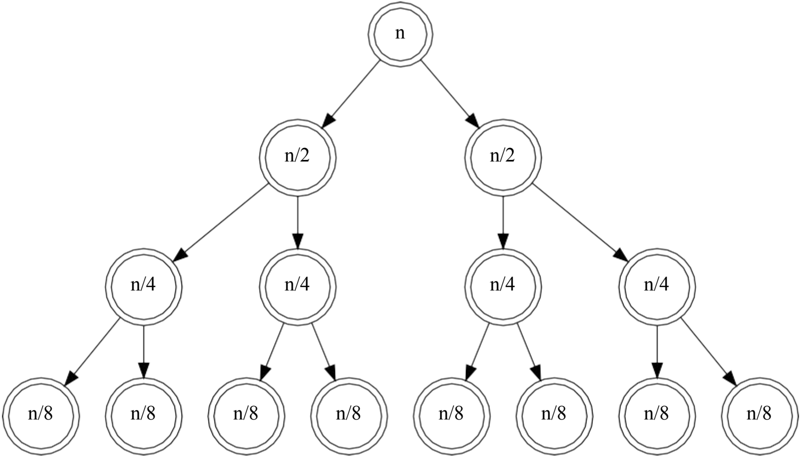
Worksheet 1

Worksheet 1, efficiency of mergesort and quicksort

Running Time of Merge Sort

- The list is divided in half on each recursive call
- From binary search, we know that we will recurse on the order of lg n times before we get to a base case
- On the outermost level, the merge step takes n time (one step per element in the list)
- On the two resulting subproblems of size $n/2$, the total merge time is $n/2 + n/2 = n$
- On the resulting four subproblems of size $n/4$ (two from each of the two $n/2$ subproblems), the total merge time is still n, and so on
- On each of lg n levels of recursion, we do a total of n work
- Thus, our running time is $O(n \lg n)$

Running Time of Mergesort: Tree



Best Case for Quicksort

- Assume that we choose a pivot that partitions the list exactly in half on each recursive call
- As before, we know that we will recurse on the order of lg n times before we get to a base case
- On the outermost level, the partition step takes n time (one step per element in the list)
- On the two resulting subproblems of size $n/2$, the total partition time is $n/2 + n/2 = n$
- On the resulting four subproblems of size $n/4$ (two from each of the two $n/2$ subproblems), the total partitioning time is still n, and so on
- On each of lg n levels of recursion, we do a total of n work
- Thus, our running time is $O(n \lg n)$

Worst Case for Quicksort

- It's always possible that we choose a pivot that partitions the list badly
- Consider: we pass a sorted list to quicksort and choose the leftmost element as the pivot
- On each recursive call operating on a list of size n, we partition it into a list of $n - 2$ elements and a "list" of just 1 element
- Now, we have n levels of recursion, whose partitions take $1 + 2 + . . . + n = O(n^2)$ time
- We have an $O(n^2)$ algorithm? Did we just waste a lot of time discussing this?

Worst Case for Quicksort...

- Instead of choosing the leftmost element for the pivot, we could choose the middle element
- This fixes the above case, but we can still construct a list to exhibit quicksort's worst-case behavior
- Another popular approach is choosing the median element among the first, middle, and last elements
- Even if there are lists that will take $O(n^2)$ time, this is rare in practice
- Consider: if we partition so that 90 percent of the elements are in one list and 10 percent are in the other, quicksort is still $O(n \lg n)$
  - The depth of recursion changes to $\log_{10/9} n$, but this is still logarithmic