CS 240 — Lecture 8

Bartosz Antczak *Instructor: Eric Schost* *January 26, 2017*

Bounds on Sorting

So far, our most efficient sorting algorithm has a worst-case running time of $\Theta(n \log n)$. Can we do better than this? Yes and no! It depends on our *comparison model*.

8.1 The Comparison Model

In the **comparison model**, data can only be accessed in two ways:

- Comparing two elements
- Moving elements around (i.e., copying, swapping)

All of the algorithms we've seen so far are in the comparison model.

8.1.1 Theorem 1

Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ *comparison operations*

Proof

- A correct algorithm takes different actions (i.e., moves, swaps, etc.) for each of the $n!$ possible permutations.
- An algorithm can be viewed as a *decision tree*, where each internal node is a comparison, and each leaf is a set of actions
- Each permutation must correspond to a leaf
- The worst-case number of comparisons is the longest path to a leaf
- Since the tree has at least $n!$ leaves, the height is at least ...

Therefore the worst-case number of comparisons in $\Omega(n \log n)$.

8.2 Non-comparison-based Sorting

8.2.1 Radix Sort

This algorithm sorts an array of numbers in base R (e.g., R is most commonly 2, 10, 128, or 256) by comparing each digit of the numbers rather than the whole number itself. Before we begin the algorithm, we want to ensure that every element has the same number of digits. We achieve this by finding the number with the largest digit, and adding the remaining leading 0s to every other element in the array (e.g., {3, 23, 4} becomes {03, 23, 04}).

```
// A: array of size n, containing m-digit numbers
// l, r, d: integers, 0 <= l, r <= n-1, 1 <=d <= m
RadixSort(A, l, r, d) {
  if (l < r) {
    // partition A[l...r] into bins according to d-th digit
    count-sort(A[l...r])
    if (d < m) {
      for (i = 0 to R-1) {
        (let li and ri be boundaries of i-th bin)
        RadixSort(A, li, ri, d+1)
      }
    }
  }
}
```

How do we sort the R bins? We do so by *counting*:

```
// A: array of size n containing numbers in {0, . . . , R     1}
count-sort(A) {
  // count how many numbers of each value there are
  C = array of size R, filled with zeros
  for i 0 0 to n-1 do {
    increment C[A[i]]
  }
  /* find left boundary for each kind (this is used to determine the starting index for the
     number of value i */
  I = array of size R, I[0] = 0
  for i = 1 to R-1 do {
    I[i] = I[i-1] + C[i-1]
  }
  // copy, then move back in sorted order
  B = copy(A)
  for i = 0 to n-1 do {
    A[I[B[i]]] = B[i]
    increment I[d]
  }
}
```

If the largest number of digits that an element in the array has is m, then the algorithm does n operations, m times (i.e., this algorithm has a runtime of $O(mn) = O(n)$).