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 computation 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)

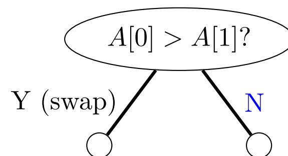
All of the algorithms we've seen so far are in the comparison model.

Theorem 1. *Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations*

Proof. An algorithm can be viewed as a *decision tree*, where each internal node is a comparison. For instance, the following algorithm sorts arrays of length 2:

```
Sort-2Array(A){
  if (A[0] > A[1]){
    swap(A[0], A[1])
  }
}
```

It can be represented by the following decision tree

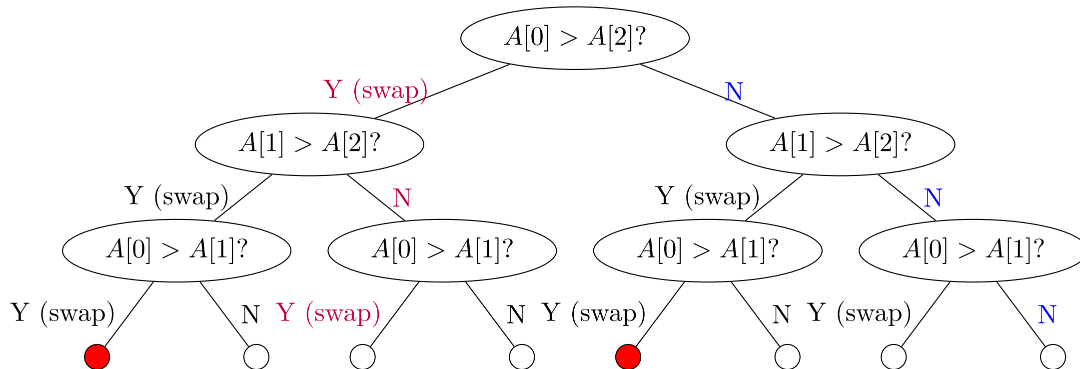


On input $[1, 2]$, we take the path N (in blue) from the root; on input $[2, 1]$, we take the path Y (black). In particular, the number of leaves (2) is the number of possible permutations of $[1, 2]$.

Now, consider the following algorithm for arrays of length 3.

```
Sort-3Array(A){
  if (A[0] > A[2]){
    swap(A[0], A[2])
  }
  if (A[1] > A[2]){
    swap(A[1], A[2])
  }
  if (A[0] > A[1]){
    swap(A[0], A[1])
  }
}
```

It can be represented by the following decision tree



On input $[1, 2, 3]$, we take the path NNN (in blue); on input $[3, 1, 2]$, we take the path YNY (in purple). Here the number of leaves is 8, which is not the number of inputs (6). This is because two leaves are not reachable (in solid red). Once we remove them (and all branches that lead to them), we are left with a tree with 6 leaves, which correspond to 6 possible inputs.

The construction generalizes to any input size; we build a tree with $n!$ leaves, that correspond to $n!$ permutations of $[1, \ldots, n]$. The height of this tree is the length of the longest path root to leaf, that is, the maximum number of comparisons the algorithm will do, over all $n!$ inputs.

To finish the proof, we will use the fact that in a binary tree of height h, with N nodes and M leaves, we have

$$h \geq \log_2(N) - 1 \geq \log_2(M) - 1.$$

The second inequality is easy (because $N \geq M$). For the first one, we saw (when we studied heaps) that in a binary tree of height h, the maximum number of nodes is $2^{h+1} - 1$. So $N \leq 2^{h+1} - 1 \leq 2^{h+1}$; taking logs, we get our inequality.

Back to our decision tree. Here, we have $M = n!$ leaves, so the height is at least $\log_2(n!) - 1$, which is $\Omega(n \log(n))$. □

8.2 Non-comparison-based Sorting: radix sort

8.2.1 Countsort

```
// 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]
  }
}
```

8.2.2 MSD 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*: 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 $O(mn) = O(n)$).