1 Quick Select

- Suppose we want to find a value in a sorted list, but we don’t care about other values – we just want to know where a certain value is in the sorted array/list (so, in other words, find the kth smallest value in the input).
- Instead of sorting the entire input, we can modify quick sort to do selection in $\Theta(n)$ (assuming good pivot choices).
- In quick sort, we recursed twice – once on each side of the pivot. Since we know in quick select which side the desired value is on, we don’t need to sort the other side. We simply perform one recursive call per partitioning instead of two, performing n lookups in total, resulting in a $\Theta(n)$ search.

2 Linear Time Sorting

- Comparison based sorts are all $\Omega(n \lg n)$. Why?
- For each comparison operation that we perform, we can make a binary decision. So we say that if we make $d$ decisions, we can have a total of $2^d$ possible actions.
- If there are $n$ items to be sorted, there are a total of $n!$ orderings of the values (permutations).
- Somehow, the $2^d$ actions must allow us to deal with all the $n!$ permutations, so $2^d \geq n!$.
- Therefore, $d \geq n \lg n$, and if each of the $d$ decisions takes constant time (a comparison), then we can see that comparison based sorting can never be better than $\Theta(n \lg n)$ for some inputs.
- What about sorting without doing comparisons?
- Instead of making binary decisions, we make n-ary decisions, thereby breaking up the sorting problem much more quickly.

3 Bucket Sort

- If the input value range is from 1 to $d$ (they need to be some integer-typed values), then we can create $d$ buckets, each of which is like a hashtable bucket.
- To sort, simply scan the input, placing each item into a bucket by its value (chained on the end of the linked-list in the bucket).
- To retrieve the values in sorted order, scan the $d$ buckets, concatenating the items from all the buckets.
- Takes $O(d + n)$ time for $n$ items. If $d$ is $O(n)$, then bucket sort runs in $O(n)$.
- Provides stable sorting.

4 Announcements

- Reading: Data Structures (Into Java) – 8.9-8.12; Goodrich & Tamassia – 10.4-10.7
5 Radix Sort

- Instead of sorting based on the entire value’s range at once, we break the inputs into their digits or characters (or other sub-sortable pieces), bucket sort on those, and then bucket sort each subsequent position based on the previous.

- Suppose we break the input keys into d buckets, like with bucket sort, but d is based on the size of a digit (or bit, or character, ...).

- Suppose we are sorting Strings of length 3. We perform radix sort by using the last character as the piece of input we divide the input on – we have 26 buckets.

- After sorting the least significant, we use the ordering we got from the first stage for the second stage, in which we sort based on the next to last character, just as we did before.

- After all characters have been bucket sorted like this, since each bucket sort is stable, we are guaranteed a fully sorted result, taking \( O(K \times n) \), where \( K \) is the size of each key.

- Since we’re taking the size of each key into account for radix sort, we should revise our running times of comparison based sorts – if we have \( n \) different values, then each key must have \( K = \lg n \) bits. To compare them, we must examine \( \Theta(K) \) bits, so each comparison really is in \( \Theta(K) \). Therefore, the best comparison based sorts really are in \( \Theta(n \ (\lg n)^2) \), whereas radix sort (which looks at the \( K \) bits per key only once in total), is in \( \Theta(n \ lg \ n) \).

6 Question

Given the following values showing the steps of sorting, determine the sorting algorithm used:

```
7 4 2 8 1 0 6 3
4 2 8 0 6 7 1 3
4 8 0 1 2 6 7 3
8 0 1 2 3 4 6 7
0 1 2 3 4 6 7 8
```

White: Insertion Sort

Blue: Selection Sort

Yellow: Shell’s Sort

Green: Quick Sort

Pink: Radix Sort

Pink is correct – Radix Sort, as we’ll see in the next question.

7 Question 2

Given the values, determine how radix sort is sorting the input:

```
7 4 2 8 1 0 6 3
4 2 8 0 6 7 1 3
4 8 0 1 2 6 7 3
8 0 1 2 3 4 6 7
0 1 2 3 4 6 7 8
```

White: Insertion Sort

Blue: Selection Sort

Yellow: Shell’s Sort

Green: Quick Sort

Pink: Radix Sort

Pink is correct – Radix Sort, as we’ll see in the next question.
7 4 2 8 1 0 6 3
4 2 8 0 6 7 1 3
4 8 0 1 2 6 7 3
8 0 1 2 3 4 6 7
0 1 2 3 4 6 7 8

White: Radix is the digit
Blue: Radix is the binary digit
Yellow: Radix is the hexadecimal digit
Green: Radix is the ASCII value of the digit
Pink: Radix is the digit mod 2

Blue – We use each bit as the radix – the value by which we bucket sort at each step.