More on Sorting

Ref: Chapter 8 of text.

- Insertion-Sort, Heapsort, Quicksort, etc: *Comparison-based*.
- Fastest running time: $O(n \log n)$.
- Is there a comparison-based sorting algorithm with a running time of $o(n \log n)$?
- Answer: No. Every comparison-based algorithm must use $\Omega(n \log n)$ comparisons in the worst-case.

Assumptions:

- Input values are distinct.
- All comparisons are of the form “$a_i \leq a_j$”.
- Control and data movement operations of an algorithm are ignored.

Decision Tree Model:

- A binary tree to represent the comparisons performed by a sorting algorithm.
- Goal: To obtain an asymptotic *lower bound* on the number of comparisons used by the algorithm.
- Each internal node labeled by a *comparison* with two possible outcomes.
- Each leaf node labeled by a *permutation* of $1, 2, \ldots, n$.

![](image)

Example: To be presented in class.
In general,

- Any comparison-based algorithm (decision tree) that correctly sorts $n$ inputs must have $n!$ leaves.
- Worst-case number of comparisons used = Height of the decision tree.

**Theorem 1:** Any decision tree that correctly sorts $n$ inputs has height $\Omega(n \log n)$.

**Proof:** To be presented in class.

**Notes:**

- Generalization due to M. Ben-Or (1984): $\Omega(n \log n)$ comparisons are necessary even when each comparison involves a (fixed degree) polynomial function of the input values.
- M. Fredman and D. Willard (1990): Using comparisons and some arithmetic operations, sorting can be done in $O(n \log n / \log \log n)$ time.

**Sorting without comparisons of keys:**

- Algorithms with running time of $O(n)$.
- Impose restrictions on keys.
- $\Omega(n \log n)$ lower bound does not apply.

**Digression: stable sorting**

- A stable sorting algorithm keeps equal keys in the same order as they appear in the input.
- Useful when sorting on keys consisting of two or more attributes.
- Crucial for correctness of some sorting algorithms such as Radix-Sort.

**Counting Sort:**

- Assumption: Each key is an integer in the range 1 through $k$ for some integer $k$.
- Used when $k = O(n)$, where $n$ is the number of values to be sorted.
- Not an in-place sort (needs auxiliary arrays).
Basic idea: (for sorting into increasing order)
- For a key $x$, find the number, say $t$, of keys that are less than $x$.
- If all keys are distinct, then $x$ must be in position $t + 1$ in the sorted order.
- Modification is needed when keys are not distinct.


Exercise: Give an example to point out that if the loop in Step 4 of Counting-Sort runs from 1 to $n$, the resulting algorithm is not a stable sort.

Running time:
- Steps 1 and 3: $O(k)$.
- Steps 2 and 4: $O(n)$.
- Overall: $O(n + k)$.
- If $k = O(n)$, then running time = $O(n)$.

Radix-Sort:
- Assumption: Each key is an integer with $d$ digits (in some fixed radix).
- Basic idea: Use a stable sort on each of the $d$ digits going from the least significant to the most significant digit.


Notes:
1. Radix-Sort won’t work correctly if the order of sorting is from the most significant digit to the least significant digit. (Try sorting the list 28, 27, 35.)
2. Radix-Sort won’t work correctly if the sorting algorithm used for each digit is not a stable one. (Try sorting the list 94, 87, 89.)

Correctness: To be discussed in class.
Running time: (assuming decimal numbers)

- Since each digit is in the range 0 through 9, time to sort by each digit = \( O(n) \) (using Counting-Sort).
- So, overall time: \( O(dn) \).
- If \( d \) is \( O(1) \), then running time = \( O(n) \).
- A trade-off is possible between key size and running time.

Bucket-Sort:

- Assumes that each input value is in the interval \([0, 1)\).
- Expected running time: \( O(n) \).

Basic idea:

- Input array \( A[1 .. n] \) (with \( 0 \leq A[i] < 1 \) for \( 1 \leq i \leq n \)).
- Divide the interval \([0, 1)\) into \( n \) equal intervals each of size \( 1/n \).

- Use an array \( B[0 .. n - 1] \) of pointers (buckets). The list pointed to by \( B[j] \) contains all the numbers in the interval \([j/n, (j + 1)/n)\).
- Distribute the elements of \( A \) into the buckets. (Value \( A[i] \) gets into the bucket \( [nA[i]] \)).
- Sort each of the lists.
- Concatenate the sorted lists in order.


Running time:

- Worst-case: If Insertion-Sort is used for each list, the worst-case time is \( O(n^2) \). (Exercise)
- Average-case: If the items are drawn from a uniform distribution over \([0, 1)\), the expected running time is \( O(n) \). (To be discussed in class.)