Recitation 7

- Sorting Stability
- Counting Sort
- Radix Sort

Sort Stability

A sorting algorithm is **stable** if elements with the same key appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important when satellite data are carried around with the element being sorted. For example, in order for radix sort to work correctly, the digit sorts must be stable.

A good example of when stable sorting is useful can be found in Pset 1. In Pset 1 we had to break ties alphabetically on the 50 most frequent words. One way of doing this was by first sorting alphabetically. Then, sorting again by frequency using a stable sort. Since we already sorted alphabetically and stable sorting breaks ties based on whichever element appears first in the input array, we would break ties in the stable sort alphabetically.

Is Heap Sort Stable?

No. An example of heap sorting \{3, 3, 3, 2, 3\} for max heaps can illustrate the point. Can also use \{2, 2, 2, 3, 2\} for min heaps.

Is Merge Sort Stable?

Merge sort can be stable as long as the merge operation is implemented properly. Specifically, on a tie, merge sort must not change the order of the two items. Furthermore, when merging, merge sort must merge the items from the subarray that had lower indices first on ties.

Counting Sort

Counting sort is an algorithm that takes an array \(A\) of \(n\) elements with keys in the range \{1, 2, ..., \(k\)\} and sorts the array in \(O(n + k)\) time. It is a stable sort.

**Algorithm Outline:** Count key occurrences using an auxiliary array \(C\) with \(k\) elements, all initialized to 0. We make one pass through the input array \(A\), and for each element \(i\) in \(A\) that we see, we increment \(C[i]\) by 1. After we iterate through the \(n\) elements of \(A\) and update \(C\), the value at index \(j\) of \(C\) corresponds to how many times \(j\) appeared in \(A\). This step takes \(O(n)\) time to iterate through \(A\).
We can continue from here and transform $C$ to an array where $C[j]$ refers to how many elements are $\leq j$. We do this by iterating through $C$ and adding the value at the previous index to the value at the current index, since the number of elements $\leq j$ is equal to the number of elements $\leq j - 1$ (i.e. the value at the previous index) plus the number of elements $= j$ (i.e. the value at the current index). The final result is an array $C$ where the value of $C[j]$ is the number of elements $\leq j$ in $A$.

Now we iterate through $A$ backwards starting from the last element of $A$. For each element $i$ we see, we check $C[i]$ to find out how many elements are there $\leq i$. From this information, we know exactly where we can put $i$ in the sorted array. Once we insert $i$ into the sorted array, we decrement $C[i]$ so that if we see a duplicate element, we know that we have to insert it right before the previous $i$. Once we finish iterating through $A$, we will get a sorted list as before. This time, we provided a mapping from each element $A$ to the sorted list. Note that since we iterated through $A$ backwards and decrement $C[i]$ every time we see $i$, we preserve the order of duplicates in $A$. That is, if there are two 3s in $A$, we map the first 3 to an index before the second 3. This now makes counting sort stable. We will need the stability of counting sort when we use radix sort.

Iterating through $C$ to change $C[j]$ from being the number of times $j$ is found in $A$ to being the number of times an element $\leq j$ is found in $A$ takes $O(k)$ time. Iterating through $A$ to map the elements of $A$ to the sorted list takes $O(n)$ time. Since filling up $C$ to begin with also took $O(n)$ time, the total runtime of this stable version of counting sort is $O(n + k + n) = O(2n + k) = O(n + k)$. 

We: 4  1  3  4  3  
C: 1  0  2  2  

A: 4  1  3  4  3  
B: 1  3  3  4  4  
C': 1  1  3  5  

Radix Sort

Radix sort is a digit-by-digit sorting algorithm that works even for large integers. Radix sort is built on top of counting sort.

- imagine each integer in base $b$  
  $\implies d = \log_b k$ digits $\in \{0, 1, \ldots, b-1\}$.

- sort (all $n$ items) by least significant digit $\rightarrow$ can extract in $O(1)$ time

- sort by most significant digit $\rightarrow$ can extract in $O(1)$ time
  sort must be stable: preserve relative order of items with the same key  
  $\implies$ don’t mess up previous sorting

  For example:

  \[
  \begin{array}{cccc}
  3 & 2 & 9 & 7 \\
  4 & 5 & 7 & 3 \\
  6 & 5 & 7 & 4 \\
  8 & 3 & 9 & 4 \\
  7 & 2 & 0 & 3 \\
  3 & 5 & 5 & 8 \\
  \end{array}
  \quad \rightarrow \quad \begin{array}{cccc}
  7 & 2 & 0 & 7 \\
  4 & 5 & 7 & 3 \\
  6 & 5 & 7 & 4 \\
  8 & 3 & 9 & 4 \\
  7 & 2 & 0 & 3 \\
  8 & 3 & 9 & 8 \\
  \end{array}
  \]

  sort sorted sorted sorted

- use counting sort for digit sort
  - $\implies \Theta(n + b)$ per digit
  - $\implies \Theta((n + b)d) = \Theta((n + b)\log_b k)$ total time
  - minimized when $b = n$
  - $\implies \Theta(n \log_n k)$
  - $= O(nc)$ if $k \leq n^c$

Lower Bounds on Search and Sort

In this section, we will show

- searching among $n$ preprocessed items requires $\Omega(\lg n)$ time
• sorting \( n \) items requires \( \Omega(n \log n) \)

These statements will be true given that we’re using the comparison model. Later we will show that if we can escape the comparison model, we can achieve faster sorting.

**Comparison Model of Computation**

In the comparison model, we have:

- input items can be anything, as long as they support comparisons (e.g. \(<, >, \leq, \text{ etc.}\))
- time cost = # comparisons

Algorithms like binary search and merge sort fall under the comparison model, because they both rely on making comparisons between the items in a list.

**Decision Trees**

Any comparison algorithm can be viewed/specified as a tree of all possible comparison outcomes & resulting output, for a particular input size \( n \):

- example, binary search for \( n = 3 \):
• internal node = binary decision
• leaf = output (algorithm is done)
• root-to-leaf path = algorithm execution
• path length (depth) = running time
• height of tree = worst-case running time

Search Lower Bound

Because there are \( n + 1 \) possible outcomes for Search (the outcome is either the index of the element or maybe \(-1\) if the element is not in the list), the decision tree for Search must have at least \( n + 1 \) leaves. The height of a tree with \( n \) leaves is \( \Theta(\lg n) \), and because the height of the tree corresponds to the running time, we have a lower bound for search of \( \Omega(\lg n) \).

\[
\text{# leaves} \geq \text{# possible answers} \geq n \quad \text{(at least 1 per } A[i] \text{)}
\]

• decision tree is binary

\[
\implies \text{height} \geq \lg \Theta(n) = \lg n + \Theta(1) \quad \text{lg } \Theta(1)
\]

Sorting Lower Bound

• all \( n! \) are possible answers

\[
\text{# leaves} \geq n!
\]

\[
\implies \text{height} \geq \lg n!
\]
\[
= \lg(1 \cdot 2 \cdots (n-1) \cdot n)
\]
\[
= \lg 1 + \lg 2 + \cdots + \lg(n-1) + \lg n
\]
\[
= \sum_{i=1}^{n} \lg i
\]
\[
\geq \sum_{i=n/2}^{n} \lg i
\]
\[
\geq \sum_{i=n/2}^{n} \frac{\lg n}{2} = \frac{n}{2} \lg n - \frac{n}{2} = \Omega(n \lg n)
\]
• in fact $\lg n! = n \lg n - O(n)$ via Stirling’s Formula:

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \implies \lg n! \sim n \lg n - (\lg e)n + \frac{1}{2} \lg n + \frac{1}{2} \lg(2\pi)$$