CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 11: October 4, 2016

Integer (or Radix) Sorting

- In the last lecture we showed that we can sort n integers in the range $[1, m]$ in $O(m+n)$ time.
- In this lecture we will show that we can sort n integers in the range $[0, n^c - 1]$ in $O(n)$ time, for any constant c. The case of $c = 2$ is very common in graph algorithms and computational geometry algorithms.
- To prove this result, we will use bucket sorting and the idea of radix sorting.
- The idea of radix sorting is to sort the given keys with respect to some number of bits at a time.
- Consider the problem of sorting the following two digit numbers:
  $45, 17, 56, 43, 26, 35, 49, 22, 15, 52, 12.$
  One way of sorting these numbers is to first sort them with respect to their least significant digits (LSDs); and then sort them with respect to their most significant digitis (MSDs).
- When we sort these keys with respect to their LSDs we get:
  $22, 52, 12, 43, 45, 15, 56, 26, 17, 49.$
  When we now sort them with respect to their MSDs we get:
  $12, 15, 17, 22, 26, 43, 45, 49, 52, 56,$
  which is the correct sorted sequence!
- Notice that in the second phase of sorting, we are only required to sort the numbers with respect to their MSDs (without paying attention to the LSDs). For the keys whose MSD is 1, a perfectly valid ordering could be $15, 17, 12$!
- Thus we realize that the idea of radix sorting works if we do not disturb any ordering that might have been introduced by previous phases of sorting among equal keys. We can formalize this notion as follows:
- **Stability:** We say a sorting algorithm is *stable* if equal keys will remain in the same relative order in the output as they were in the input.

- Moral: The idea of radix sorting works if we use a stable sorting algorithm in every phase of sorting (except for the first phase).
- We can make the bucket sort algorithm stable by always inserting keys at the tails of the lists and outputting any list starting from the head.

The Sorting Algorithm

- Let $X = k_1, k_2, \ldots, k_n$ be the input sequence. Each key is an integer in the range $[0, n^c - 1]$.
- The binary representation of each key will be a binary string with $\leq c \log n$ bits. (Recall that any integer m can be represented in binary with $\lceil \log(m+1) \rceil$ bits.)
- Assume without loss of generality that each key has exactly $c \log n$ bits. We partition the $c \log n$ bits in each key into c parts. The least significant $\log n$ bits will constitute Part 1, the next least significant $\log n$ bits will constitute Part 2, $\cdots$, the most significant $\log n$ bits will constitute Part c.
- The steps in the algorithm are:

  for $i = 1$ **to** c **do**
  Stably sort the keys with respect to their ith parts using bucket sort;

- **Analysis:** In any phase of sorting we are sorting n $\log n$-bit integers. Note that the maximum value of any integer with $\log n$ bits is $n-1$. (In general, the maximum value of any m-bit integer is $2^m - 1$.) In other words, in each phase of integer sorting, we have to sort n integers in the range $[0, n-1]$. If we use bucket sort, this will take $O(n)$ time. Since we only have a constant number of phases, the total run time is $O(n)$. As a result, we get the following:

  Theorem: We can sort n integers in the range $[0, n^c - 1]$ in $O(n)$ time, c being any constant. □
- An interesting question is if we can do radix sorting starting from the MSBs.
- Consider the problem of sorting the following two digit numbers:
  $62, 15, 45, 25, 38, 27, 69, 31, 21, 72$.
  If we sort them with respect to their MSDs we get:
  $15, 25, 27, 21, 38, 31, 45, 62, 69, 72$.
  If we now sort them with respect to their LSDs, we do not get a sorted sequence!

- However, after the first sorting phase, we can have separate buckets, each bucket having equal values (for the MSD). In our example, the buckets will be:
  $\{15\}, \{25, 27, 21\}, \{38, 31\}, \{45\}, \{62, 69\}, \{72\}$.
  In the second phase we sort the buckets separately (sorting the keys in any bucket with respect to their LSDs).

Uniqueness of keys

- In our prior discussions on sorting algorithms we have assumed that the keys are distinct. In practice this assumption may not hold.
- When there are repetitions we can make the keys distinct by adding an additional $\log n$ bits to each key as follows.
- If $k_1, k_2, \ldots, k_n$ is the input sequence, replace this sequence with $(k_1, 1), (k_2, 2), \ldots, (k_n, n)$.
- Each key has now become a pair. We can define a linear order among pairs as follows. $(a, b) < (c, d)$ if either $\{a < c\}$ or $\{a = c$ and $b < d\}$. This ordering is called *lexicographic ordering*.
- Note that the pairs themselves are distinct and each key has an additional $\log n$ bits. Sorting $\log n$-bit integers takes only $O(n)$ time.

Selection

- The problem of selection takes as input a sequence $X = k_1, k_2, \ldots, k_n$ (where each key is an arbitrary real number) and an integer i (with $1 \leq i \leq n$). The problem is to output the ith smallest key of X.
- **Some special cases:** When $i = 1$, we are interested in finding the smallest key of X. This can be done in $O(n)$ time. When $i = n$, we are interested in identifying the larget element of X. This also can be done in $O(n)$ time. When $i = \frac{n}{2}$, we are looking for the *median* of X. It turns out that from out of all possible values of i, the case of $i = \frac{n}{2}$ is the most difficult to solve.

Quick Select

- We can devise an algorithm for selection that is similar to quick sort. This algorithm is called quick select and works as follows:

```
QuickSelecr(X, i)
    if |X| = 1, output k_1 and quit;
```

Pick a pivot k from X;
Partition X into X_1 and X_2 where
$X_1 = \{q \in X : q < k\}$ and $X_2 = \{q \in X : q > k\}$.
if $|X_1| = i - 1$ **then** output k and quit;
if $|X_1| \geq i$ **then** QuickSelect(X_1, i);
else QuickSelect$(X_2, i - |X_1| - 1)$;

- Let $T(n)$ be the run time of this algorithm on any input of size n. Then we have:

$$T(n) = n + \max\{T(|X_1|), T(|X_2|)\}.$$

- One of the worst cases happens when one of the parts is empty on each recursive call. In this case, $T(n) = n + T(n-1)$ which solves to: $T(n) = \Theta(n^2)$.
- One of the best cases is when both X_1 and X_2 are of nearly the same size. In this case: $T(n) = n + T(n/2) = \Theta(n)$.
- We can also show that the average run time of quick select is $O(n)$.

A worst case linear time algorithm

- Blum, Floyd, Pratt, Rivest, and Tarjan (1973) have given a worst case linear time algorithm for selection. We refer to this as the BFPRT algorithm.
- BFPRT algorithm is the same as quick select except that it uses a special procedure to choose the pivot element.
- If $X = k_1, k_2, \ldots, k_n$ is the input sequence, the algorithm groups the elements of X into groups of size 5 each. Let these groups be $G_1, G_2, \ldots, G_{n/5}$. The median of each of these groups is found. Let these medians be $M_1, M_2, \ldots, M_{n/5}$, respectively. The median M of these medians is found recursively. M is used as the pivot and the quick select algorithm is run from thereon.
- More details will be given in the next lecture.