Lecture 23: Priority Queues, Part 2
10:00 AM, Mar 19, 2018

Contents

1 Sorting, Revisited 1
2 Counting Sort 1
3 Bucket Sort 2
4 Radix Sort 6
   4.1 MSD Radix Sort 7
   4.2 LSD Radix Sort 8
   4.3 Radix Sort Analyses 9

Objectives

By the end of this lecture, you will know:

- just how good a comparison-based sorting algorithm can do in the worst case; and
- more sorting algorithms (specifically, counting sort, bucket sort, and radix sort)

1 Sorting, Revisited

By this point in CS 17/18, we have discussed many sorting algorithms, including heap sort, bubble sort, insertion sort, selection sort, quicksort... These algorithms are all examples of comparison sorting algorithms, because they incorporate a sequence of comparisons between the items in the list to be sorted. There is one key fact common to all comparison sorting algorithms: they require at least $\Omega(n \log n)$ comparisons in the worst case, where $n$ is the number of items to be sorted.

Let’s try to understand why this is the case. Suppose we have three numbers, $x$, $y$, and $z$. In order to rearrange them in sorted order, we might begin by comparing $x$ and $y$. Depending on their relationship, we could then compare $x$ and $z$ or $y$ and $z$. Now depending on their relationship, we might have determined the sorted order, or we might have to do a further comparison. The point is: we can draw these pairwise-comparisons as a decision tree, and that tree will have $n!$ leaves (the total number of permutations), where $n$ is the size of the given list.

Now a tree of depth $d$ has no more than $2^d$ leaves; equivalently, a tree with $x$ leaves has depth at least $\log x$. Therefore, the depth of our decision tree, which has $n!$ leaves, is at least $\log(n!)$, making this the necessary number of comparisons necessary in the worst-case. While it is relatively
straightforward to see that \( \log(n!) \) is \( O(n \log n) \) (because \( n! \leq n^n \), so \( \log n! \leq \log(n^n) = n \log n \)), it is also the case that \( \log(n!) \) is \( \Omega(n \log n) \), so that \( \log(n!) \) is in fact \( \Theta(n \log n) \)\(^1\).

To get around this lower bound, we are now going to discuss two other means of sorting, which are not based on comparisons, but are only applicable in certain special cases. These approaches obtain speedups by exploiting the structure of the data to be sorted.

### 2 Counting Sort

The simplest non-comparison based sorting algorithm is called *counting sort*. Consider a list of \( n \) 0s and 1s, stored as an array. We could use the heavyweight machinery of mergesort to sort this list, but we can also just count the number of 0s in the array (say \( k \)), and then set the first \( k \) entries of the array to 0, and the remaining \( n - k \) entries of the array to 1. Voila! The array is sorted, and in just \( O(n + n) = O(n) \) time. (We traverse the array once to find the 0s and a second time to reset its values.)

This idea generalizes to sorting integers in a small range, say from 1 to \( m \). We traverse the given array (say \( A \)) once, counting how many times 1 occurs, how many times 2 occurs, etc., up to \( m \). We store the number of times integer \( i \) occurs in \( A \) in the \( i \)th index of another array, say \( C \). We then create a sorted array, \( B \), by setting its first \( C[0] \) entries to the value 0, the next \( C[1] \) entries to the value 1, and so on. (Note that the sum of the values stored in \( C \) is the size of \( A \) and \( B \).) This generalized counting sort runs in time \( O(n + m) \). Furthermore, if \( m \in O(n) \), then this sort in its entirety is \( O(n) \).

Here is an example: suppose the array \( A \) consists of the integers 2, 3, 2, 1, 2, in that order. Then \( C[1] = 1, C[2] = 3, \) and \( C[3] = 1 \). So, looping through the arrays \( C \) and \( A \) (more or less simultaneously), we store one 1 in \( B \), followed by three 2’s, followed by one 3, at which point the array \( B \) contains all the items in \( A \), sorted.

You might wonder whether it is necessary to create a brand new array \( B \), or if we can just overwrite the contents of the existing array \( A \). No, it is not necessary to create a new array. Once we finish traversing \( A, C \) contains all the relevant information in \( A \) needed to sort it. So, we can overwrite the original contents of \( A \) in sorted order. Algorithms that overwrite their original data are called *in-place* algorithms.

Next, we study a generalization of counting sort, called *bucket sort*, in which case we store, instead of the counts, the values themselves, in so-called buckets (implemented as linked lists).

### 3 Bucket Sort

*Bucket sort*, in its simplest form, is a variant of counting sort that sorts integers in the range \([0, m]\). The auxiliary data structure it uses is an array of linked lists. We call these linked lists *buckets*. Bucket sort takes \( O(m + n) \) time and space.

Here is the algorithm:

- **Step 1**: Initialize an array of \( m + 1 \) empty buckets (i.e., linked lists).

\(^1\)See, for example, [http://planetmath.org/asymptoticboundsforfactorial](http://planetmath.org/asymptoticboundsforfactorial)
• **Step 2**: Insert each occurrence of the integer \( i \) into bucket \( i \).

• **Step 3**: Combine the entries in all the buckets by iterating through the linked lists in order and storing the values in order.

As an example, let’s sort the list of integers 2, 3, 2, 1, 2. We start with four empty buckets:

First, we insert 2.

Next, we insert 3.

Then, we insert 2 again.

Next, we insert 1.
And then, we insert 2 yet again.

Finally, we extract values in each bucket in order, to obtain the sorted list 1, 2, 2, 2, 3.

**Space Complexity** Let’s consider the space required to sort \( n \) integers with values ranging from 1 to \( m \). We’ll need an array of \( m \) elements, so that is \( O(m) \) space. We also need \( m \) lists, and distributed among these lists will be \( O(n) \) integers.

Here’s a graphical representation of what’s going on:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The four rows in this table represent the four buckets. The columns represent the integers we inserted into the buckets in the order we inserted them. There’s a 1 in row \( i \), column \( j \) if the \( j \)th item was inserted into bucket \( i \).

Now a naive analysis would say: there can be as many as \( n \) 1s in a given row, and since there are \( m \) rows, we can expect at most \( O(mn) \) 1s in total. But we are smarter than that. Notice that there can only ever be one 1 in any given column! That means that the total number of 1s in the table is never more than \( n \). In other words, the total amount of space required by all the lists is \( O(n) \).

Therefore, the amount of space required overall is \( O(m + n) \).

**Time Complexity** Now let’s calculate the run time of bucket sort, analyzing each step of the algorithm in turn:

- **Step 1**: The run time of initializing each bucket is \( O(1) \), and there are \( O(m) \) buckets, so the time required for this step is \( O(1) \times O(m) = O(m) \).
• **Step 2:** The run time per integer is $O(1)$ because all we do in this step is index into an array and insert into a list. There are $n$ integers, so the time required for this step is $O(1) \times n = O(n)$.

• **Step 3:** The naive analysis:
  
  – There are $m$ buckets.
  – Run time per bucket is proportional to the number of integers in the bucket.
  – There are at most $n$ integers in any bucket.
  – Total run time is $O(n) \times m = O(mn)$.

  But, once again, we are smarter than that. There are only $n$ items in total, so the total time taken for this step is at most $O(n)$.

In sum, the total time taken for steps 1 and 2 is $O(m + n)$, and the total time taken for step 3 is at most $O(n)$. Hence, the total run time for bucket sort is $O(m + n)$.

These analyses were *aggregate* in the sense that we computed the time and space complexities for a sequence of operations “in the aggregate,” as opposed to tallying the individual complexities of the operations in the sequence.

**Bucket Sort In Practice**  In the previous example, we created one bucket per integer. But when bucket sort is used in practice it is because the length of the list to be sorted is massive, so much so that it makes sense to break down a large sorting problem into many smaller ones.

More specifically, if we are sorting many many integers in the range $[0, m]$, we would divide them up into $k$ buckets, each of size $(m + 1)/k$. We would then sort each non-empty bucket using our favorite sorting algorithm (on linked lists). Finally, by combining the contents of each bucket in order, we would obtain a sorted list.

Here is an example:

• **Step 1:** Initialize an array of $k$ empty buckets (i.e., linked lists), each of size $m/k$.

• **Step 2:** Insert each occurrence of the integer $i$ into the appropriate bucket.

• **Step 3:** Sort each non-empty bucket using your favorite sorting method.

• **Step 4:** Combine the entries in all the buckets by iterating through the linked lists in order and storing the values in order.

As an example, let’s sort the list of integers 34, 5, 14, 19, 30, 37. We start with four empty buckets, and ranges 0-9, 10-19, 20-29, and 30-39.

<table>
<thead>
<tr>
<th>0−9</th>
<th>10 − 19</th>
<th>20 − 29</th>
<th>30 − 39</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Insert 34.
Insert 5.

Insert 14.

Insert 19.

Insert 30.

Insert 37.

Next, we sort each bucket.

Finally, we extract the values in each bucket in order to obtain the sorted list 5, 14, 19, 30, 34, 37.
4 Radix Sort

What if you want to sort very big integers? Well, there’s a nice trick. Do you remember learning about radix notation in grade school? You might not remember the name, but something like this should look familiar:

\[ 231 = 2 \cdot 10^2 + 3 \cdot 10^1 + 1 \cdot 10^0 \]

Indeed, you can express any number in base 10 this way. More generally, you can express any natural number this way, given any base \( m \), using symbols in the range \([0, m - 1]\):

\[ a_2 \cdot m^2 + a_1 \cdot m^1 + a_0 \cdot m^0 \]

If we are given a list of numbers, each with \( k \) digits, we can “bucket” each number \( k \) times, by each of its \( k \) digits. But where should we start? Should we move from left to right, or from right to left. Perhaps surprisingly, both strategies work!

Moving from left to right (i.e., bucketing numbers from their Most Significant Digit to their least significant digit) leads to an algorithm known as MSD radix sort. Moving from right to left (i.e., bucketing numbers from their Least Significant Digit to their most significant digit) leads to an algorithm known as LSD radix sort. We will explore each of these two schemes in turn.

4.1 MSD Radix Sort

We illustrate MSD radix sort by example, and then we outline the steps in the algorithm, and analyze its run time.

Let’s say that we would like to sort the list of natural numbers \( \langle 329, 457, 657, 839, 456, 720, 355 \rangle \).\(^2\) MSD radix sort begins by sorting the list by the most significant digit (the leftmost digit), so that 329 and 355 are put in one bucket (a 3’s bucket), and 457 and 456 are put in another (a 4’s bucket). All other numbers (657, 839, and 720) are put in their own buckets (6’s, 8’s, and 7’s buckets, respectively). Next, the 3’s bucket is sorted according to the second most significant digit, after which 329 is in a 32’s bucket, and 355 is in a 35’s bucket. Then, the 4’s bucket is sorted according to its second most significant digit, which puts both 457 and 456 in a 45’s bucket. All other buckets are also sorted, but since they are of size 1, these sorts effectively do nothing (657 is placed in 65’s bucket, 720 is placed in 72’s bucket and 839 is placed in 83’s bucket). The 45’s bucket is then sorted by the rightmost digit, so that 456 ends up in a 456’s bucket, and 457 in a 457’s bucket.

Now that all the numbers have been bucketed and re-bucketed \( k \) times, it is time to concatenate buckets. First 456 and 457 are concatenated to form the list \( \langle 456, 457 \rangle \); at the same time, 329 and 355 are concatenated into the list \( \langle 329, 355 \rangle \). These two lists are concatenated together, and with the singleton lists containing 657, 720, and 839, which yields the final sorted list.

\(^2\)This example is borrowed from the standard algorithms textbook: *Introduction to Algorithms*, by Cormen, Leiserson, Rivest, and Stein (2001).
Here are steps performed by MSD radix sort:

- **Step 1**: Bucket the numbers based on their most significant digit.

- **Step 2**: Sort each of the resulting buckets by calling MSD radix sort recursively, with each recursive call bucketing the numbers by their next most significant digit.

- **Step 3**: When there are no more digits by which to bucket, combine buckets, and return the sorted list.

We defer analyzing MSD radix sort until after we both present and analyze LSD radix sort.

### 4.2 LSD Radix Sort

Like MSD radix sort, we illustrate LSD radix sort by example, and then we outline the steps in the algorithm. Our example is again borrowed from the standard algorithms textbook.\(^3\)

Unlike MSD radix sort, which is a recursive algorithm, LSD radix sort is best understood as an iterative algorithm. In the figures that follow, the arrows below the numbers point to the digit that will be sorted in the next iteration, while the bracketed digits were already sorted in past iterations.

In this example, the numbers have three digits; hence, they will be sorted in three iterations.

The first step is to bucket by the least significant digit, and then recombine the buckets:

---

\(^3\) *Introduction to Algorithms*, by Cormen, Leiserson, Rivest, and Stein (2001).
The second step is the same: bucket by the second-most least significant digit, and then recombine the buckets:

Third, bucket by the third-most least significant digit, and then recombine the buckets:

As illustrated by this example, LSD radix sort can be described as follows:

- For each digit, from least to most significant:
  - Bucket the given list of numbers based on the relevant digit.
  - Combine the buckets into a single list sorted according to that digit.
Correctness  MSD radix sort (i.e., sorting from left to right) is intuitive. LSD radix sort (i.e., sorting from right to left) is less intuitive. Why does it work? The key is stability:

- At the end of stage 0, the elements are sorted by digit 0.
- At the end of stage 1, the elements are sorted by digit 1. If two elements have the same digit 1, they remain sorted by digit 0.
- At the end of stage 2, the elements are sorted by digit 2. If two elements have the same digit 2, they remain sorted by digit 1. If they have the same digit 1, they remain sorted by digit 0.
- At the end of the last stage, \( n \), the elements are sorted by digit \( n \). If two elements have the same digit \( n \), they remain sorted by digit \( n - 1 \). If they have the same digit \( n - 1 \), they remain sorted by digit \( n - 2 \), and so on.

4.3 Radix Sort Analyses

We analyze MSD and LSD radix sort in the aggregate, like we did bucket sort. Assume we are given a list of \( n \) natural numbers, in which the greatest has \( d \) digits.

LSD Radix Sort  In LSD radix sort, there are \( d \) iterations, one per digit. During each iteration:

- All numbers are bucketed: bucketing each one costs \( O(1) \), so, in total, bucketing \( n \) numbers costs \( O(n) \)
- All buckets are combined: although we do not know how many numbers are in each bucket, \( \text{in total, } n \text{ numbers are combined, so the total amount of work done combining is } O(1) \times n = O(n) \)

So, during each iteration, the total amount of work done is \( O(n) + O(n) = O(n) \), which means that the run time of LSD radix sort is \( O(dn) \). Assuming \( d \) is a constant, this sort is \( O(n) \).

MSD Radix Sort  MSD radix sort is a recursive algorithm, so what we want to know is: (i) how much total work is done at each level of the recursion tree, as a function of \( n \); and (ii) how many levels there are in the recursion tree. Let’s say \( f(n) \) work is done at each level. As there are \( d \) levels, one per digit, the total work done by MSD radix sort is then \( O(d \times f(n)) \). But what is \( f(n) \)?

- With each recursive call, \( b \) new buckets are created from 10 old buckets, one for each digit in the range \([0, b - 1]\). That’s good to know. But what we don’t know is how many numbers were in each of the old buckets; nor do we know how many will end up in each of the new buckets. So are we stuck? No! We can do an analysis in the aggregate.

  At each level of the recursion tree, \( \text{in total, } \) there are \( n \) numbers to be distributed across all the new buckets from all the old buckets. As bucketing costs \( O(1) \) per number, the total time required for bucketing is \( O(1) \times n = O(n) \).

- With each recursive call, there are \( b \) buckets to combine. As above, that’s useful information. But what we don’t know is how many numbers are in each of these buckets. So are we stuck? No! We can do an analysis in the aggregate.
At each level of the recursion tree, in total, there are $n$ numbers distributed across all buckets. So the total time required for combining is the total time to combine $n$ numbers in $b$ buckets into a single linked list: $O(n)$.

Therefore, $f(n) = O(n) + O(n) = O(n)$, so that the run time of MSD radix sort, like LSD radix sort, is $O(dn)$. Assuming $d$ is a constant, this sort is $O(n)$.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)