1 Bucket Sort

Suppose you are in a new class, and on the first day the teacher wants to get you moving and talking to your classmates. So she asks all the students to form a line, ordered by age (breaking ties by month and day, if necessary). What happens? First, students start asking each other how old they are, and groups start to form based on age, in years. But then this isn’t enough – so students start asking about birthdays: month, probably, and then days.\(^1\)

The action underlying this activity is bucket sort, a generic sorting template that uses a divide-and-conquer approach to sorting. Intuitively, objects are grouped into finer and finer “buckets” (think “year”, “month”, “day”) until they are in their correct sorted position. More generally, bucket sort has the following pseudocode.

bucket_sort(A):
  divide the elements of A into M different buckets
  for each non-empty bucket:
      sort the bucket, either with recursion and smaller buckets, or some other sorting algorithm
      combine the sorted buckets into a single sorted list, and copy the sorted list back into A

Note that the algorithm we describe is not “purely” recursive, since we allow the use of a different sorting algorithm inside the buckets. Consider quicksort, whose pseudocode is given here:

quicksort(A):
  if A.length < 2: return
  else:
      choose a pivot element in A
      divide A into two parts: those elements less than the pivot, and those greater than the pivot
      sort each part recursively, using quicksort
      combine the sorted parts together with the pivot and return the sorted array

This falls under the the bucket sort model, with each function call creating two buckets. Note that unlike the age example, here the buckets are determined as the algorithm runs, and can depend on the input. A common optimization in practice is to not recurse all the way down to lists of size 1. Instead, implementations stop when the partitions (buckets) reach size 10, and then use insertion sort instead – running faster at this size than quicksort!

1.1 Radix Sort

We will use this technique to develop an algorithm for sorting a list of positive integers known as “Most Significant Digit Radix Sort.” The “buckets” we’ll use will be labelled with the digits 0-9.

As an example, consider the following list of integers:

\[\{465, 94, 405, 91, 333, 362, 460, 5\}\]

First, we put each integer into buckets depending on their hundreds digit:

\[\{94, 91, 5\} \quad \{333, 362\} \quad \{465, 405, 460\}\]

Note that in each bucket, the elements appear in the same order they did in the original list. Then in each bucket, we sort recursively, starting at the tens digit. We’ll only show first first bucket here.

\[94, 91, 5 \rightarrow [5] \quad 94, 91\]

The next set of recursive calls sort by units digit; so, [5] stays the same, and [94, 91] becomes [91], [94]. Then the recursive calls exit and the sorted array is formed: first [91], [94] are combined to [91, 94], which is then combined with the [5] to become [5, 91, 94]. The other two buckets are sorted in a similar way, and so

\[94, 91, 5 \quad 333, 362 \quad 465, 405, 460 \quad \text{becomes} \quad 5, 91, 94 \quad 333, 362 \quad 405, 460, 465\]

\(^1\)The really precise students ask each other about the time of day to break ties, but who really knows that?
Note that the “combine” step is extremely simple, since our buckets are already positioned in the right order. So we only need to concatenate them to get the sorted list:

\[
[5, 91, 94, 333, 362, 405, 460, 465]
\]

2 A Quick Guide to Matrices

A matrix is a 2-D rectangular grid of numbers, e.g.

\[
A = \begin{pmatrix}
2 & 3 \\
1 & -1
\end{pmatrix}
\]

The above example is a 2-by-2 matrix (in this course, we’ll deal only with square matrices, i.e., matrices with the same number of rows and columns). We use uppercase letters \( A, B, C \), etc. to represent matrices, and lowercase letters to refer specific elements in the array: using the \( A \) from above, we have \( a_{11} = 2, a_{12} = 3, a_{21} = 1, \) and \( a_{22} = -1 \). Note that \( a_{ij} \) denotes the element at that \( i \)-th row and \( j \)-th column.

We can add two square matrices of the same dimensions by adding them element by element. For example:

\[
\begin{pmatrix}
2 & 3 \\
1 & -1
\end{pmatrix} + \begin{pmatrix}
5 & 0 \\
1 & 4
\end{pmatrix} = \begin{pmatrix}
2 + 5 & 3 + 0 \\
1 + 1 & -1 + 4
\end{pmatrix} = \begin{pmatrix}
7 & 3 \\
2 & 3
\end{pmatrix}
\]

Multiplying matrices takes a bit more effort. Let \( A \) and \( B \) be two \( n \)-by-\( n \) matrices, and define their product matrix \( C = AB \) as an \( n \)-by-\( n \) matrix with the following elements:

\[
c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \tag{2.1}
\]

Here is an example of multiplication for the two matrices used in the addition example:

\[
\begin{pmatrix}
2 & 3 \\
1 & -1
\end{pmatrix} \begin{pmatrix}
5 & 0 \\
1 & 4
\end{pmatrix} = \begin{pmatrix}
2 \cdot 5 + 3 \cdot 1 & 2 \cdot 0 + 3 \cdot 4 \\
1 \cdot 5 + (-1) \cdot 1 & 1 \cdot 0 + (-1) \cdot 4
\end{pmatrix} = \begin{pmatrix}
13 & 12 \\
4 & -4
\end{pmatrix}
\]

You should be able to figure out the number of basic additions and multiplications are required for both matrix addition and matrix multiplication – multiplication takes more steps than addition!

2.1 Block multiplication

Let \( A \) and \( B \) be two \( n \)-by-\( n \) matrices for an even \( n \), and let \( C = AB \) be their product. We could use the standard formula for multiplication to compute \( C \), but there is another way, by dividing \( A, B, \) and \( C \) into \( \frac{n}{2} \)-by-\( \frac{n}{2} \) blocks.

\[
A = \begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix}
\]

\[
C = \begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
\]

Then we can actually calculate the value of each block of \( C \) separately:

\[
C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\]

That is, these equations show how to reduce the problem of multiplying two \( n \)-by-\( n \) matrices to multiplying (and adding) some \( \frac{n}{2} \)-by-\( \frac{n}{2} \) matrices.
For example, consider the following matrices:

\[
A = \begin{pmatrix}
1 & 2 & 3 & 4 \\
3 & 3 & 7 & -1 \\
4 & 0 & 0 & 3 \\
-1 & -2 & -3 & -4
\end{pmatrix}, \quad B = \begin{pmatrix}
0 & 1 & 4 & 5 \\
5 & 0 & -2 & -2 \\
3 & 0 & 5 & 0 \\
2 & 2 & 6 & 7
\end{pmatrix}
\]

and let \( C = AB \). Then we can reduce this problem to computing some 2-by-2 products. For example,

\[
C_{1,1} = \begin{pmatrix}
1 & 2 \\
3 & 3
\end{pmatrix} \begin{pmatrix}
0 & 1 \\
5 & 0
\end{pmatrix} + \begin{pmatrix}
3 & 4 \\
7 & -1
\end{pmatrix} \begin{pmatrix}
3 & 0 \\
2 & 2
\end{pmatrix}
\]

\[
= \begin{pmatrix}
10 & 1 \\
15 & 3
\end{pmatrix} + \begin{pmatrix}
17 & 8 \\
19 & -2
\end{pmatrix}
\]

\[
= \begin{pmatrix}
27 & 9 \\
34 & 1
\end{pmatrix}
\]