SORTING IN LINEAR TIME - VARIATIONS ON THE BUCKET SORT*

Edward Corwin, Antonette Logar
South Dakota School of Mines and Technology
501 E St Joseph St
Rapid City, SD 57701
(605) 355-3453
edward.corwin@sdsmt.edu

ABSTRACT

The Bucket Sort is an algorithm for sorting in linear time. That is, we can sort a list of n numbers in expected time $O(n)$. Implementing the Bucket Sort shows off several important aspects of data structures and operating systems in some surprising ways. Implementation details will be discussed leading to faster variations that are not obvious.

1. THE BUCKET SORT

Unfortunately, the term "bucket sort" is used to refer to different algorithms by different authors. This paper will use the term as it is used in the text *Introduction to Algorithms* by Cormen, *et al.* [1]. The basic bucket sort takes n real numbers in [0, 1) and puts them into "buckets" that are n equal-sized subintervals, [0, 1/n), [1/n, 2/n), ... [(n-1)/n, 1). That is, the number of buckets is the same as the number of numbers to be sorted. For example, if there were ten numbers to be sorted, there would be ten buckets corresponding to splitting the interval from 0 to 1 into ten equal-sized subintervals. The buckets are managed as linked lists to allow for many values to wind up in the same subinterval. Once the numbers are in buckets, the buckets are sorted using an insertion sort. After the buckets are sorted, they are concatenated to form a sorted list of the original n values. It can be shown that the expected number of comparisons made during the insertion-sort phase to sort a list of uniformly-distributed random numbers is about 2n. This proof can be found in [1]. [The proof and pathological cases in which a sizable fraction of the values map to the same bucket are not part of the scope of this paper.] The

scatter and gather phases of the sort are clearly linear in *n*. Thus, the expected time for the bucket sort is linear in the number of values to be sorted. Other ranges and other distributions can be handled, but that is not important to this paper.

Pseudocode for bucket sort

```
for i = 1 to n
        put a[i] in bucket floor(n*a[i])
for i = 1 to n
        do an insertion sort on bucket i
for i = 1 to n
        copy contents of bucket i into sorted list
```

2. A SLOW IMPLEMENTATION

Many students implement the linked-list for the buckets in a way such that each item being inserted is put into a newly allocated node that is linked into the list. [See pseudocode for slow version below.] Thus, if there are one million values to be sorted, there are one million calls to a memory allocate function. Timing tests indicate that the memory allocation calls can take about 20% of the total run time of the algorithm. Since all the small allocated nodes need to be tracked by the memory manager at run time, this also takes much more space than is needed and unnecessarily limits the size of a list that can be sorted. For some students, this is their first clear exposure to the idea of allocating all the space that is needed in one allocate call and then using the individual spots as needed.

Pseudocode for slow version

```
for i = 1 to n
        allocate a node for a[i]
        put the value of a[i] into the new node
        link the new node into bucket floor(n*a[i])
```

Pseudocode for improved version

```
allocate an array of n nodes using one call to an allocate function
for i = 1 to n
        put the value of a[i] into node[i]
        link node[i] into bucket floor(n*a[i])
```

3. VIRTUAL MEMORY PROBLEMS

Once the allocation problem is fixed, some very long lists can be sorted. Systems that have 32-bit addresses should allow for allocations in the neighborhood of a billion bytes and, thus, lists that are hundreds of millions of entries long. However, virtual memory is usually implemented by storing pages on disk when they are not needed, and the overhead of swapping the pages in and out of memory can slow the algorithm down significantly. Most students understand clearly that an internal sorting algorithm such as the bucket sort is likely to break down for lists that cause paging to disk. This is a good

example of an appropriate use for external sorting algorithms even though the list can fit in the virtual memory space allowed by the system.

A less obvious problem occurs when the list being sorted is of a size that fits in RAM but is still quite large. For example, consider a list that is approximately a gigabyte on a system with enough RAM to hold such a list. No disk accesses will be needed to access the entire list. However, there is still a virtual memory issue in terms of the efficiency of address translation. Associative storage hardware on the CPU chip is used to hold address translation information for a modest number of recently used pages. This is often called the Translation Look-aside Buffer or TLB. As long as the pages indicated by the TLB are accessed, the overhead of the virtual memory address translation is minimal. However, in the bucket sort we often follow links to somewhat random locations in memory. This will cause misses in the TLB which require action by the operating system to reload the TLB to include information on the newly required page. This interrupt and running of kernel software, while not nearly as much of a problem as swapping to disk, will cause the bucket sort to be slower than linear long before swapping to disk is needed. For example, in Tables 1 and 2, it can be seen that doubling the size of the list doubles the time required for the bucket sort [see column labeled 'alloc 1' for the version discussed above] up to 8,000,000 items, but increasing the list size to 16,000,000 requires 2.37 times the run time as 8,000,000.

4. ONE SURPRISING EASY IMPROVEMENT

Sorting the buckets involves following pointers that may cause virtual memory overhead problems. Concatenating the buckets leads to much the same type of performance problem. Many students attempted to solve this problem by inserting items in order rather than doing a sort after inserting all values at the front of their buckets. This helps a little [see Tables 1 and 2, column labeled 'insert'], but it is possible to cut the run time significantly depending on how virtual memory is handled on the system. The three phases of the bucket sort (scatter, insertion sort, gather) all take about the same amount of time. By swapping the order of operations to scatter, gather, insertion sort, the time for the insertion sort drops to a fraction of the original. Thus, by reordering operations, about one fourth of the time can be saved. This is due to the fact that the gather operation puts the values in consecutive locations in virtual memory allowing the insertion sort to run with almost no virtual memory overhead. Interestingly, this improvement occurs on the Linux systems (running on Pentium processors), but not on the Sun systems (running Solaris on Sun hardware) as can be seen in Tables 1 and 2 in the column labeled 'gather 1st'. As an example, on one run sorting 262,000 items, the scatter time was 0.134 seconds, the gather time was 0.166 seconds, and the insertion sort time was 0.157 seconds for the original version. Reordering left the scatter and gather times the same and reduced the insertion sort time to 0.047 seconds. This is a reduction of 70% in the insertion sort time and an overall reduction of 23%. This simple sorting application has proven to be an effective tool for demonstrating the importance of understanding the interplay between virtual memory system implementations and analysis of algorithms.

5. OTHER POSSIBLE IMPROVEMENTS

The bucket sort can be modified to make the virtual memory problems less severe. By counting the number of items in each bucket, the items can be placed into the new list (the concatenated buckets) in nearly the correct positions. Note that the items in the new list will occupy contiguous memory locations. The insertion sort is then performed on the entire list. However, since the insertion sort is known to be extremely efficient for lists in which the items are nearly in their proper place, the time required to sort the list as a single entity is less than the time required to sort the buckets individually. In addition, the overhead for virtual memory is minimized if items in contiguous memory locations are accessed sequentially as they are here.

On the Sun systems, there was a speed up of 15% with this change, while the Linux systems showed no improvement using this technique. [See the column labeled 'count' in Tables 1 and 2.]

Pseudocode for counting version of bucket sort

```
for i = 1 to n
        add 1 to count of items in bucket floor(n*a[i])
for i = 0 to n-1
        sum counts to get number of items before bucket i
for i = 1 to n
        put a[i] in new list, a_new[], based on sum of counts[i]
        update sum of counts so that next item goes in correct spot
bucket sort a_new
```

In terms of discussions of algorithms and complexity that are usually done in data structures and algorithms courses, this is not really any different from the previous versions. However, in terms of operating systems and virtual memory concerns, this has better performance characteristics since there is less time spent accessing random locations in memory and more time spent accessing sequential locations.

Another possible fix to the problem of virtual memory overhead for internal sorting is to use an external sorting algorithm. Many students assume that anything that fits in memory should be sorted using an internal sort, but the above discussion shows that the choice of sorting algorithm is not so clear. Future work will compare the performance of external and internal sorting algorithms when the list size is large enough to cause performance degradation as a result of virtual memory overhead.

Finally, students in an operating systems course might look into the possibility of adding hooks into the virtual memory system to help with the management of the TLB. We hope to try this in a course next year.

6. CONCLUSION

The bucket sort is often included in sorting discussions due to its capability to sort lists in linear time. However, that analysis must be tempered with knowledge about memory management and virtual memory implementation details. Allocating memory in blocks, rather than a single node at a time, impacts the run time of the algorithm. Similarly, the order of the three component operations (scatter, gather, and insertion sort)

also noticeably affects the run time. This change in run time is a consequence of virtual memory overhead. Counting the number of items in each bucket instead of inserting them into the buckets also has better virtual memory characteristics. It should be noted that these are implementation details that do not technically affect the time complexity of the algorithm. As a practical matter, these improvements can reduce run time by more than 50%.

Finally, it is only fair to note that the quicksort is often competitive with the bucket sort, even with the improvements suggested here. The reason is that the quicksort makes efficient use of virtual memory because it processes items sequentially. Although the quicksort is O(n log n) and the bucket sort is O(n), the constants are such that the quicksort is often faster for small lists. As the size of the list grows, the bucket sort becomes faster until the virtual memory problems discussed above are encountered. For very large lists, the quicksort is again competitive.

1. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2nd edition, McGraw-Hill, 2001

8. APPENDIX - TIMINGS ON SUN AND LINUX SYSTEMS

Table 1: Times from Sun with Sun Compiler

n	quick	alloc n	alloc 1	insert	gather 1st	count
1000000	2.1	3.14	2.19	1.95	1.71	1.63
2000000	4.4	6.62	4.77	4.35	3.84	3.61
4000000	9.26	13.85	10.32	9.07	8.02	7.62
8000000	19.57	28.32	21.34	18.34	16.27	15.54
16000000	39.93	64.93	50.64	40.45	42.73	31.2
32000000	84.22	134.17	106.35	89.7099	89.4899	63.5
64000000	173.25	370.63	315.37	X	198.63	186.98

Table 2: Times from Linux (Intel) with g++

n	quick	alloc n	alloc 1	insert	gather 1st	count
125000	0.08	0.15	0.1	0.09	0.09	0.09
250000	0.16	0.29	0.24	0.22	0.19	0.21
500000	0.35	0.62	0.49	0.47	0.4	0.43
1000000	0.75	1.27	1.02	0.950001	0.809999	0.86
2000000	1.6	2.56	2.06	1.94	1.65	1.75
4000000	3.44	5.19	4.22	3.94	3.31	3.54
8000000	7.1	10.77	8.75	8.28	6.86	7.3
16000000	15.04	23.8	20.34	19.39	16.13	16.37