The Flashsort1 Algorithm

Dr. Dobb's Journal February 1998

By Karl-Dietrich Neubert

Karl does basic research in solid-state and atomic physics, biophysical systems, and computer algorithms. He can be reached at karl-dietrich.neubert@usa.net.

Many programmers are familiar with the theoretical result that "no sorting algorithm can be faster than *O(n log n)* on average." But this result assumes that you only have limited information; it assumes you only know how to compare two items. However, what if your keys are the integers from 0 to 100? In that case, you can simply place each item directly into the correct location -- no comparisons are necessary. There are many other situations where you can guess the final position of each item. This month, Karl shows how this basic idea can be used to fashion an *O(n)* sorting algorithm.

-- Tim Kientzle

Most discussions of sorting algorithms are based on comparing elements. Although it's faster to classify elements, algorithms based on classification suffer because they require considerable auxiliary memory. The Flashsort1 algorithm I present here addresses this problem. It sorts in *O(n)* time and requires less than 0.1*n* auxiliary memory to sort *n* elements. I achieve this by using classification to do the long-range ordering with in-place permutation, then use a simple comparison method for the final short-range ordering of each class. I assume here that the elements are approximately evenly distributed real numbers, although the algorithm can be adapted to other types of data.

The algorithm (see Listing One) consists of three logical blocks: classification, permutation, and straight insertion. Classification determines the size of each class of elements. Permutation does long-range reordering to collect elements of each class together, and straight insertion does the final short-range ordering.

Classification

The whole idea of a classification sort is simple: Assuming the elements *A(i)* are about evenly distributed, you can compute the approximate final position directly from the element value, with no comparisons. If the maximal element is *Amax* and the minimal element is *Amin*, you can compute:

$$K(A(i))=1+INT((m-1)(A(i)-Amin)/(Amax-Amin))$$

The result will be a value from 1 to *m*, which I'll call the class of *A(i)*. Note that, on average, you have approximately *n/m* elements in each class. (In fact, the class *K=m* only has elements equal to *Amax*, and the remaining classes are slightly larger.)

Since the main part of the algorithm relies on it, I start by computing the actual number of elements in each class. This requires simply scanning the data and counting the total number of elements.

Since I know the size of each class, I can predict where each class will appear in the final array. The vector *L* is used to track this information. Initially, each *L(K)* indicates the upper end of the section that will contain the elements in class *K*. In particular, *L*(1) is simply the number of elements in class 1, and the final *L(m)* is equal to *n*. As I rearrange things, each *L(K)* is updated.

The first part of the classification is *O(n)*, since it requires scanning the input to find the size of each class. Then there's an *O(m)* part, which adds up the class sizes to determine the initial *L(K)* values.

Permutation

The next step is to move elements into the correct class. Obviously, whenever I move one element into its new location, I have to move the old element from that location. My code uses a single temporary variable to hold the evicted element. I can then compute its new location and continue until I'm forced to stop.

The first problem is determining where to put each element. Rather than try to place an element *A(i)* exactly into its final location, I simply compute its class index *K* and place it into the location indicated by *L(K)*. I then decrement *L(K)*. As I move elements into their appropriate classes, these class pointers grow downward.

This process ends when the first class has been filled up. At that time, there are generally still elements that need to be moved because permutations usually consist of several cycles. Therefore, the second problem arises: How do you find those elements that are cycle leaders? Fortunately, it's easy to find such elements, since they're located in the wrong class. I keep a variable *j* in my implementation and use *j* to search through the array to find the next item that satisfies the condition $j<L(K(A(j)))$. The corresponding item *A(j)* serves as the new cycle leader.

Each item is moved exactly once. I can simply count the total number of moves and stop when I've moved every element. (In fact, I can stop before that; if there's only one element unmoved, it must already be in the right place.)

This process is all *O(n)*. Each item is moved once, and *j* scans through the entire array at most once.

Short-Range Ordering

Once the permutation step has moved everything into approximately the correct place, you have a partially sorted array to work with. Following Robert Sedgewick (see "Implementing Quicksort Programs," *Communications of the ACM* 21, 1978), I use a straight-insertion sort over the entire array. Assuming the classes are about the same size, this requires $O((n/m)^2)$ time for each class.

Minimizing Run Time

By adjusting the number of classes, you can fine tune the overall running time of the algorithm. If you decrease the number of classes, the classification time decreases, but the time for the straight-insertion sort increases. To estimate the optimal number of classes, I start by estimating the total time of the algorithm:

- The permutation step is *O(n)*, as is part of the classification. This contributes $a1 \cdot n$ to the total time.
- The rest of the classification is *O(m)*, contributing $a2 \cdot m$ to the total.
- The straight-insertion sort contributes $a3 \cdot m \cdot (n/m)^2$.

Here, $a1$, $a2$, and $a3$ are some numbers that I'll determine later.

I can use simple calculus to determine the optimum value. From differentiating $t=a1 \cdot n+a2 \cdot m+a3 \cdot m \cdot (n/m)^2$, I get $dt/dm=a2-a3(n/m)^2=0$. Solving for *m*, I have $m/n=\sqrt{(a3/a2)}$. In particular, I don't need to estimate $a1$.

By assuming shifts were twice as expensive as comparisons, Nicolas Wirth ("Algorithm and Datenstrukturen," B.G. Teubner, 1983) arrived at $a3=3/8$ (using some arbitrary unit of time) for the time constant of straight-insertion sorting. Following his lead, I estimated $a2=2$, which gives an optimal *m* of $0.43n$.

Figure 1 shows experimental values for the time of the various procedures as a function of *m/n*. As expected, the permutation part takes the same amount of time, regardless of *m*. Between about $m=0.2n$ and $m=0.5n$, there is a broad minimum of the overall time. For larger classes, the straight insertion becomes excessive; and for smaller classes, it takes longer to perform the initial classification. In particular, if $m>n$, then many classes are empty, and straight insertion gains you no additional time.

Figure 2 shows the experimental frequency distribution of the run time for the different procedures at fixed *n* and fixed *m/n*. The time needed for classification is, of course, constant. The permutation time varies because of the

varying time required to locate the next cycle leader. The straight-insertion sort varies depending on the size of each class. The distribution for the total run time reflects the combination of these two distributions.

The total time for Flashsort1 does not depend on the original order; partially ordered or reverse ordered elements are sorted just as fast as randomly ordered ones. However, if the values are not evenly distributed, the run time can increase unpredictably. One way to address this may be to use another classification and permutation step to sort large classes before the final straight insertion.

Comparison to Other Sorts

Figure 3 shows some experimental results on the relative run times. The separate curves indicate the running time of Flashsort1, Heapsort, and Quickersort using sequences of uniform random numbers as test numbers. As expected, the run time of Flashsort1 increases linearly with n. For $m=0.1n$, Flashsort1 is always faster than Heapsort, and is faster than Quickersort if $n>80$. For $n=10000$, Flashsort1 is approximately twice as fast as Quickersort. For $m=0.2n$, Flashsort1 is about 15 percent faster than for $m=0.1n$; for $m=0.05n$, it is about 30 percent slower, but if $n>200$, it is still significantly faster than Quickersort.

Discussion

Flashsort1 sorts by permutation in-situ in $O(n)$ time with only a short auxiliary pointer vector. In "Mathematical Analysis of Algorithms," Donald Knuth remarked "...that research on computational complexity is an interesting way to sharpen our tools for the more routine problems we face from day to day." As he points out, time effective in-situ permutation is inherently connected with the problem of finding the cycle leaders, and in-situ permutation could easily be performed in $O(n)$ time if we were allowed to manipulate n extra "tag" bits specifying how much of the permutation has been carried out at any time. Without such tag bits, he concludes "it seems reasonable to conjecture that every algorithm for in-situ permutation will require at least $n \log n$ steps on the average." In this article, I showed how n elements can be put into m property classes with m being only a small fraction of n. Instead of n tag bits, I need only an auxiliary vector of length m to sort the elements into m classes. As a final step, small numbers of partially distinguishable elements are sorted locally within their classes by a conventional sort algorithm.

Acknowledgment

I'd like to express my thanks to Christopher McManus of the Veterans Affairs Medical Center in Washington, D.C., for stimulating discussions and helpful comments on the manuscript.

References

Duccin, F. "Tri par addressage direct," *R.A.I.R.O. Informatique/Computer Science* 13, (1979), 225-259.

Feurzeig, W. "Algorithm 23: Math Sort," *Communications of the ACM* 3, (1960), 601.

Gamzon, E., C.F. Picard. "Algorithme de tri par addressage direct," *C. R. Acad. Sc*. Paris 269, (1969), 38-41.

Knuth, D.E. "Mathematical Analysis of Algorithms," *Information Processing* 71, North Holland Publishing, 1972.

Macleod, I.D.G. "An Algorithm for In Situ Permutation," *The Australian Computer Journal* 2, (1970), 16-19.

Scowen, R.S. "Algorithm 271: Quickersort," *Communications of the ACM* 8, (1965), 669-670.

Williams, J.W.J. "Algorithm 232: Heapsort," *Communications of the ACM* 7, (1964), 347-348.

DDJ

Listing One

```
      SUBROUTINE FLASH1 (A. N. L. M)C     SORTS ARRY A WITH N ELEMENTS BY USE OF INDEX VECTOR L
C     OF DIMENSION M WITH M ABOUT 0.1 N.
C     COPYRIGHT (C)  K. - D. NEUBERT  1997


      DIMENSION A(1),L(1)
C     ============================ CLASS FORMATION =====
      ANMIN=A(1)
      NMAX=1
      DO I=1,N
         IF( A(I).LT.ANMIN) ANMIN=A(I)
         IF( A(I).GT.A(NMAX)) NMAX=I
      END DO
      IF (ANMIN.EQ.A(NMAX)) RETURN
      C1=(M - 1) / (A(NMAX) - ANMIN)
      DO K=1,M
         L(K)=0
      END DO
      DO I=1,N
         K=1 + INT(C1 * (A(I) - ANMIN))
         L(K)=L(K) + 1
      END DO
      DO K=2,M
         L(K)=L(K) + L(K - 1)
      END DO
      HOLD=A(NMAX)
      A(NMAX)=A(1)
      A(1)=HOLD
C     ============================== PERMUTATION =====
      NMOVE=0
      J=1
      K=M
      DO WHILE (NMOVE.LT.N - 1)
         DO WHILE (J.GT.L(K))
            J=J + 1
            K=1 + INT(C1 * (A(J) - ANMIN))
         END DO
         FLASH=A(J)
         DO WHILE (.NOT.(J.EQ.L(K) + 1))
            K=1 + INT(C1 * (FLASH - ANMIN))
            HOLD=A(L(K))
            A(L(K))=FLASH
            FLASH=HOLD
            L(K)=L(K) - 1
            NMOVE=NMOVE + 1
         END DO
      END DO
C     ======================== STRAIGHT INSERTION =====
      DO I=N-2,1,-1
         IF  (A(I + 1).LT.A(I)) THEN
            HOLD=A(I)
            J=I
            DO WHILE  (A(J + 1).LT.HOLD)
               A(J)=A(J + 1)
               J=J + 1
            END DO
            A(J)=HOLD
         ENDIF
      END DO
C     ========================== RETURN,END FLASH1 =====
      RETURN
      END
```

[Back to Article]

♂