

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let $T(n)$ be the maximum number of comparisons that this algorithm makes on any input of size n . Notice that the running time for the algorithm must be at least as large as $T(n)$, since we are not counting data movement or other computations at all. The algorithm defines a decision tree. Observe that the height of the decision tree is exactly equal to $T(n)$, because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

As we have seen earlier, any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. This means that this sorting algorithm can *distinguish* between at most $2^{T(n)}$ different final actions. Let's call this quantity $A(n)$, for the number of different final actions the algorithm can take. Each action can be thought of as a specific way of permuting the original input to get the sorted output.

How many possible actions must any sorting algorithm distinguish between? If the input consists of n distinct numbers, then those numbers could be presented in any of $n!$ different permutations. For each different permutation, the algorithm must “unscramble” the numbers in an essentially different way, that is it must take a different action, implying that $A(n) \geq n!$. (Again, $A(n)$ is usually not exactly equal to $n!$ because most algorithms contain some redundant unreachable leaves.)

Since $A(n) \leq 2^{T(n)}$ we have $2^{T(n)} \geq n!$, implying that

$$T(n) \geq \lg(n!).$$

We can use *Stirling's approximation* for $n!$ (see page 35 in CLR) yielding:

$$\begin{aligned} n! &\geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \\ T(n) &\geq \lg \left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \right) \\ &= \lg \sqrt{2\pi n} + n \lg n - n \lg e \in \Omega(n \lg n). \end{aligned}$$

Thus we have, the following theorem.

Theorem: Any comparison-based sorting algorithm has worst-case running time $\Omega(n \lg n)$.

This can be generalized to show that the *average-case* time to sort is also $\Omega(n \lg n)$ (by arguing about the average height of a leaf in a tree with at least $n!$ leaves). The lower bound on sorting can be generalized to provide lower bounds to a number of other problems as well.

Lecture 17: Linear Time Sorting

(Tuesday, Mar 31, 1998)

Read: Chapt. 9 of CLR.

Linear Time Sorting: Last time we presented a proof that it is not possible to sort faster than $\Omega(n \lg n)$ time assuming that the algorithm is based on making 2-way comparisons. Recall that the argument was based on showing that any comparison-based sorting could be represented as a decision tree, the decision tree must have at least $n!$ leaves, to distinguish between the $n!$ different permutations in which the keys could be input, and hence its height must be at least $\lg(n!) \in \Omega(n \lg n)$.

This lower bound implies that if we hope to sort numbers faster than in $O(n \lg n)$ time, we cannot do it by making comparisons alone. Today we consider the question of whether it is possible to sort without the use of comparisons. The answer is yes, but only under very restrictive circumstances.

Many applications involve sorting small integers (e.g. sorting characters, exam scores, last four digits of a social security number, etc.). We present three algorithms based on the theme of speeding up sorting in special cases, by *not* making comparisons.

Counting Sort: Counting sort assumes that each input is an integer in the range from 1 to k . The algorithm sorts in $\Theta(n + k)$ time. If k is known to be $\Theta(n)$, then this implies that the resulting sorting algorithm is $\Theta(n)$ time.

The basic idea is to determine, for each element in the input array, its *rank* in the final sorted array. Recall that the rank of a item is the number of elements in the array that are less than or equal to it. Notice that once you know the rank of every element, you sort by simply copying each element to the appropriate location of the final sorted output array. The question is how to find the rank of an element without comparing it to the other elements of the array? Counting sort uses the following three arrays. As usual $A[1..n]$ is the input array. Recall that although we usually think of A as just being a list of numbers, it is actually a list of records, and the numeric value is the *key* on which the list is being sorted. In this algorithm we will be a little more careful to distinguish the entire record $A[j]$ from the key $A[j].key$.

We use three arrays:

$A[1..n]$: Holds the initial input. $A[j]$ is a record. $A[j].key$ is the integer key value on which to sort.

$B[1..n]$: Array of records which holds the sorted output.

$R[1..k]$: An array of integers. $R[x]$ is the rank of x in A , where $x \in [1..k]$.

The algorithm is remarkably simple, but deceptively clever. The algorithm operates by first constructing R . We do this in two steps. First we set $R[x]$ to be the number of elements of $A[j]$ whose key is equal to x . We can do this initializing R to zero, and then for each j , from 1 to n , we increment $R[A[j].key]$ by 1. Thus, if $A[j].key = 5$, then the 5th element of R is incremented, indicating that we have seen one more 5. To determine the number of elements that are less than or equal to x , we replace $R[x]$ with the sum of elements in the subarray $R[1..x]$. This is done by just keeping a running total of the elements of R .

Now $R[x]$ now contains the rank of x . This means that if $x = A[j].key$ then the final position of $A[j]$ should be at position $R[x]$ in the final sorted array. Thus, we set $B[R[x]] = A[j]$. Notice that this copies the entire record, not just the key value. There is a subtlety here however. We need to be careful if there are duplicates, since we do not want them to overwrite the same location of B . To do this, we decrement $R[i]$ after copying.

Counting Sort

```

CountingSort(int n, int k, array A, array B) { // sort A[1..n] to B[1..n]
    for x = 1 to k do R[x] = 0                // initialize R
    for j = 1 to n do R[A[j].key]++           // R[x] = #(A[j] == x)
    for x = 2 to k do R[x] += R[x-1]          // R[x] = rank of x
    for j = n downto 1 do {                   // move each element of A to B
        x = A[j].key                          // x = key value
        B[R[x]] = A[j]                       // R[x] is where to put it
        R[x]--                               // leave space for duplicates
    }
}
```

There are four (unnested) loops, executed k times, n times, $k - 1$ times, and n times, respectively, so the total running time is $\Theta(n + k)$ time. If $k = O(n)$, then the total running time is $\Theta(n)$. The figure below shows an example of the algorithm. You should trace through a few examples, to convince yourself how it works.

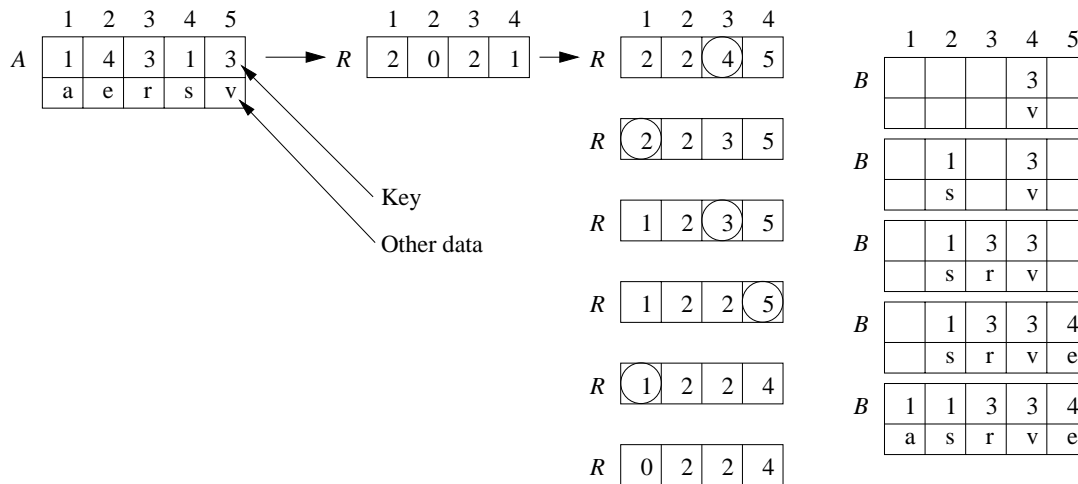


Figure 17: Counting Sort.

Obviously this not an in-place sorting algorithm (we need two additional arrays). However it is a stable sorting algorithm. I'll leave it as an exercise to prove this. (As a hint, notice that the last loop runs down from n to 1. It would not be stable if the loop were running the other way.)

Radix Sort: The main shortcoming of counting sort is that it is only really (due to space requirements) for small integers. If the integers are in the range from 1 to 1 million, we may not want to allocate an array of a million elements. Radix sort provides a nice way around this by sorting numbers one digit at a time. Actually, what constitutes a “digit” is up to the implementor. For example, it is typically more convenient to sort by bytes rather than digits (especially for sorting character strings). There is a tradeoff between the space and time.

The idea is very simple. Let's think of our list as being composed of n numbers, each having d decimal digits (or digits in any base). Let's suppose that we have access to a stable sorting algorithm, like Counting Sort. To sort these numbers we can simply sort repeatedly, starting at the lowest order digit, and finishing with the highest order digit. Since the sorting algorithm is stable, we know that if the numbers are already sorted with respect to low order digits, and then later we sort with respect to high order digits, numbers having the same high order digit will remain sorted with respect to their low order digit. As usual, let $A[1..n]$ be the array to sort, and let d denote the number of digits in A . We will not discuss how it is that A is broken into digits, but this might be done through bit manipulations (shifting and masking off bits) or by accessing elements byte-by-byte, etc.

Radix Sort

```

RadixSort(int n, int d, array A) {
    for i = 1 to d do {
        Sort A (stably) with respect to i-th lowest order digit;
    }
}

```

Here is an example.

576	49[4]	9[5]4	[1]76	176
494	19[4]	5[7]6	[1]94	194
194	95[4]	1[7]6	[2]78	278
296	\Rightarrow 57[6]	\Rightarrow 2[7]8	\Rightarrow [2]96	\Rightarrow 296
278	29[6]	4[9]4	[4]94	494
176	17[6]	1[9]4	[5]76	576
954	27[8]	2[9]6	[9]54	954

The running time is clearly $\Theta(d(n+k))$ where d is the number of digits, n is the length of the list, and k is the number of values a digit can have. This is usually a constant, but the algorithm's running time will be $\Theta(dn)$ as long as $k \in O(n)$.

Notice that we can be quite flexible in the definition of what a “digit” is. It can be any number in the range from 1 to cn for some constant c , and we will still have an $\Theta(n)$ time algorithm. For example, if we have $d = 2$ and set $k = n$, then we can sort numbers in the range $n * n = n^2$ in $\Theta(n)$ time. In general, this can be used to sort numbers in the range from 1 to n^d in $\Theta(dn)$ time.

At this point you might ask, since a computer integer word typically consists of 32 bits (4 bytes), then doesn't this imply that we can sort any array of integers in $O(n)$ time (by applying radix sort on each of the $d = 4$ bytes)? The answer is yes, subject to this word-length restriction. But you should be careful about attempting to make generalizations when the sizes of the numbers are not bounded. For example, suppose you have n keys and there are no duplicate values. Then it follows that you need at least $B = \lceil \lg n \rceil$ bits to store these values. The number of bytes is $d = \lceil B/8 \rceil$. Thus, if you were to apply radix sort in this situation, the running time would be $\Theta(dn) = \Theta(n \log n)$. So there is no real asymptotic savings here. Furthermore, the locality of reference behavior of Counting Sort (and hence of RadixSort) is not as good as QuickSort. Thus, it is not clear whether it is really faster to use RadixSort over QuickSort. This is at a level of similarity, where it would probably be best to implement both algorithms on your particular machine to determine which is really faster.

Lecture 18: Review for Second Midterm

(Thursday, Apr 2, 1998)

General Remarks: Up to now we have covered the basic techniques for analyzing algorithms (asymptotics, summations, recurrences, induction), have discussed some algorithm design techniques (divide-and-conquer in particular), and have discussed sorting algorithm and related topics. Recall that our goal is to provide you with the necessary tools for designing and analyzing efficient algorithms.

Material from Text: You are only responsible for material that has been covered in class or on class assignments. However it is always a good idea to see the text to get a better insight into some of the topics we have covered. The relevant sections of the text are the following.

- Review Chaps 1: InsertionSort and MergeSort.
- Chapt 7: Heaps, HeapSort. Look at Section 7.5 on priority queues, even though we didn't cover it in class.
- Chapt 8: QuickSort. You are responsible for the partitioning algorithm which we gave in class, not the one in the text. Section 8.2 gives some good intuition on the analysis of QuickSort.
- Chapt 9 (skip 9.4): Lower bounds on sorting, CountingSort, RadixSort.