Chapter 9 Fast Sorting and Selection

Antarctica Gentoo Penguins. U.S. government image, 2010.
Credit: Lt. Elizabeth Crapo/NOAA.

Contents

9.1 Bucket-Sort and Radix-Sort 267
9.2 Selection 270
9.3 Weighted Medians 276
9.4 Exercises 279

Most teachers keep grades recorded on computers these days. The analysis and record-keeping functions that computers provide are simply no match for paper and pencil. For instance, teachers can easily compute averages, minimums, maximums, and other statistics using functions that involve fast computations. One common computation in such applications, for instance, is bucketing or histogramming. In this computation, n students are assigned integer scores in some range, such as 0 to 100, and are then sorted based on these scores. From this sorting step, the teacher can then display a histogram that shows how many students have received each possible score, which can then be used to determine cutoffs for various letter grades. (See Figure 9.1.)

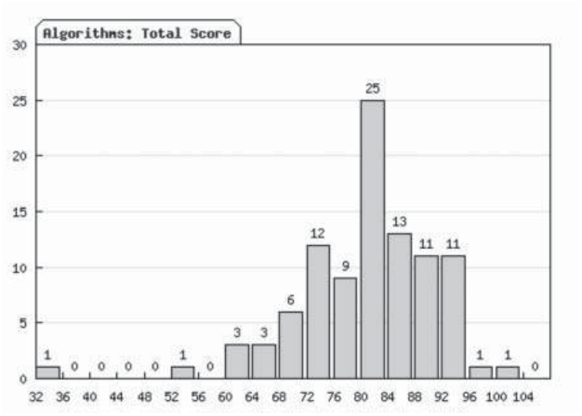

Figure 9.1: A histogram of scores from a recent Algorithms course taught by one of the authors.

When we think about the algorithmic issues in performing such a computation, it is easy to see that this is a type of sorting problem. But it is not the most general kind of sorting problem, since the keys the teacher is using to sort is simply integers in a given range. So a natural question to ask is whether we can sort n elements faster than in $O(n \log n)$ time for such specialized sorting problems; hence, beating the lower-bound on the time to sort n elements using a comparison-based algorithm, as was established in Section 8.3. Interestingly, as we explore in this chapter, it is possible to sort n elements as fast as $O(n)$ time, provided the keys being used to sort these elements are integers in a reasonably small range.

Another common computation for teachers to perform is to compute a median score, that is, a score from among n scores such that there are at most $n/2$ elements larger than this score and at most $n/2$ elements smaller than this score. Of course, such a number can be found easily if we were to sort the scores, but it would be ideal if we could find medians in $O(n)$ time without having to perform a sorting operation. As we show in this chapter, even if elements simply have a pairwise comparison rule that defines a total order, then we can find the kth smallest element in $O(n)$ time, for any value of k, including $k = n/2$. Thus, we can find medians in $O(n)$ time.

9.1 Bucket-Sort and Radix-Sort

In the previous chapter, we showed that $\Omega(n \log n)$ time is necessary, in the worst case, to sort an n-element sequence with a comparison-based sorting algorithm. A natural question to ask, then, is whether there are other kinds of sorting algorithms that can be designed to run asymptotically faster than $O(n \log n)$ time. Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted. Even so, such scenarios often arise in practice, so discussing them is worthwhile. In this section, we consider the problem of sorting a sequence of items, each a key-element pair.

9.1.1 Bucket-Sort

Consider a sequence, S, of n items whose keys are integers in the range $[0, N-1]$, for some integer $N \geq 2$, and suppose that S should be sorted according to the keys of the items. In this case, it is possible to sort S in $O(n+N)$ time. It might seem surprising, but this implies, for example, that if N is $O(n)$, then we can sort S in $O(n)$ time. Of course, the crucial point is that, because of the restrictive assumption about the format of the elements, we can avoid using comparisons.

The main idea is to use an algorithm called ***bucket-sort***, which is not based on comparisons, but on using keys as indices into a bucket array, B, that has entries from 0 to $N-1$. An item with key k is placed in the "bucket" $B[k]$, which itself is a list (of items with key k). After inserting each item of the input sequence S into its bucket, we can put the items back into S in sorted order by enumerating the contents of the buckets $B[0], B[1], \ldots, B[N-1]$ in order. We give a pseudocode description of bucket-sort in Algorithm 9.2.

```
Algorithm bucketSort($S$):
   Input: Sequence $S$ of items with integer keys in the range $[0, N-1]$
   Output: Sequence $S$ sorted in nondecreasing order of the keys

    let $B$ be an array of $N$ lists, each of which is initially empty
    for each item $x$ in $S$ do
        let $k$ be the key of $x$
        remove $x$ from $S$ and insert it at the end of bucket (list) $B[k]$
    for $i \leftarrow 0$ to $N-1$ do
        for each item $x$ in list $B[i]$ do
            remove $x$ from $B[i]$ and insert it at the end of $S$
```

Algorithm 9.2: Bucket-sort.

Analysis and the Property of Being a Stable Sorting Algorithm

It is easy to see that bucket-sort runs in $O(n+N)$ time and uses $O(n+N)$ space, just by examining the two **for** loops. Namely, the first loop runs in time $O(n)$ and the second loop runs in time $O(n+N)$. Also, sequence S uses $O(n)$ space and array B has size N. Thus, bucket-sort is efficient when the range N of values for the keys is small compared to the sequence size n, say $N = O(n)$ or $N = O(n \log n)$. Still, the performance of bucket-sort deteriorates as N grows compared to n.

In addition, an important property of the bucket-sort algorithm is that it works correctly even if there are many different elements with the same key. Indeed, we described it in a way that anticipates such occurrences.

When sorting key-element items, an important issue is how equal keys are handled. Let $S = ((k_0, e_0), \ldots, (k_{n-1}, e_{n-1}))$ be a sequence of items. We say that a sorting algorithm is ***stable*** if, for any two items (k_i, e_i) and (k_j, e_j) of S such that $k_i = k_j$ and (k_i, e_i) precedes (k_j, e_j) in S before sorting (that is, $i < j$), we have that item (k_i, e_i) also precedes item (k_j, e_j) after sorting. Stability is important for a sorting algorithm because applications may want to preserve the initial ordering of elements with the same key.

Our informal description of bucket-sort in Algorithm 9.2 does not guarantee stability. This is not inherent in the bucket-sort method itself, however, for we can easily modify our description to make bucket-sort stable, while still preserving its $O(n+N)$ running time. Indeed, we can obtain a stable bucket-sort algorithm by always removing the ***first*** element from sequence S and from each list $B[i]$ during the execution of the algorithm.

9.1.2 Radix-Sort

One of the reasons that stable sorting is so important is that it allows the bucket-sort approach to be applied to more general contexts than to sort integers. Suppose, for example, that we want to sort items with keys that are pairs (k, l), where k and l are integers in the range $[0, N-1]$, for some integer $N \geq 2$. In a context such as this, it is natural to define an ordering on these items using the ***lexicographical*** (dictionary) convention, where $(k_1, l_1) < (k_2, l_2)$ if

- $k_1 < k_2$ or
- $k_1 = k_2$ and $l_1 < l_2$.

This is a pair-wise version of the lexicographic comparison function, usually applied to equal-length character strings (and it easily generalizes to tuples of d numbers for $d > 2$).

The ***radix-sort*** algorithm sorts a sequence of pairs such as S, by applying a stable bucket-sort on the sequence twice; first using one component of the pair as the ordering key and then using the second component. But which order is correct? Should we first sort on the k's (the first component) and then on the l's (the second component), or should it be the other way around?

Before we answer this question, we consider the following example.

Example 9.1: *Consider the following sequence S:*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

If we stably sort S on the first component, then we get the sequence

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2)).$$

If we then stably sort this sequence S_1 using the second component, then we get the sequence

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7)),$$

which is not exactly a sorted sequence. On the other hand, if we first stably sort S using the second component, then we get the sequence

$$S_2 = ((1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7)).$$

If we then stably sort sequence S_2 using the first component, then we get the sequence

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)),$$

which is indeed sequence S lexicographically ordered.

So, from this example, we are led to believe that we should first sort using the second component and then again using the first component. This intuition is exactly right. By first stably sorting by the second component and then again by the first component, we guarantee that if two elements are equal in the second sort (by the first component), then their relative order in the starting sequence (which is sorted by the second component) is preserved. Thus, the resulting sequence is guaranteed to be sorted lexicographically every time. We leave the determination of how this approach can be extended to triples and other d-tuples of numbers to a simple exercise (R-9.2). We can generalize the results of this section as follows:

Theorem 9.2: *Let S be a sequence of n key-element items, each of which has a key $(k_1, k_2, \ldots, k_d)$, where k_i is an integer in the range $[0, N-1]$ for some integer $N \geq 2$. We can sort S lexicographically in time $O(d(n+N))$ using radix-sort.*

As important as it is, sorting is not the only interesting problem dealing with a total order relation on a set of elements. There are some applications, for example, that do not require an ordered listing of an entire set, but nevertheless call for some amount of ordering information about the set.

9.2 Selection

There are a number of applications in which we are interested in identifying a single element in terms of its rank relative to an ordering of the entire set. Examples include identifying the minimum and maximum elements, but we may also be interested in, say, identifying the ***median*** element, that is, the element such that half of the other elements are smaller and the remaining half are larger. In general, queries that ask for an element with a given rank are called ***order statistics***.

In this section, we discuss the general order-statistic problem of selecting the kth smallest element from an unsorted collection of n comparable elements. This is known as the ***selection*** problem. Of course, we can solve this problem by sorting the collection and then indexing into the sorted sequence at rank index $k-1$. Using the best comparison-based sorting algorithms, this approach would take $O(n \log n)$ time. Thus, a natural question to ask is whether we can achieve an $O(n)$ running time for all values of k, including the interesting case of finding the median, where $k = \lceil n/2 \rceil$.

Prune-and-Search

This may come as a small surprise, but we can indeed solve the selection problem in $O(n)$ time for any value of k. Moreover, the technique we use to achieve this result involves an interesting algorithmic technique, which is known as ***prune-and-search***. In applying this technique, we solve a given problem that is defined on a collection of n objects by pruning away a fraction of the n objects and recursively solving the smaller problem. When we have finally reduced the problem to one defined on a constant-sized collection of objects, then we solve the problem using some brute-force method. Returning back from all the recursive calls completes the construction. In some cases, we can avoid using recursion, in which case we simply iterate the prune-and-search reduction step until we can apply a brute-force method and stop.

9.2.1 Randomized Quick-Select

In applying the prune-and-search technique to the selection problem, we can design a simple and practical method, called ***randomized quick-select***, for finding the kth smallest element in an unordered sequence of n elements on which a total order relation is defined. Randomized quick-select runs in $O(n)$ ***expected*** time, taken over all possible random choices made by the algorithm, and this expectation does not depend whatsoever on any randomness assumptions about the input distribution. We note though that randomized quick-select runs in $O(n^2)$ time in the ***worst case***, the justification of which is left as an exercise (R-9.5).

Suppose we are given an unsorted sequence S of n comparable elements together with an integer $k \in [1, n]$. At a high level, the quick-select algorithm for finding the kth smallest element in S is similar in structure to the randomized quick-sort algorithm described in Section 8.2.1. We pick an element x from S at random and use this as a "pivot" to subdivide S into three subsequences L, E, and G, storing the elements of S less than x, equal to x, and greater than x, respectively. This is the prune step. Then, based on the value of k, we determine which of these sets should then be solved recursively. We describe randomized quick-select in Algorithm 9.3, and we illustrate it in Figure 9.4.

```
Algorithm quickSelect($S, k$):
  Input: Sequence $S$ of $n$ comparable elements, and an integer $k \in [1, n]$
  Output: The $k$th smallest element of $S$
  if $n = 1$ then
      return the (first) element of $S$
  pick a random element $x$ of $S$
  remove all the elements from $S$ and put them into three sequences:
     • $L$, storing the elements in $S$ less than $x$
     • $E$, storing the elements in $S$ equal to $x$
     • $G$, storing the elements in $S$ greater than $x$.
  if $k \le |L|$ then
      quickSelect($L, k$)
  else if $k \le |L| + |E|$ then
      return $x$        // each element in $E$ is equal to $x$
  else
      quickSelect($G, k - |L| - |E|$)
```

Algorithm 9.3: Randomized quick-select algorithm.

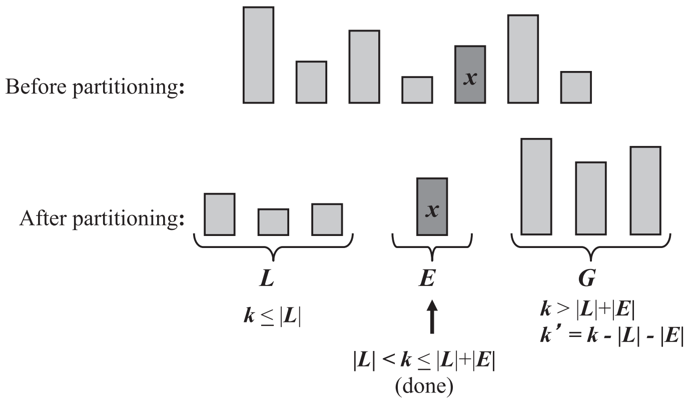

Figure 9.4: A schematic illustration of the quick-select algorithm.

Analyzing Randomized Quick-Select

We mentioned above that the randomized quick-select algorithm runs in expected $O(n)$ time. Fortunately, justifying this claim requires only the simplest of probabilistic arguments. The main probabilistic fact that we use is the ***linearity of expectation***. Recall that this fact states that if X and Y are random variables and c is a number, then $E(X + Y) = E(X) + E(Y)$ and $E(cX) = cE(X)$, where we use $E(\mathcal{Z})$ to denote the expected value of the expression $\mathcal{Z}$.

Let $t(n)$ denote the running time of randomized quick-select on a sequence of size n. Since the randomized quick-select algorithm depends on the outcome of random events, its running time, $t(n)$, is a random variable. We are interested in bounding $E(t(n))$, the expected value of $t(n)$. Moreover, since quick-select is a recursive algorithm that makes at most one additional recursive call with each invocation, the recursion tree for quick-select is simply a path. (See Figure 9.5.)

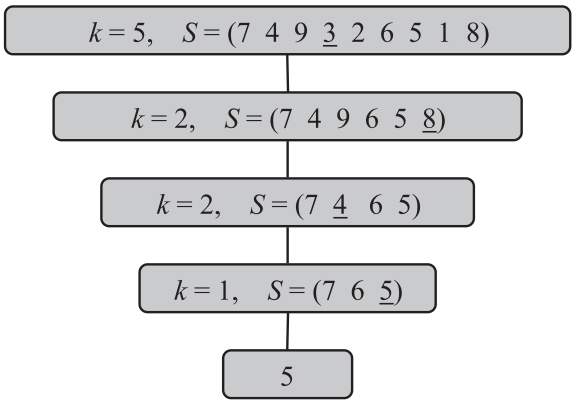
$k = 5$, $S = (7\ 4\ 9\ \underline{3}\ 2\ 6\ 5\ 1\ 8)$
$k = 2$, $S = (7\ 4\ 9\ 6\ 5\ \underline{8})$
$k = 2$, $S = (7\ \underline{4}\ 6\ 5)$
$k = 1$, $S = (7\ 6\ \underline{5})$
5

Figure 9.5: An example recursion tree for the quick-select algorithm. For each invocation, we show the integer, k, the list, S, and we underline the random pivot that is chosen.

Say that a pivot used in an invocation of randomized quick-select is "good" if it partitions S so that the size of L and G is at most $3n/4$, and it is "bad" otherwise. Clearly, a given pivot is good with probability $1/2$. (See Figure 9.6.)

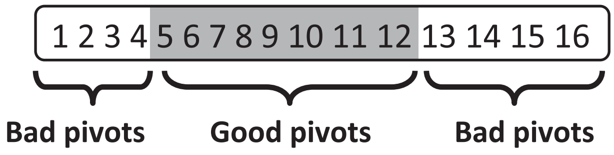

Figure 9.6: Good and bad pivots. We show an example list of 16 elements in sorted order, even though the list, S, will usually be unsorted.

Let $g(n)$ denote the number of consecutive recursive invocations (including the present one) before getting a good invocation. Then

$$t(n) \leq bn \cdot g(n) + t(3n/4),$$

where $b > 0$ is a constant (to account for the overhead of each call). We are, of course, focusing on the case where n is larger than 1, for we can easily characterize in a closed form that $t(1) = b$. Applying the linearity of expectation property to the general case, then, we get

$$E\left(t(n)\right) \leq E\left(bn \cdot g(n) + t(3n/4)\right) = bn \cdot E\left(g(n)\right) + E\left(t(3n/4)\right).$$

Since a recursive call is good with probability $1/2$, and whether a recursive call is good or not is independent of its parent call being good, the expected value of $g(n)$ is the same as the expected number of times we must flip a fair coin before it comes up "heads." This implies that $E(g(n)) = 2$. Thus, if we let $T(n)$ be a shorthand notation for $E(t(n))$ (the expected running time of the randomized quick-select algorithm), then we can write the case for $n > 1$ as

$$T(n) \leq T(3n/4) + 2bn.$$

As with the merge-sort recurrence equation, we would like to convert this equation into a closed form. To do this, let us again iteratively apply this equation assuming n is large. So, for example, after two iterative applications, we get

$$T(n) \leq T((3/4)^2 n) + 2b(3/4)n + 2bn.$$

At this point, we see that the general case is

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i.$$

In other words, the expected running time of randomized quick-select is $2bn$ times the sum of a geometric progression whose base is a positive number less than 1. Thus, by Theorem 1.12 on geometric summations, we obtain the result that $T(n)$ is $O(n)$.

To summarize, we have the following:

Theorem 9.3: *The expected running time of randomized quick-select on a sequence of size n is $O(n)$.*

As we mentioned earlier, there is a variation of quick-select that does not use randomization and runs in $O(n)$ worst-case time, which we discuss next.

9.2.2 Deterministic Selection

In this section, we discuss how to modify the quick-select algorithm to make it deterministic, yet still run in $O(n)$ time on an n-element sequence. The main idea is to modify the way we choose the pivot so that it is chosen deterministically, not randomly, based on the following approach:

1. Partition the set S into $\lceil n/5 \rceil$ groups of size 5 each (except, possibly, for one group).
2. Sort each group and identify its median element.
3. Apply the algorithm recursively on these $\lceil n/5 \rceil$ "baby medians" to find their median.
4. Use this element (the median of the baby medians) as the pivot and proceed as in the quick-select algorithm.

We give the details for this method in Algorithm 9.7.

```
Algorithm DeterministicSelect($S, k$):
  Input: Sequence $S$ of $n$ comparable elements, and an integer $k \in [1, n]$
  Output: The $k$th smallest element of $S$
  if $n = 1$ then
      return the (first) element of $S$
  Divide $S$ into $g = \lceil n/5 \rceil$ groups, $S_1, \ldots, S_g$, such that each of groups
  $S_1, \ldots, S_{g-1}$ has 5 elements and group $S_g$ has at most 5 elements.
  for $i \leftarrow 1$ to $g$ do
      Find the baby median, $x_i$, in $S_i$ (using any method)
  $x \leftarrow$ DeterministicSelect($\{x_1, \ldots, x_g\}$, $\lceil g/2 \rceil$)
  remove all the elements from $S$ and put them into three sequences:
      • $L$, storing the elements in $S$ less than $x$
      • $E$, storing the elements in $S$ equal to $x$
      • $G$, storing the elements in $S$ greater than $x$.
  if $k \le |L|$ then
      DeterministicSelect($L, k$)
  else if $k \le |L| + |E|$ then
      return $x$        // each element in $E$ is equal to $x$
  else
      DeterministicSelect($G, k - |L| - |E|$)
```

Algorithm 9.7: The deterministic selection algorithm.

Analysis of Deterministic Selection

We now show that the above deterministic selection algorithm runs in linear time.

The algorithm has two recursive calls. The first one is performed on the set of baby medians, which has size

$$g = \lceil n/5 \rceil.$$

The second recursive call is made on either set L (elements smaller than the pivot, x) or set G (elements larger than the pivot, x). Recall that each group but one contains 5 elements and our pivot, x, is the median of the baby medians from all of these groups. Thus, we have that for $\lceil g/2 \rceil$ groups, at least half of the group elements are less than or equal to x. Since group S_g could be part of this half, we have that number of elements in S that are less than or equal to x is at least

$$3\left(\left\lceil \frac{g}{2} \right\rceil - 1\right) + 1 = 3\left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \geq \frac{3n}{10} - 2.$$

With a similar argument, we obtain that the above value is also a lower bound on the number of elements of S less than or equal to x.

We conclude that the second recursive call is performed on a set of size at most

$$n - \left(\frac{3n}{10} - 2\right) = \frac{7n}{10} + 2.$$

Overall, for a sufficiently large value of n, the running time for the deterministic selection algorithm, $T(n)$, can be characterized by the following recurrence relation:

$$T(n) \leq T(n/5 + 1) + T(7n/10 + 2) + bn.$$

where $b > 0$ is a constant.

To solve the recurrence, we guess that $T(n) \leq cn$, for some constant $c > 0$. Expanding the recurrence, we have the following:

$$\begin{aligned} T(n) &\leq T(n/5 + 1) + T(7n/10 + 2) + bn \\ &\leq cn/5 + c + 7cn/10 + 2c + bn \\ &= 9cn/10 + bn + 3c. \end{aligned}$$

Pick $c = 11b$. We obtain

$$T(n) \leq 9cn/10 + bn + 3c \leq 9cn/10 + cn/11 + 3c.$$

Thus, we have $T(n) \leq cn$ for n large enough such that

$$cn/11 + 3c \leq cn/10,$$

that is, for $n \geq 330$. Therefore, the running time of the deterministic selection algorithm is $O(n)$.

We summarize the above analysis with the following theorem.

Theorem 9.4: *Given an input sequence with n elements, the deterministic selection algorithm runs in $O(n)$ time.*

Admittedly, the constant in the $O(n)$ running time for deterministic selection, as estimated by the analysis, is fairly high. Thus, in practice, it is probably more efficient to use the randomized quick-select algorithm than this deterministic selection algorithm. Still, it is useful to know that we can match the expected $O(n)$ running time for randomized quick-select with a deterministic algorithm.

9.3 Weighted Medians

In some applications, elements have weights and we wish to find a median of the elements that respects these weights. For example, suppose we have a set of people with distinct ages and weights, who want to cross a river, and we have a ship that has capacity to support roughly half the total weight, W, of all the people. Suppose further that these people are willing to admit to their ages, but not their weights. Thus, we would like to announce to them that everyone who is younger than some amount, x, can go on the first sailing and everyone else can go on the sailing after that. That is, we are interested in finding the age, x, such that the total weight of everyone younger than x is at most $W/2$, and the total weight of everyone older than x is at most $W/2$.

Formal Definition of a Weighted Median

Formally, let us assume we are given a set,

$$X = \{x_1, x_2, \ldots, x_n\},$$

of n distinct elements taken from some total order, such that each element x_i has a positive weight w_i. Suppose further that these weights sum up to

$$W = \sum_{i=1}^{n} w_i.$$

We are interested in finding an element, x_m in X, such that

$$\sum_{x_i < x_m} w_i \;\le\; \frac{W}{2}$$

and

$$\sum_{x_i > x_m} w_i \;\le\; \frac{W}{2}.$$

Note that these sums do not include the weight of the element, w_m, itself. Note that the above conditions imply that the smallest x_m such that

$$\sum_{x_i \le x_m} w_i \;>\; \frac{W}{2},$$

is a weighted median element.

A Solution Based on Sorting

For a first cut at a solution, let us consider an algorithm based on sorting. Specifically, imagine that we sort the elements in X by their values (and not their weights). Then we can scan this sorted sequence, from the beginning, while keeping a running total of the weights of the elements we have encountered so far. As soon as this running total goes over $W/2$, then we will have found a weighted median, w_m. (See Algorithm 9.8.)

```
Algorithm SortedMedian($X$):
   Input: A set, $X$, of distinct elements, with each $x_i$ in $X$ having a positive
      weight, $w_i$
   Output: The weighted median for $X$
    Let $W$ be the sum of all the weights of the elements in $X$
    Let the sequence $(x_1, x_2, \ldots, x_n)$ be the result of sorting $X$
    $w \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        $w \leftarrow w + w_i$
        if $w > W/2$ then
            return $x_i$
```

Algorithm 9.8: A sorting-based algorithm for the weighted median problem.

This algorithm is clearly correct, since it incrementally builds the set of elements less than the weighted median until it is found, based on the formal characterization given above. The total running time of this method is dominated by the running time for the sorting step, which takes $O(n \log n)$ time, since we are assuming only that the elements in X come from some total order.

A Solution Based on Prune-and-Search

We can design a more efficient algorithm for the weighted median problem, however, by using the prune-and-search technique. The main idea is reminiscent of binary search. Use the linear-time selection algorithm from the previous section to find the median, y, in X, without taking weights into consideration. Then compute the weight of every element less than y and the weight of every element less than or equal to y. If the first of these is greater than $W/2$, then y is too large; hence, we should recursively solve the problem on elements less than y. If the second of these sums is less than $W/2$, then y is too small; hence, we should recursively solve the problem on elements greater than y. Otherwise, y is the weighted median. The method, PruneMedian, which is shown in Algorithm 9.9, performs this computation, where we initially pass in the set X and half its total weight, $W/2$, of the elements in X.

```
Algorithm PruneMedian($X, W$):
  Input: A set, $X$, of distinct elements, $\{x_1, \ldots, x_n\}$, with each $x_i$ in $X$ having
     a positive weight, $w_i$; and a weight, $W$
  Output: The element, $y$, in $X$, such that the total weight of the elements in $X$
     less than $y$ is at most $W$ and the total weight of the elements in $X$ less than
     or equal to $y$ is greater than or equal to $W$
   if $n = 1$ then
       return $x_1$
   Let $y \leftarrow$ DeterministicSelect($X, \lceil n/2 \rceil$)
   Let $w_1 \leftarrow \sum_{x_i<y} w_i$
   Let $w_2 \leftarrow \sum_{x_i \le y} w_i$
   if $w_1 > W$ then   // $y$ is too large
       Let $X'$ be the set of elements in $X$ that are less than $y$
       Call PruneMedian($X', W$)
   else if $w_2 < W$ then   // $y$ is too small
       Let $X'$ be the set of elements in $X$ greater than $y$
       Let $W'$ be the sum of the weights of the elements in $X - X'$
       Call PruneMedian($X', W - W'$)
   else
       return $y$
```

Algorithm 9.9: A prune-and-search algorithm for the weighted median problem.

Analysis of the Prune-and-Search Weighted Median Algorithm

The running time, $T(n)$, for this algorithm can be characterized by $T(n)$ being at most a constant if $n = 1$, and the following recurrence equation otherwise:

$$T(n) \le T(\lceil n/2 \rceil) + bn,$$

where $b > 0$ is a constant. This gives us the following:

Theorem 9.5: *The weighted median problem can be solved in $O(n)$ time.*

Proof: To see that the prune-and-search weighted median algorithm runs in $O(n)$ time, we will prove the claim that $T(n) \le cn$, for some constant $c > 0$ by induction. For the case when $n < 4$, we take c larger than the constant time needed for the algorithm when this is the input size. Otherwise, for $n \ge 4$,

$$\begin{aligned} T(n) &\le T(\lceil n/2 \rceil) + bn \\ &\le T(n/2 + 1) + bn \\ &\le cn/2 + c + bn \\ &\le cn, \end{aligned}$$

for $c \ge 4b$, since, in this case, $cn/4 \ge bn$ and $cn/4 \ge c$. ■

Therefore, we can solve the weighted median problem in linear time using the prune-and-search technique.

9.4 Exercises

Reinforcement

R-9.1 Which, if any, of the algorithms bubble-sort, heap-sort, merge-sort, and quick-sort are stable?

R-9.2 Describe a radix-sort method for lexicographically sorting a sequence S of triplets (k, l, m), where k, l, and m are integers in the range $[0, N-1]$, for some $N \geq 2$. How could this scheme be extended to sequences of d-tuples $(k_1, k_2, \ldots, k_d)$, where each k_i is an integer in the range $[0, N-1]$?

R-9.3 Is the bucket-sort algorithm in-place? Why or why not?

R-9.4 Give a pseudocode description of an in-place quick-select algorithm.

R-9.5 Show that the worst-case running time of quick-select on an n-element sequence is $\Omega(n^2)$.

R-9.6 Explain where the induction proof for showing that deterministic selection runs in $O(n)$ time would fail if we formed groups of size 3 instead of groups of size 5.

R-9.7 What does the weighted median algorithm return if the weights of all the elements are equal?

Creativity

C-9.1 Show that any comparison-based sorting algorithm can be made to be stable, without affecting the asymptotic running time of this algorithm.

Hint: Change the way elements are compared with each other.

C-9.2 Suppose we are given two sequences A and B of n integers, possibly containing duplicates, in the range from 1 to $2n$. Describe a linear-time algorithm for determining if A and B contain the same set of elements (possibly in different orders).

C-9.3 Suppose we are given a sequence S of n elements, each of which is an integer in the range $[0, n^2 - 1]$. Describe a simple method for sorting S in $O(n)$ time.

Hint: Think of alternate ways of viewing the elements.

C-9.4 Let $S_1, S_2, \ldots, S_k$ be k different sequences whose elements have integer keys in the range $[0, N-1]$, for some parameter $N \geq 2$. Describe an algorithm running in $O(n + N)$ time for sorting all the sequences (not as a union), where n denotes the total size of all the sequences.

C-9.5 Suppose we are given a sequence, S, of n integers in the range from 1 to n^3. Give an $O(n)$-time method for determining whether there are two equal numbers in S.

C-9.6 Let A and B be two sequences of n integers each, in the range $[1, n^4]$. Given an integer x, describe an $O(n)$-time algorithm for determining if there is an integer a in A and an integer b in B such that $x = a + b$.

C-9.7 Given an unordered sequence S of n comparable elements, describe a linear-time method for finding the $\lceil \sqrt{n} \rceil$ items whose rank in an ordered version of S is closest to that of the median.

C-9.8 Show how a deterministic $O(n)$-time selection algorithm can be used to design a quick-sort-like sorting algorithm that runs in $O(n \log n)$ ***worst-case*** time on an n-element sequence.

C-9.9 Given an unsorted sequence S of n comparable elements, and an integer k, give an $O(n \log k)$ expected-time algorithm for finding the $O(k)$ elements that have rank $\lceil n/k \rceil$, $2\lceil n/k \rceil$, $3\lceil n/k \rceil$, and so on.

C-9.10 Suppose you are given two sorted lists, A and B, of n elements each, all of which are distinct. Describe a method that runs in $O(\log n)$ time for finding the median in the set defined by the union of A and B.

C-9.11 Given a set of n elements that come from a total order, show that you can find the second smallest element in this set using $n + \lceil \log n \rceil - 2$ comparisons.

C-9.12 Given an array, A, of n numbers in the range from 1 to n, describe an $O(n)$-time method for finding the ***mode***, that is, the number that occurs most frequently in A.

C-9.13 Suppose instead of choosing a single pivot in the quick-select algorithm, we chose $\log n$ pivots. Show that the probability that at least one of them is good is at least $1 - 1/n$.

Applications

A-9.1 Suppose you would like to find the most average kitten in your collection of n kitten photographs, based on cuteness. So as to avoid your own personal biases, any time you would need to compare two kitten photos, x and y, rather than doing this yourself, you use an online crowdsourcing application, Decider, to decide which kitten is cuter. You may assume that there is a total ordering of your kitten photographs, based on cuteness, but, for any given time that you make a call to Decider to compare two kittens, x and y, there is an independent 50-50 chance that the people Decider picks to perform this comparison cannot agree on which kitten is cuter. That is, for each comparison of two kittens that Decider is asked to perform, it is as if it flips a fair coin and answers the comparison accurately if the coin turns up "heads" and answers "cannot decide" if the coin turns up "tails." Moreover, this behavior occurs independent of previous comparison requests, even for the same pair of kitten photographs. Describe an efficient algorithm that can correctly use Decider to find the median kitten photograph, based on cuteness, and show that your algorithm makes an expected number of calls to Decider that is $O(n)$.

A-9.2 Search engines often index their collections of documents so that they can easily return an ordered set of documents that contain a given query word, w. Such a data structure is known as an ***inverted file***. In order to construct an inverted file, we might start with a set of n triples of the form, (w, d, r), where w is a word, d is an identifier for a document that contains the word w, and r is a rank score for the popularity of the document d. Often, the next step is to sort these triples, ordered first by w, then by r, and then by d. Assuming each of the values, w, d, and r, are represented as integers in the range from 1 to $4n$, describe a linear-time algorithm for sorting a given set of n such triples.

A-9.3 Suppose you are the postmaster in charge of putting a new post office in a small town, where all the houses are along one street, where the new post office should go as well. Let us view this street as a line and the houses on it as a set of real numbers, $\{x_1, x_2, \ldots, x_n\}$, corresponding to points on this line. To make everyone in town as happy as possible, the location, p, for the new post office should minimize the sum,

$$\sum_{i=1}^{n} |p - x_i|.$$

Describe an efficient algorithm for finding the optimal location for the new post office, show that your algorithm is correct, and analyze its running time.

A-9.4 Suppose University High School (UHS) is electing its student-body president. Suppose further that everyone at UHS is a candidate and voters write down the student number of the person they are voting for, rather than checking a box. Let A be an array containing n such votes, that is, student numbers for candidates receiving votes, listed in no particular order. Your job is to determine if one of the candidates got a majority of the votes, that is, more than $n/2$ votes. Describe an $O(n)$-time algorithm for determining if there is a student number that appears more than $n/2$ times in A.

A-9.5 Consider the election problem from the previous exercise, but now describe an algorithm running in $O(n)$ time to determine the student numbers of every candidate that received more than $n/3$ votes.

A-9.6 Computational ***metrology*** deals with algorithms for the science of measurement. For instance, suppose we are given a set, S, of n points in 3-dimensional spaces, which is defined by sampling the surface of a manufactured part with a laser range-finding device. A possible problem in computational metrology is to precisely determine how flat the points in S are, based on some mathematical definition of "flatness." But in order to determine how flat a set of points is in 3-dimensional space, we must have some reference plane. Therefore, let us define the reference plane for S to be the plane, $z = c$, that minimizes the sum of distances from points in S to this plane, that is, the plane that minimizes the sum

$$F(S) = \sum_{p \in S} |z(p) - c|,$$

where $z(p)$ denotes the z-coordinate of the point p. The ***flatness*** of S is then defined as $F(S)$. Describe an efficient algorithm for computing the flatness of S defined in this way. What is the running time of your method?

A-9.7 Suppose you are the owner of a chain of premium coffee shops that sell high-priced coffee with fancy Italian names to college students. You have learned that there is a street in a large college town that is lined with n dormitories and there currently is no coffee shop on this street. Your goal is to place a new coffee shop on this street so as to optimize the distance from this shop to the various dormitories. To simplify things, let us model the street as a line and each dormitory as a point, d_i, which is a real number on this line. In addition, we know the number of people, p_i, who live in each dormitory, d_i. You are interested in finding the location, x, that minimizes the cost function,

$$\sum_{i=1}^{n} p_i |d_i - x|.$$

Describe an efficient algorithm for finding the point, x, where to place your coffee shop, that minimizes this cost. What is the running time of your algorithm?

Chapter Notes

Knuth's classic text on *Sorting and Searching* [131] contains an extensive history of the sorting problem and algorithms for solving it, starting with the census card sorting machines of the late 19th century. Gonnet and Baeza-Yates [85] provide experimental comparisons and theoretical analyses of a number of different sorting algorithms. The term "prune-and-search" originally comes from the computational geometry literature (such as in the work of Clarkson [45] and Megiddo [154, 155]).