Problem Set 2 Solutions

Problem 2-1. Is this (almost) sorted?

Harry Potter, the child wizard of Hogwarts fame, has once again run into trouble. Professor Snape has sent Harry to detention and assigned him the task of sorting all the old homework assignments from the last 200 years. Being a wizard, Harry waves his wand and says, *ordinatus sortitus*, and the papers rapidly pile themselves in order.

Professor Snape, however, wants to determine whether Harry's spell correctly sorted the papers. Unfortunately, there are a large number n of papers and determining whether they are in perfect order takes $\Omega(n)$ time.

Professor Snape instead decides to check whether the papers are *almost* sorted. He wants to know whether 90% of the papers are sorted: is it possible to remove 10% of the papers and have the resulting list be sorted?

In this problem, we will help Professor Snape to find an algorithm that takes as input a list A containing n distinct elements, and acts as follows:

- If the list A is sorted, the algorithm always returns **true**.
- If the list A is *not* 90% sorted, the algorithm returns **false** with probability at least 2/3.

(a) Professor Snape first considers the following algorithm:

> Repeat k times:
>
> 1. Choose a paper i independently and uniformly at random from the open interval $(1, n)$. (That is, $1 < i < n$.)
> 2. Compare paper $A[i-1]$ and $A[i]$. Output **false** and halt if they are not sorted correctly.
> 3. Compare paper $A[i]$ and $A[i+1]$. Output **false** and halt if they are not sorted correctly.
>
> Output **true**.

Show that for this algorithm to correctly discover whether the list is almost sorted with probability at least $2/3$ requires $k = \Omega(n)$. *Hint:* Find a sequence that is not almost sorted, but with only a small number of elements that will cause the algorithm to return **false**.

Solution: We show that Snape's algorithm does not work by constructing a counterexample that has the following two properties:

- A is *not* 90% sorted.

- Snape's algorithm outputs false with probability $2/3$ only if $k = \Omega(n)$.

In particular, we consider the following counter-example:

$$A = [\lfloor n/2 \rfloor + 1, \ldots, n, 1, 2, 3, \ldots, \lfloor n/2 \rfloor] \,.$$

Lemma 1 *A is not 90% sorted.*

Proof. Assume, by contradiction, that the list is 90% sorted. Then, there must be some 90% of the elements that are correctly ordered with respect to each other. There must be one of these correctly ordered elements in the first half of the list, i.e., with index $i \leq \lfloor n/2 \rfloor$. Also, there must be one of these correctly ordered elements in the second half of the list, i.e. with index $j > \lfloor n/2 \rfloor$. However, $A[i] > A[j]$, by construction, which is a contradiction. Therefore A is not 90% sorted.

Lemma 2 *Snape's algorithm outputs false with probability $2/3$ only if $k = \Omega(n)$.*

Proof. Notice that on each iteration of the algorithm, there are only two choices that allow the algorithm to detect that the list is not sorted: $i = \lfloor n/2 \rfloor$ or $i = \lfloor n/2 \rfloor + 1$. Define indicator random variables as follows:

$$X_\ell = \begin{cases} 1 & \text{if } i = \lfloor n/2 \rfloor \text{ or } i = \lfloor n/2 \rfloor + 1 \text{ on iteration } \ell, \\ 0 & \text{otherwise.} \end{cases}$$

Notice, then, that $\Pr\{X_\ell = 1\} = 2/n$ and $\Pr\{X_\ell = 0\} = (1 - 2/n)$. Therefore, the probability that Snape's algorithm *does not* output false for all k iterations (i.e., that Snape's algorithm does not work) is:

$$\begin{aligned} &= \prod_{\ell=1}^{k} \Pr\left(X_\ell = 0\right) \\ &= \left(1 - \frac{2}{n}\right)^k \end{aligned}$$

We want to determine the minimum value of k for which Snape's algorithm works, that is, the minimum value of k such that the probability of failure is no more than $1/3$:

$$\left(1 - \frac{2}{n}\right)^k \leq \frac{1}{3} \,.$$

Solving for k, we determine that:

$$k \geq \frac{\ln\left(1/3\right)}{\ln\left(1 - \frac{2}{n}\right)} \,.$$

We now recall the following math fact (see CLRS 3.11):

$$\left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e} \,.$$

From this, we calculate that:

$$\ln\left(1-\frac{1}{x}\right) \leq -\frac{1}{x}\,.$$

We then conclude the following:

$$\begin{aligned} k &\geq \frac{\ln(1/3)}{\ln\left(1-\frac{2}{n}\right)} \\ &\geq \frac{\ln(1/3)}{-2/n} \\ &\geq \frac{n\ln 3}{2}\,. \end{aligned}$$

We conclude that Snape's algorithm is correct only if $k = \Omega(n)$.

(b) Imagine you are given a bag of n balls. You are told that at least 10% of the balls are blue, and no more than 90% of the balls are red. Asymptotically (for large n) how many balls do you have to draw from the bag to see a blue ball with probability at least $2/3$? (You can assume that the balls are drawn with replacement.)

Solution: Since the question only asked the asymptotic number of balls drawn, $\Theta(1)$ (plus some justification) is a sufficient answer. Below we present a more complete answer.

Assume you draw k balls from the bag (replacing each ball after examining it).

Lemma 3 *For some constant k sufficiently large, at least one ball is blue with probability $2/3$.*

Proof. Define indicator random variables as follows:

$$X_i = \begin{cases} 1 & \text{if ball } i \text{ is blue} \\ 0 & \text{if ball } i \text{ is red} \end{cases}$$

Notice, then, that $\Pr X_i = 1 = 1/10$ and $\Pr X_i = 0 = 9/10$. We then calculate the probability that at least one ball is blue:

$$\begin{aligned} &= 1 - \prod_{i=1}^{k} \Pr\left(X_i = 0\right) \\ &= 1 - \left(\frac{9}{10}\right)^k \\ &\geq \frac{2}{3}\,. \end{aligned}$$

Therefore, if $k = \lg(1/3)/\lg 0.9$, the probability of drawing at least one blue ball is at least $2/3$.

(c) Consider performing a "binary search" on an unsorted list:

```
BINARY-SEARCH(A, key, left, right)          ▷ Search for key in A[left . . right].
1  if left = right
2     then return left
3     else  mid ← ⌈(left + right)/2⌉
4           if key < A[mid]
5              then return BINARY-SEARCH(A, key, left, mid − 1)
6              else  return BINARY-SEARCH(A, key, mid, right)
```

Assume that a binary search for key_1 in A (even though A is not sorted) returns slot i. Similarly, a binary search for key_2 in A returns slot j. Explain why the following fact is true: if $i < j$, then $key_1 \leq key_2$. Draw a picture. *Hint:* First think about why this is obviously true if list A is sorted.

Solution:

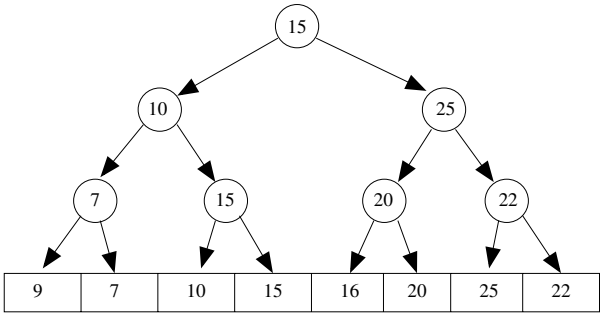

Figure 1: An example of a binary-search decision tree of an unordered array.

For the purpose of understanding this problem, think of the decision-tree version of the binary search algorithm. (Notice that unlike sorting algorithms, the decision tree for binary search is relatively small, i.e., $O(n)$ nodes.)

Consider the example in Figure 1, in which a binary search is performed on the unsorted array $[9\ 7\ 10\ 15\ 16\ 20\ 25\ 22]$. Assume $key_1 = 20$ and $key_2 = 25$. Both 20 and 25 are ≥ 25, and choose the right branch from the root. At this point, the two binary searches diverge: $20 < 25$ and $25 \geq 25$. Therefore key_1 takes the left branch and key_2 takes the right branch. This ensures that eventually key_1 and key_2 are ordered correctly.

We now generalize this argument. For key_1, let $x_1, x_2, \ldots, x_k$ be the k elements that are compared against key_1 in line 4 of the binary search (where $k = O(\lg n)$). (In the example, $x_1 = 15$, $x_2 = 25$, and $x_3 = 20$.) Let $y_1, y_2, \ldots, y_t$ be the t elements compared against key_2. We know that $x_1 = y_1$. Let ℓ be the smallest number such that $x_\ell \neq y_\ell$. (In particular, $\ell > 1$.) Since $i < j$, by assumption, we know that key_1 cannot

branch right while key_2 simultaneously branches left. Hence we conclude that

$$key_1 < x_{\ell-1} = y_{\ell-1} \leq key_2 \ .$$

(Note that a relatively informal solution was acceptable for the problem, as we simply asked that you "explain why" this is true. The above argument can be more carefully formalized.)

(d) Professor Snape proposes a randomized algorithm to determine whether a list is 90% sorted. The algorithm uses the function $\text{RANDOM}(1, n)$ to choose an integer independently and uniformly at random in the closed interval $[1, n]$. The algorithm is presented below.

```
IS-ALMOST-SORTED(A, n, k)                ▷ Determine if A[1..n] is almost sorted.
1  for r ← 1 to k
2      do i ← RANDOM(1, n)               ▷ Pick i uniformly and independently.
3         j ← BINARY-SEARCH(A, A[i], 1, n)
4         if i ≠ j
5           then return false
6  return true
```

Show that the algorithm is correct if k is a sufficiently large constant. That is, with k chosen appropriately, the algorithm always outputs **true** if a list is correctly sorted and outputs **false** with probability at least $2/3$ if the list is not 90% sorted.

Solution: Overview: In order to show the algorithm correct, there are two main lemmas that have to be proved: (1) If the list A is sorted, the algorithm always returns **true**; (2) If the list A is *not* 90% sorted, the algorithm returns **false** with probability at least 2/3. We begin with the more straightforward lemma, which essentially argues that binary search is correct. We then show that if the list is not 90% sorted, then at least 10% of the elements fail the "binary search test." Finally, we conclude that for a sufficiently large constant k, if the list is not 90% sorted, then the algorithm will output **false** with probability at least $2/3$.

Lemma 4 *If the list A is sorted, the algorithm always returns* **true***.*

Proof. This lemma follows from the correctness of binary search on a sorted list, which was shown in recitation one. The invariant is that *key* is in array A between *left* and *right*.

For the rest of this problem, we label the elements as "good" and "bad" based on whether they pass the binary sort test.

$$label(i) = \begin{cases} \text{good} & \text{if } i = \text{BINARY-SEARCH}(A, A[i], 1, n) \\ \text{bad} & \text{if } i \neq \text{BINARY-SEARCH}(A, A[i], 1, n) \end{cases}$$

Notice that it is not immediately obvious which elements are good and which elements are bad. In particular, some elements may appear to be sorted correctly, but be bad because of other elements being missorted. Similarly, some elements may appear entirely out of place, but be good because of other misplaced elements. A key element of the proof is showing that a badly sorted list has a lot of bad elements.

Lemma 5 *If the list A is* not *90% sorted, then at least 10% of the elements are bad.*

Proof. Assume, by contradiction, that fewer than 10% of the elements are bad. Then, at least 90% of the elements are good. Recall the definition of a 90% sorted list: if 10% of the elements are removed, then the remaining elements are in sorted order. Therefore, remove all the bad elements from the array. We now argue that the remaining elements are in sorted order. Consider any two of the remaining good elements, key_1 and key_2, where key_1 is at index i and key_2 is at index j. If $i < j$, then Part(c) shows that $key_1 \leq key_2$. Similarly, if $j < i$, then Part(c) shows that $key_2 \leq key_1$. That is, the two elements are in correctly sorted order. Since all pairs of elements are in sorted order, the array of good elements is in sorted order.

Once we have shown that there are a lot of bad elements, it remains to show that we find a bad element through random sampling.

Lemma 6 *If the list A is* not *90% sorted, the algorithm returns* **false** *with probability at least 2/3.*

Proof. From Lemma 5, we know that at least 10% of the elements are bad. From Part(b), we know that if we choose $k > \lg(1/3)/\lg 0.9$, then with probability $2/3$ we find a bad element. Therefore, we conclude that the algorithm returns **false** with probability at least 2/3.

(e) Imagine instead that Professor Snape would like to determine whether a list is $1-\epsilon$ sorted for some $0 < \epsilon < 1$. (In the previous parts $\epsilon = 0.10$.) For large n, determine the appropriate value of k, asymptotically, and show that the algorithm is correct. What is the overall running time?

Solution: Lemma 4 is the same as in Part(d). A simple modification of Lemma 5 shows that if the array is not $(1-\epsilon)$-sorted, then there must be at least ϵn bad elements; otherwise, the remaining $(1-\epsilon)n$ elements would form a $(1-\epsilon)$-sorted list. Finally, it remains to determine the appropriate value of k

In this case, we want to choose k such that

$$(1-\epsilon)^k \leq \frac{1}{3}$$

We choose $k = c/\epsilon$, then we can conclude (using CLRS 3.11) that:

$$\begin{aligned}(1-\epsilon)^k &\leq \left((1-\epsilon)^{1/\epsilon}\right)^c \\ &\leq \left(\frac{1}{e}\right)^c\end{aligned}$$

We therefore conclude that if $k = \Theta(1/\epsilon)$, the algorithm will find a bad element with probability at least $2/3$. The running time of the algorithm is $O(\lg n/\epsilon)$.

Problem 2-2. Sorting an almost sorted list.

On his way back from detention, Harry runs into his friend Hermione. He is upset because Professor Snape discovered that his sorting spell failed. Instead of sorting the papers correctly, each paper was within k slots of the proper position. Hermione immediately suggests that insertion sort would have easily fixed the problem. In this problem, we show that Hermione is correct (as usual). As before, $A[1\,..\,n]$ in an array of n distinct elements.

(a) First, we define an "inversion." If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. What permutation of the array $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

Solution: The permuation $\{n, n-1, \ldots, 2, 1\}$ has the largest number of inversions. It has $\binom{n}{2} = n(n-1)/2$ inversions.

(b) Show that, if every paper is initially within k slots of its proper position, insertion sort runs in time $O(nk)$. *Hint:* First, show that INSERTION-SORT(A) runs in time $O(n + I)$, where I is the number of inversions in A.

Solution: Overview: First we show that INSERTION-SORT(A) runs in time $O(n+I)$, where I is the number of inversions, by examining the insertion sort algorithm. Then we count the number of possible inversions in an array in which every element is within k slots of its proper position. We show that there are at most $O(nk)$ inversions.

Lemma 7 INSERTION-SORT*(A) runs in time $O(n + I)$, where I is the number of inversions in A.*

Proof. Consider an execution of INSERTION-SORT on an array A. In the outer loop, there is $O(n)$ work. Each iteration of the inner loop fixes exactly one inversion. When the algorithm terminates, there are no inversions left. Hence, there must be I iterations of the inner loop, resulting in $O(I)$ work. Therefore the running time of the algorithm is $O(n + I)$.

We next count the number of inversions in an array in which every element is within k slots of its proper position.

Lemma 8 *If every element is within k slots of its proper position, then there are at most $O(nk)$ inversions.*

Proof. We provide an upper bound on the number of inversions. Consider some particular element, $A[i]$. There are at most $4k$ elements that can be inverted with $A[i]$,

in particular those elements in the range $A[i-2k\,..\,i+2k]$. Therefore, i is a part of at most $4k$ inversions, and hence there are at most $4nk$ inversions.

From this we conclude that the running time of insertion sort on an array in which every element is within k slots of its proper position is $O(nk)$.

As a side note, it seems possible to prove this directly, without using inversions, by showing that the inner loop of insertion sort never moves an element more than k slots. However, this is not as easy as it seems: even though an element is always *begins* within k slots of its final position, it is necessary to show that it never moves farther away. For example, what if it moves $k+2$ slots backwards, and then is later moved 3 slots forward? However, perhaps one can show that an element never moves more than $4k$ slots.

(c) Show that sorting a list in which each paper is within k slots of its proper position takes $\Omega(n \lg k)$ comparisons. *Hint:* Use the decision-tree technique.

Solution: We already know that sorting the array requires $\Omega(n \lg n)$ comparisons. If $k > n/2$, then $n \lg n = \Omega(n \lg k)$, and the proof is complete. For the remainder of this proof, assume that $k \leq n/2$.

Our goal is to provide a lower-bound on the number of leaves in a decision tree for an algorithm that sorts an array in which every element is within k slots of its proper position. We therefore provide a lower bound on the number of possible permutations that satisfy this condition.

First, break the array of size n into $\lfloor n/k \rfloor$ blocks, each of size k, and the remainder of size $n \pmod k$. For each block, there exist $k!$ permutations, resulting in at least $(k!)^{\lfloor n/k \rfloor}$ total permutations of the entire array. None of these permutations move an element more than k slots.

Notice that this undercounts the total number of permutations, since no element moves from one k-element block to another, and we ignore permutations of elements in the remainder block.

We therefore conclude that the decision tree has at least $(k!)^{\lfloor n/k \rfloor}$ leaves. Since the decision tree is a binary tree, we can then conclude that the height of the decision tree is

$$\begin{aligned}
&\geq \lg\left((k!)^{\lfloor n/k \rfloor}\right) \\
&\geq \left\lfloor \frac{n}{k} \right\rfloor \lg(k!) \\
&\geq \left\lfloor \frac{n}{k} \right\rfloor (c_1 k \lg k) \\
&\geq c_1(n-k)\lg k \\
&\geq \frac{c_1 n \lg k}{2} \\
&= \Omega(n \lg k)
\end{aligned}$$

(The last step follows because of our assumption that $k \leq n/2$.)

(d) Devise an algorithm that matches the lower bound, i.e., sorts a list in which each paper is within k slots of its proper position in $\Theta(n \lg k)$ time. *Hint:* See Exercise 6.5-8 on page 142 of CLRS.

Solution: For the solution to this problem, we are going to use a heap. We assume that we have a heap with the following subroutines:

- MAKE-HEAP() returns a new empty heap.
- INSERT$(H, \textit{key}, \textit{value})$ inserts the key/value pair into the heap.
- EXTRACT-MIN(H) removes the key/value pair with the smallest key from the heap, returning the value.

First, consider the problem of merging t sorted lists. Assume we have lists $A_1, \ldots, A_t$, each of which is sorted. We use the following strategy (pseudocode below):

1. Make a new heap, H.
2. For each of the t lists, insert the first element into the list. For list i, perform INSERT$(H, A_t[1], i)$.
3. Repeat n times:
   (a) Remove the smallest element from the heap using EXTRACT-MIN(H). Let v be the value returned, which is the identity of the list.
   (b) Put the extracted element from list v in order in a new array.
   (c) Insert the next element from the list v.

The key invariant is to show that after every iteration of the loop, the heap contains the smallest element in every list. (We omit a formal induction proof, as the question only asked you to devise an algorithm.) Notice that each EXTRACT-MIN and INSERT operation requires $O(\lg k)$ time, since there are never more than $2k$ elements in the heap. The loop requires only a constant amount of other work, and is repeated n times, resulting in $O(n \lg k)$ running time.

In order to apply this to our problem, we consider A as a set of sorted lists. In particular, notice that $A[i] < A[i + 2k + 1]$, for all $i \leq n - 2k - 1$: the element at slot i can at most move forwards during sorting to slot $i + k$ and the element at slot $i + 2k + 1$ can at most move backwards during sorting to slot $i + k + 1$.

For the moment, assume that n is divisble by $2k$. We consider the the $t = 2k$ lists defined as follows:

$$\begin{aligned} A_1 &= \Big[A[1], A[2k+1], A[4k+1], \ldots A[n-(2k-1)]\Big] \\ A_2 &= \Big[A[2], A[2k+2], A[4k+2], \ldots A[n-(2k-2)]\Big] \\ \ldots A_t &= \Big[A[2k], A[4k], A[6k], \ldots, A[n]\Big] \end{aligned}$$

Each of these lists is sorted, and each is of size $\leq n$. Therefore, we can sort these lists in $O(n \lg k)$ time using the procedure above.

We now present the more precise pseudocode:

```
SORT-ALMOST-SORTED(A, n, k)  ▷ Sort A if every element is within k slots of its proper position.
1  H ← MAKE-HEAP()
2  for i ← 1 to 2k
3       do INSERT(H, A[i], i)
4  for i ← 1 to n
5       do j ← EXTRACT-MIN(H)
6          B[i] ← A[j]
7          if j + 2k ≤ n
8            then INSERT(H, A[j + 2k], j)
9  return B
```

Recall that a heap is generally used to store a key and its associated value, even though we often ignore the value when describing the heap operations. In this case, the value is an index j, while the key is the element $A[j]$. As a result, the heap returns the index of the next smallest element in the array.

Correctness and performance follow from the argument above.

Notice that there is a second way of solving this problem. Recall that we already know how to merge two sorted lists that (jointly) contain n elements in $O(n)$ time. It is possible, then to merge the lists in a *tournament*. We give an example for $k = 8$, where $A + B$ means to merge lists A and B:

$$\begin{aligned}
&\text{Round 1:} \quad (A_1 + A_2),\ (A_3 + A_4),\ (A_5 + A_6),\ (A_7 + A_8) \\
&\text{Round 2:} \quad (A_1 + A_2 + A_3 + A_4),\ (A_5 + A_6 + A_7 + A_8) \\
&\text{Round 3:} \quad (A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8)
\end{aligned}$$

Notice that there are $\lg k$ merge steps, each of which merges n elements (dispersed through up to k lists) and hence has a cost of $O(n)$. This leads to the desired running time of $O(n \lg k)$.

Problem 2-3. Weighted Median.

For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the ***weighted (lower) median*** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}\,.$$

(a) Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of $x_1, x_2, \ldots, x_n$ with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

Solution: Let x_k be the median of $x_1, x_2, \ldots, x_n$. By the definition of median, x_k is larger than exactly $\left\lfloor \frac{n+1}{2} \right\rfloor - 1$ other elements x_i. Then the sum of the weights of elements less than x_k is

$$\begin{aligned}
\sum_{x_i < x_k} w_i &= \frac{1}{n} \cdot \left(\left\lfloor \frac{n+1}{2} \right\rfloor - 1 \right) \\
&= \frac{1}{n} \cdot \left\lfloor \frac{n-1}{2} \right\rfloor \\
&\leq \frac{n-1}{2n} \\
&< \frac{n}{2n} \\
&< \frac{1}{2}
\end{aligned}$$

Since all the elements are distinct, x_k is also smaller than exactly $n - \left\lfloor \frac{n+1}{2} \right\rfloor$ other elements. Therefore

$$\begin{aligned}
\sum_{x_i > x_k} w_i &= \frac{1}{n} \cdot \left(n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) \\
&= 1 - \frac{1}{n} \cdot \left\lfloor \frac{n+1}{2} \right\rfloor \\
&\leq 1 - \left(\frac{1}{n} \right) \left(\frac{n}{2} \right) \\
&\leq \frac{1}{2}
\end{aligned}$$

Therefore by the definition of weighted median, x_k is also the weighted median.

(b) Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

Solution: To compute the weighted median of n elements, we sort the elements and then sum up the weights of the elements until we have found the median.

```
1  WEIGHTED-MEDIAN(A)
2  k ← 1
3  s ← 0                          ▷ s = Total weight of all x_i < x_k
4  while s + w_k < 1/2
5        do  s ← s + w_k
6            k ← k + 1
7  return x_k
```

The loop invariant of this algorithm is that s is the sum of the weights of all elements less than x_k:

$$s = \sum_{x_i < x_k} w_i$$

We prove this is true by induction. The base case is true because in the first iteration $s = 0$. Since the list is sorted, for all $i < k$, $x_i < x_k$. By induction, s is correct because in every iteration through the loop s increases by the weight of the next element.

The loop is guaranteed to terminate because the sum of the weights of all elements is 1. We prove that when the loop terminates x_k is the weighted median using the definition of weighted median.

Let s' be the value of s at the start of the next to last iteration of the loop: $s = s' + w_{k-1}$. Since the next to last iteration did not meet the termination condition, we know

$$\begin{aligned} s' + w_{k-1} &< \frac{1}{2} \\ \sum_{x_i < x_k} w_i = s &< \frac{1}{2} \end{aligned}$$

Note that if the loop has zero iterations this is still true since $s = 0 < \frac{1}{2}$. This proves the first condition for being a weighted median. Next we prove the second condition.

The sum of the weights of elements greater than x_k is

$$\sum_{x_i > x_k} w_i = 1 - \left(\sum_{x_i < x_k} w_i \right) - w_k = 1 - s - w_k$$

By the loop termination condition,

$$\begin{aligned} \sum_{x_i < x_k} w_i = s &\geq \frac{1}{2} - w_k \\ -s &\leq -\frac{1}{2} + w_k \end{aligned}$$

$$\begin{aligned} 1 - s - w_i &\leq \frac{1}{2} \\ \sum_{x_i > x_k} w_i &\leq \frac{1}{2} \end{aligned}$$

Thus x_k also satisfies the second condition for being the weighted median. Therefore x_k is the median and the algorithm is correct.

The running time of the algorithm is the time required to sort the array plus the time required to find the median. We can sort the array in $O(n \lg n)$ time. The loop in WEIGHTED-MEDIAN has $O(n)$ iterations requiring $O(1)$ time per iteration, so the overall running time is $O(n \lg n)$.

(c) Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3 of CLRS.

Solution: The weighted median can be computed in $\Theta(n)$ worst case time given a $\Theta(n)$ time median algorithm. The basic strategy is similar to a binary search: the algorithm computes the median and recurses on the half of the input that contains the weighted median.

```
1  LINEAR-TIME-WEIGHTED-MEDIAN(A, l)
2  n ← length[A]
3  m ← MEDIAN(A)
4  B ← ∅                          ▷ B = {A[i] < m}
5  C ← ∅                          ▷ C = {A[i] ≥ m}
6  w_B ← 0                        ▷ w_B = total weight of B
7  if length[A] = 1
8     then return A[1]
9  for i ← 1 to n
10       do if A[i] < m
11             then w_B ← w_B + w_i
12                  Append A[i] to array B
13             else Append A[i] to array C
14 if l + w_B > 1/2               ▷ Weighted median ∈ B
15    then LINEAR-TIME-WEIGHTED-MEDIAN(B, l)
16    else LINEAR-TIME-WEIGHTED-MEDIAN(C, w_B)
```

The initial call to this algorithm is LINEAR-TIME-WEIGHTED-MEDIAN$(A, 0)$. In this algorithm, A is an array that contains the median of the initial input and l is the total weight of the elements of the initial input that are less than all the elements of A. B contains all elements less than the median, C contains all elements greater or equal to the median, and w_B is the total weight of the elements in B.

To prove this algorithm is correct, we show that the following precondition holds for every recursive call: the weighted median y of the initial A is always present in the recursive calls of A, and l is the total weight of all elements x_i less than all the the elements of A. This precondition is trivially true for the initial call. We prove that the precondition is also true in every recursive call by induction. Assume for induction that the precondition is true. First let us consider the case in which $l + w_B > \frac{1}{2}$. Since y must be in A, at line 14 y must be either in B or C. Since the total weight of all elements less than any element in C is greater than $1/2$, then by definition the weighted median cannot be in C, so it must be in B. Furthermore, we have not discarded any elements less than any element in B, so l is correct and the precondition is satisfied.

If $l + w_B \leq \frac{1}{2}$ on line 14, then y must be in C. All elements of C are greater than all elements of B, so the total weight of the elements less than the elements of C is $l+w_B$ and the precondition of the recursive call is also satisfied. Therefore by induction the precondition is always true.

This algorithm always terminates because the size of A decreases for every recursive call. When the algorithm terminates, the result is correct. Since the weighted median is always in A, then when only one element remains it must be the weighted median.

The algorithm runs in $\Theta(n)$ time. Computing the median and splitting A into B and C takes $\Theta(n)$ time. Each recursive call reduces the size of the array from n to $\lceil n/2 \rceil$. Therefore the recurrence is $T(n) = T(n/2) + \Theta(n) = \Theta(n)$.

(d) The ***post-office location problem*** is defined as follows. We are given n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i\, d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

Argue that the weighted median is a best solution for the one-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

Solution:

We argue that the solution to the one-dimensional post-office location problem is the weighted median of the points. The objective of the post-office location problem is to choose p to minimize the cost

$$c(p) = \sum_{i=1}^{n} w_i d(p, p_i)$$

We can rewrite $c(p)$ as the sum of the cost contributed by points less than p and points greater than p:

$$c(p) = \left(\sum_{p_i < p} w_i (p - p_i) \right) + \left(\sum_{p_i > p} w_i (p_i - p) \right)$$

Note that if $p = p_k$ for some k, then that point does not contribute to the cost. This cost function is continuous because $\lim_{p \to x} c(p) = c(x)$ for all x. To find the minima of this function, we take the derivative with respect to p:

$$\frac{dc}{dp} = \left(\sum_{p_i < p} w_i \right) - \left(\sum_{p_i > p} w_i \right)$$

Note that this derivative is undefined where $p = p_i$ for some i because the left- and right-hand limits of $c(p)$ differ. Note also that $\frac{dc}{dp}$ is a non-decreasing function because as p increases, the number of points $p_i < p$ cannot decrease. Note that $\frac{dc}{dp} < 0$ for $p < \min(p_1, p_2, \ldots, p_n)$ and $\frac{dc}{dp} > 0$ for $p > \max(p_1, p_2, \ldots, p_n)$. Therefore there is some point p^* such that $\frac{dc}{dp} \leq 0$ for points $p < p^*$ and $\frac{dc}{dp} \geq 0$ for points $p > p^*$, and this point is a global minimum. We show that the weighted median y is such a point. For all points $p < y$ where p is not the weighted median and $p \neq p_i$ for some i,

$$\sum_{p_i < p} w_i < \sum_{p_i > p} w_i$$

This implies that $\frac{dc}{dp} < 0$. Similarly, for points $p > y$ where p is not the weighted median and $p \neq p_i$ for some i,

$$\sum_{p_i < p} w_i > \sum_{p_i > p} w_i$$

This implies that $\frac{dc}{dp} > 0$. For the cases where $p = p_i$ for some i and $p \neq y$, both the left- and right-hand limits of $\frac{dc}{dp}$ always have the same sign so the same argument applies. Therefore $c(p) > c(y)$ for all p that are not the weighted median, so the weighted median y is a global minimum.

(e) Find the best solution for the two-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Solution:

Solving the 2-dimensional post-office location problem using Manhattan distance is equivalent to solving the one-dimensional post-office location problem separately for each dimension. Let the solution be $p = (p_x, p_y)$. Notice that using Manhattan distance we can write the cost function as the sum of two one-dimensional post-office location cost functions as follows:

$$g(p) = \left(\sum_{i=1}^{n} w_i |x_i - p_x| \right) + \left(\sum_{i=1}^{n} w_i |y_i - p_y| \right)$$

Notice also that $\frac{\delta g}{\delta p_x}$ does not depend on the y coordinates of the input points and has exactly the same form as $\frac{dc}{dp}$ from the previous part using only the x coordinates as input. Similarly, $\frac{\delta g}{\delta p_x}$ depends only on the y coordinate. Therefore to minimize $g(p)$, we can minimize the cost for the two dimensions independently. The optimal solution to the two dimensional problem is to let p_x be the solution to the one-dimensional post-office location problem for inputs $x_1, x_2, \ldots, x_n$, and p_y be the solution to the one-dimensional post-office location problem for inputs $y_1, y_2, \ldots, y_n$.